Reverse-engineering state machine diagrams from legacy C-code

van Zeeland, D.H.A.

Award date:
2009

Reverse-engineering state machine diagrams from legacy C-code

D.H.A. van Zeeland

Master's thesis
Department of Mathematics & Computer Science
Software Engineering & Technology
Eindhoven University of Technology

Reverse-engineering state machine diagrams from legacy C-code

D.H.A. van Zeeland

Master's thesis March 30, 2009
Department of Mathematics & Computer Science
Software Engineering & Technology Group
Eindhoven University of Technology

Reverse-engineering state machine diagrams from legacy C-code

D.H.A. van Zeeland

Department of Mathematics & Computer Science
Software Engineering & Technology Group
P.O. Box 513, 5600 MB Eindhoven, The Netherlands.

Master's thesis

Abstract

The Unified Modeling Language (UML) is the de facto standard modeling language in software development. Currently models are specifically generated during the design phase of software projects, however they are also useful when a software system enters its maintenance phase. For many software systems availability of accurate documentation is scarce. To better understand existing legacy systems we need to extract dynamic behavior and represent this behavior graphically in models.

New challenges in UML diagram extraction arise when additional UML diagrams are considered. In this report we present ongoing work on extracting state machines from legacy C code, motivated by the popularity of state machine models in embedded software. Via reverse engineering we extract state machines from source code. To validate the approach we consider an approximately ten-years old embedded system provided by the industrial partner. The system lacks up-to-date documentation and is reported hard to maintain.

During this project, a prototype that allows extracting state machine diagram from legacy C source code was successfully built. The created diagrams in XMI-format can be visualized by commercial CASE-tools, like Enterprise Architect, which makes it possible to remodel the state machines. The tool further generates the state machine diagrams as pictures.

We observe that state machines can be automatically derived from legacy C source code, even from a non-object-oriented code. The approach has proven to be very successful in this case study and is, in general, promising.

Key words:
Cpp2XMI, Reverse Engineering, State Machine Diagrams, State Machine extraction, UML, Unified Modeling Language, Legacy Code, C, C++, Realtime System, Embedded System, Columbus/CAN, CppML, Graphviz, CASE-tool.

Acknowledgments

This thesis is the result of a nine-month internship at Vanderlande Industries, as well as a finalization of the masters program in computer science and engineering at the Eindhoven University of Technology. Numerous people have helped me to successfully complete this project, and I am greatly indebted to all of them.

First, I would like to thank my supervisor from the Eindhoven University of Technology (TU/e), Mark van den Brand. I enjoyed discussing the technical problems I encountered and I am grateful for all the hints and tips he gave me. Furthermore, Mark gave me the freedom to shape my research and commented several versions of my thesis, for which I owe him much gratitude.

Additionally I am grateful to my daily supervisor at Vanderlande Industries, Peter v/d Weijden. During my internship he helped me when and where needed. I furthermore appreciate the efforts of Robert Bijl, also of Vanderlande Industries, for making this internship possible and his practical input.

I would like to thank Mark van den Brand, Alexander Serebrenik, Serguei Roubtsov and Peter van der Weijden for being a member of my assessment committee.

The project was carried out at the Express Parcel group of the Research and Development department of Vanderlande Industries in collaboration with Laboratory for Quality Software and the Software Engineering and Technology group of the TU/e. I enjoyed the collaboration with all of these groups and I would like to thank all the people for their hospitality and kindness.

Last but not least, I would like to thank my family for their support throughout my studies and my friends for the wonderful years.

Dennie van Zeeland
Eindhoven, March 2009

Contents

1. Introduction

The graduation project in this thesis has been carried out at Vanderlande Industries (VI) and focuses on reverse engineering state machine diagrams from legacy C-code. In this chapter we will first give an introduction to VI, where this research was performed during the last ten months. A brief history of the company will be given. Furthermore, we will discuss the Express Parcel group of the Research & Development department. Next we will provide a list of some of the terminology used throughout this thesis. Section 1.3 contains a brief summary of the Flow System Controller (FSC). Gaining more insight on this software system is desirable, because FSC is the software system for which the documentation needed to be generated. In Section 1.4 of this introduction a problem description will be given. We will conclude this introduction with an overview of this thesis.

1.1. Vanderlande Industries

1.1.1. Company Profile of VI

VI is one of the leading companies in worldwide material handling systems (MHS). Integrated automated material handling systems that the company constructs ranges from small and medium-sized systems up to the largest currently in use throughout the world. VI being a company that specializes in Warehousing & Distribution, Baggage Handling and Express Parceling. VI is able to occupy and maintain one of the leading positions in the market. VI is a global player with offices in all key regions of the world. The company's headquarters is located in the Netherlands, but VI also possesses Customer Centers in Germany, France, Great Britain, Spain, China, South Africa and the USA. These Customer Centers maintain direct contact with customers and handle all key business functions, like for instance providing support or doing maintenance.
VI offers innovative solutions based on a wide range of products and concepts. A broad range of standard solutions is available for smaller and medium-sized companies. For larger companies, VI has proven its ability to provide tailored solutions based on its broad range of standard products. The use of these products maximizes the availability of a system during its total lifetime.
VI is a project-driven organization, which indicates that, when one of the customer centers sells a project, the rest of the organization will start to realize this order. Most of the preliminary work is done at the main location in Veghel. Also supporting departments

such as Research and Development (R&D), Engineering and Manufacturing are based in Veghel.
In the distribution segment, VI provides automated solutions for storage, order picking, consolidation and warehouse control in the distribution supply chain. For the express parcel industry the company offers a wide range of sorting systems for both parcels and letter post. VI builds systems in small local depots up to the worlds largest postal sorting centers. It also supplies baggage handling systems for airports around the globe. The systems are able to provide fast and safe storage, transportation and sorting of departing, transfer and arriving luggage. In this business segment, VI is among the worlds top three suppliers.

1.1.2. Historical overview of VI

In 1949, Vanderlande Industries was established in Veghel, the Netherlands. Eddy van der Lande founded Machinefabriek E. van der Lande as a general machinery and construction company. In the beginning, the company was dedicated to the service and repair of weaving looms and other machinery for the textile industry. Due to the increasing need for material handling in the market, the company introduced the 'LandeLift' which was developed to ease the problem of lifting heavy items from loading bays. With the 'LandeLift', the company achieved its first introduction into the basic MHS's and shortly after, a petrol driven through-belt conveyor was introduced. It was mounted onto a movable carriage that enabled bulk material to be loaded and unloaded.
In 1962, the company joined forces with the American organization 'the Rapids Standard Company'. This partnership ensured acceleration of the growth in the field of material handling. The name of the company then changed into 'Rapistan Lande'. In the late 1980's, a management buy-out led to the separation of Rapistan, and all the shares returned to Dutch ownership. The company's name changed into Vanderlande Industries, as which it is currently still known. The final episode leading to the VI company as we know it today was the integration of the German software company 'GamBit' in 1997. With this integration VI was able to develop in-house Material Flow Controllers (MFC) and Warehouse Management Systems (WMS).

1.1.3. Research & Development

VI consists of several departments, each with their own task in successfully completing projects. Research & Development (R&D) is one of these departments. One of their tasks is to develop new products for all different kinds of market segments. Therefore it is essential to know the solutions of competitors and create new and innovative solutions that are requested by the market. You could call R&D a gatekeeper to new developments in the world. Besides development of new products, improvement and maintenance of current products is also a task of R&D.
The R&D department consists of several groups. For each market (Express Parcel, Dis-

tribution and Baggage Handling) exists a group within R&D. Together with Feasibility & Technology, Product Lifecycle Support, Innovation Centre and Special Design Engineering group, they form the R&D department. Our research was performed in the FSC-team of the Express Parcel group. The FSC-team focuses on the development and maintenance of Flow System Controller (FSC).

1.2. Terminology

In this section the terminology used throughout this thesis is listed. The most commonly used abbreviations can be found in in Table 1.1

Abbreviation	Meaning
API	Application Programming Interface
ASG	Abstract Syntax Graph
AST	Abstract Syntax Tree
CASE	Computer-Aided Software Engineering
CppML	C++ Markup Language
FSC	Flow System Controller
GUI	Graphical User Interface
IO	Input / Output
LaQuSo	Laboratory for Quality Software
OO	Object-Oriented
PLC	Programmable Logical Controller
QSM	Query-driven State Merging (algorithm)
SLoC	Source Lines of Code
SM	State Machine
STD	State Machine Diagram (UML 2.x) / State chart Diagram (UML 1.x)
UML	Unified Modeling Language
VI	Vanderlande Industries
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Table 1.1.: Abbreviations

1.3. FSC

1.3.1. General information about FSC

The FSC is a universal software product developed by Vanderlande Industries. Approximately 500 FSC controllers are sold worldwide. FSC is built on the following pillars: standardized modules, flexibility, and tracking. Understanding these pillars is necessary to understand how FSC can be used. Standardized modules implies that FSC is not based on a specific project. FSC is able to control systems by using standard, reusable control units. FSC sends signals to severals actuators, like motors, and receives signals from sensors, like photocells. Flexibility implies that the system can be assembled from nearly an unlimited number of control components. All of these components are glued together in a FSC configuration. Tracking is the basis for the control of a sorting system. In FSC, the term tracking is defined as 'constantly having an accurate view of all parcel positions along the FSC controlled conveyor system'. Tracking is required for the correct execution of parcel position related actions, such as parcel data collection (scanning) and action points (divert positions). The correct timing and proper execution of the actions makes FSC a real-time system. Hence, it is not desirable to miss deadlines. By missing deadlines the correct functioning of the system is not guaranteed.

1.3.2. Communication of FSC with two layers

We can distinguish four layers in a conveyor system (Figure 1.1). The bottom layer is formed by the mechanical parts, such as conveyors. These parts are passive and driven by electro-mechanical parts in a higher layer, like motors or switches. These parts, called devices, form the device layer. These devices are connected to a controller, in this case FSC. FSC receives information from photocells and position indicators. FSC uses the received information to control other devices like motors. Sometimes, it is necessary to translate parcel data, like data from the labels on the parcels, into physical destinations. This kind of translation is volatile and customer specific. Thus, the translation depends on the customer. A host is the link between FSC and the customer. This model shows

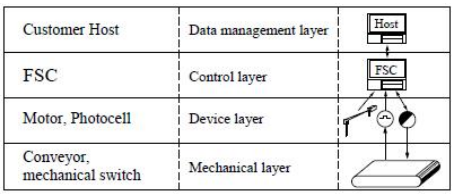

Figure 1.1.: Layer model of a conveyor system

that FSC communicates with two layers. The top layer provides information from a wider

context. Because of the flexibility of FSC, FSC needs to be configured to communicate with the system, i.e. the device layer and the mechanical layer.

1.3.3. FSC versus PLC

FSC is a software solution that runs on standard industrial computer hardware. It is an application that runs on top of the real-time QNX operating system. This is what distinguishes it from the other standard control solution in the market, i.e. Programmable Logic Controllers, PLCs. VI also works with PLCs, especially in the baggage market segment. However, FSC is the standard way to control the automated material handling systems in the parcel express market.
Each material handling system requires a different PLC to be tailored to suit the system and several PLC libraries need to be glued together to implement the system that controls the material handling system. FSC is more flexible. It is a basic software solution, which can be configured by commissioning engineers to adjust to the needs of customers. Every site has its own configuration, but they all use FSC to interpret the configuration and to control the hardware equipment.
The FSC system consists of the FSC runtime system component and the Graphical User Interface (GUI) component. The GUI component was added later to the FSC. The total package is still called FSC. The GUI can connect to multiple FSCs, but only to one FSC simultaneously. The FSC can be connected to multiple GUIs. The GUI shows diagnostics of the entire equipment. Furthermore, it contains buttons to start, stop and reset the system. Besides these buttons, the GUI contains screens where the FSC can be configured. In PLC the GUI consists of simple displays. Input/Output (I/O) is basically carried out via switches and lights.

1.3.4. Interfaces of FSC

The FSC uses a Customer Host to find the destination of the handled parcel and manipulates I/O to deliver the parcel to the requested destination. This all is carried out via standard interfaces. FSC also interfaces with equipment like weighing scales, telecoding, etc. An overview of the deployment of FSC can be found in Figure 1.2.

1.3.5. Object-oriented programming

The development of FSC started approximately 15 years ago, and nowadays VI has over 12 years of 'operational' experience with FSC. This has led to new insights, and therefore FSC is still in development. An outcome of these new insights is the development of a new version, FSC NG. Over 100 man-years were put into the development of the current FSC versions.

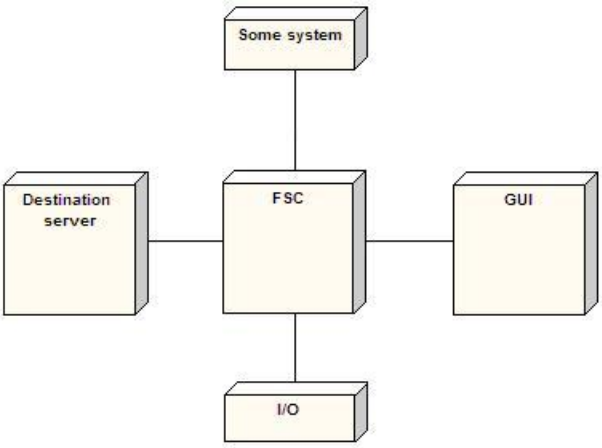

Figure 1.2.: The deployment environment of the FSC

Though FSC was implemented in C, it is based on an object-oriented model. The configuration of FSC therefore represents the real-world system. The software was initially written in C and some parts are migrated to C++. Currently, new functionality is completely written in C++. The C and C++ implementation are comparable. Where in C++ a method is used, in C a procedure is used that starts with the class name and has a reference to the runtime data as first parameter. An object coded in C can be found in Listing 1. An object written in C++ can be seen in Listing 2. The corresponding class diagram is displayed in Figure 1.3. There are more similarities between C++ and the object-oriented flavored C. However, C++ is more suitable for object-oriented programming.

Listing 1 Object-oriented C code for the Motor Object

```
1 MTR_create();
2 MTR_destroy();
3 MTR_exit();
4 MTR_init();
5 MTR_reset();
```

FSC has a few methods to implement construction and destruction of runtime objects. These are:

- Creation using `Object_create()`
- Destruction using `Object_remove()`
- Initialization using `Object_init()`
- Closing using `Object_exit()`

So the `Object_create()` and `Object_remove()` functions implement the constructor and destructor of runtime objects, respectively.

Listing 2 C++ code for the Motor Object

```
1  class CMotor: public CFscObject
2  {
3     public:
4        CMotor();
5        ~CMotor();
6        bool Exit();
7        bool Init();
8        void Reset();
9     private:
10       // Behaviour of the motor
11       // class defined here
12 }
```

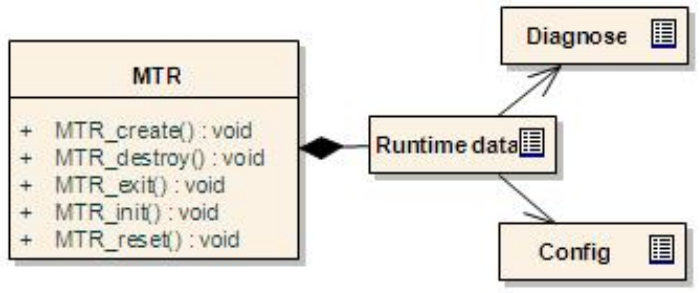

Figure 1.3.: Class diagram of the Motor Object

1.3.6. FSC Events and Timers

We will now briefly discuss the way FSC communicates with the lower and higher levels. This is done via events. FSC events are used to notify other system parts that something has happened. System parts interested in this event attach themselves to this event with the `event_attach(id, *cb_function())` method. When the trigger method is called, the callback functions of the attached system parts are called later.
We can define roughly 4 kind of event-triggers. The first and most important cause for triggering events is displacement, i.e. tracking. When packages reach certain predefined positions in the system, events are triggered to take corresponding actions. Another source for triggering events is by using timers. Timers are set to cause actions to be triggered at specific intervals or at specific times. Timers trigger timed events. The set method sets the delay until the event is triggered. A third way of triggering events is caused by the setted Input and Output (I/O). I/O-values can raise interrupts. These interrupts need to be handled by the system, and appropriate actions need to be taken to handle the interrupt. A last cause that triggers events is when serial data becomes available. In case serial data is available, a trigger is sent to the subscribers. A subscriber then needs to take action to retrieve the data.

1.4. Problem Description

In an iterative development process documentation becomes out-of-date very soon. Modifications in design, e.g. by enhancements or refactoring, automatically decrease the accuracy the current documentation. In particular during maintenance it can be a big problem to understand the code.
Goal of this MSc project is to create a tool that generates documentation of the dynamic behavior from source code of FSC. FSC is a software system that controls material handling systems produced by Vanderlande (see Section 1.3). Together with the domain experts it was decided to generate state machine diagrams from source code to gain more insights into the dynamic behavior of the runtime equipment control code of FSC.
Version 5 and 6 of FSC are still being used by customers all over the world. Thus maintenance of these current versions is still a key task of the FSC-team. The outcome of the project aids in improving these maintenance tasks by generating new documentation.
In juli 2008, the kick-off for FSC Next Generation (FSC NG) took place. FSC NG is rewritten from scratch using FSC Version 5 & 6 as a starting point. The results of this project could be used as input for FSC NG.
In conclusion, the reverse engineering of state machine diagrams from existing software could form a significant advantage for maintenance and development of current and new versions of FSC. The information should document the current functions and capabilities of FSC, and can be used during the re-design of next versions of FSC.

1.5. Thesis Outline

In Chapter 2 of this report, we will focus on the related work concerning extraction of state machines from software systems. Furthermore, we will inspect some parser generators and fact extractors that are needed for the reverse engineering process. In Chapter 3, more insight is given on how to detect state machines in source code. In that chapter we will discuss the possible patterns that could be used to implement state machines. Furthermore we will describe the state machine patterns that are being used in FSC. Chapter 4 describes the adjustments that were made to the original Cpp2XMI tool [KPvdBM06] to implement the state machine diagram extractor. These adjustments consist of an additional CppML-parser, including the search for state machine patterns, changes to the storage, a new layout generator and some extra changes to the XMI-writer. Chapter 5 discusses the results of the case study of Cpp2XMI applied to FSC. We will finish this report with conclusions and future work that could be performed in the field of state machine extraction in legacy C-code.

2. Related work and used tools

Reverse engineering dynamic behavior for systems is a hot topic. Several studies exist on reverse engineering state machines. However, it is worthy to note that a lot of work exists on static model reverse engineering, in particular class diagrams and that dynamic model reverse engineering is often left aside in reverse engineering tools. In this chapter, we will focus on related work concerning state machines extraction for software systems. Furthermore we will discuss tools that are needed in the process of reverse engineering state machines.

2.1. Dynamic analysis techniques for extracting State Machines

Walkinshaw et al. [WBHS07] created a technique that uses a Query-driven State Merging (QSM) algorithm to reverse engineer a state machine from a software system. The reverse engineering starts with a dynamic analysis, in which a collection of system traces is generated. Subsequently these results will be minimized to a level of abstraction that can be interpreted by analysts, and generates a sequence of abstraction functions. The third step is to perform trace abstraction. This activity applies the abstraction function to the set of system traces. The output will be used in the last step, where the QSM algorithm is applied to this output.
Their approach does not automatically extracts state machines, but requires human intervention. Furthermore their implementation for this technique is carried out with the Eclipse Test and Performance Tools Platform. This framework only provides a range of tools for the dynamic analysis of Java programs.

In [GZ05] a method for obtaining state machines for objects from sequence diagrams using an existing method for state chart synthesis [ZHJ04] is proposed. Their method consists of building sequence diagrams using dynamic analysis, in contrast to static analysis. Traces of program executions are analyzed to describe the behavior of each execution path of a program. The difficulty of this approach is to combine the basic sequence diagrams correctly in order to reflect the general behavior of the program. Subsequently, they complement the dynamic approach with the static analyses of the source code of the program.
Once sequence diagrams have been generated, they produce statecharts automatically. In [ZHJ04], an algebraic approach to revisit the problem of statecharts synthesis in the context of UML v2.0 is revisited. This approach is used to generate statecharts. The approach of Guéhéneuc et al. [GZ05] consists of dynamic analysis, by using logging

of program traces. However for some realtime systems it is not possible to generate logging of program traces, since the additional logging could lead to deadline misses and hence breaking the realtime behavior. Another disadvantage of this approach is that we can never be certain whether we executed all possible paths through our program. Furthermore, this approach only works for object-oriented source code. Their prototype uses Caffeine, a tool for the dynamic analysis of Java programs, to generate traces of program executions describing the behavior of each thread of a program.

Yuan et al. [YXM06] propose another method for extraction state machines for a system, called Brastra. Their approach for automatically extracting state machines from unit-test executions, abstracts the concrete state of an object by using the branch coverage exercised by methods invoked on the object. Their method involves running JUnit test classes and using the path trace file after program executions as input for their tool. Their tool post-processes the collected path trace file to collect branch coverage information. They constructed an abstraction function to map concrete states to abstract states. This approach only works if there are unit-tests available for the legacy system. In our situation, this is not the case. Furthermore, their tool only works for object-oriented languages, as it depends on tools that generate test-cases that only can generate them for object-oriented languages. Currently, the tool only supports Java language based systems. Another drawback is that the set of unit-tests need to be complete in order to generate an accurate state machine.

We can conclude that all dynamic analysis approaches have drawbacks. Either they are not suitable for C/C++ source code, or they use logging or execution traces as input for the state machine. We cannot use log information for the analysis of the software system of our industrial partner, since it is a realtime system, and we do not wish to break the realtime properties of the system. Furthermore, we are never sure if we have generated enough log-information to assure that we can properly extract the complete state machine from the log files.
Therefore, we prefer to look directly at the source code of our legacy software system. Via static analysis we want to reverse engineer state machine diagrams from source code. In the next section we will look further into existing approaches for the reverse engineering of state machine diagrams from source via static analysis.

2.2. Static analysis techniques for extracting State Machines

Walkinshaw et al. [WBAH08] propose a method to reverse engineer state machines by using symbolic execution and program conditioning of source code. Their technique can not only reverse engineer state transitions but also annotate each transition with its respective source code: the state transition function. The technique is based upon the observation that the start and exit points of a state transition function can usually be mapped to particular syntax elements of the source code. A state transition function for a state chart transition $A \rightarrow^{f()} B$ consists of the source code that is executed between the

entry and exit points of method $f()$, when it is called in the execution context represented by state A.
Program conditioning identifies those statements that are executed, provided that certain conditions expressed as constraints on program variables at particular program points are true. Given a transition $A \rightarrow^{f()} B$, if A and B can be encoded as conditions on the program variables at particular points during its execution, a program conditioner will remove those programming constructs that do not contribute to the process of reaching B from A. It will identify the transition function $f()$. A program conditioner determines which statement to retain or remove by symbolic execution.
Discovering state transitions and their functions can be done by four steps:

- Manually identify state transition points in terms of the source code syntax.
- Construct the symbolic execution tree, marking all symbolic execution states that correspond to transition points.
- Map marked transitions points to abstract machine states.
- Identify state transitions between the abstract states by detecting consecutive marked transition points in the symbolic execution tree.

The implementation of this approach is designed to compute the possible state transitions and transition functions of Java systems. It uses the Java PathFinder explicit state model checker and its symbolic execution extension to construct and traverse the symbolic execution tree. This makes their technique not suitable for our case study. Our approach differs from theirs that we do not need to manually identify state transition points. Furthermore this method assumes that transitions map to functions. In our approach that does not need to be the case.

One of the primary goals of the approach by Corbett et al., called Bandera [CDH$^+$00], is to provide automated support for the model-construction and error trace interpretation techniques. Bandera uses slicing to automate irrelevant component elimination, abstract interpretation to support data abstraction, and a model-generator that allows significant bounds for various system components. Bandera takes as input Java source code and generates a program model in the input language of one of several existing verification tools. Bandera also maps verifier outputs back to the original source code.
The approach of Corbett et al. works on Java only. Furthermore it requires manual interaction to produce the state machine models, since it requires human input to determine which parts of the source code can be eliminated, i.e. slicing. In contrast to the Bandera approach, we do not need human intervention in the process of extracting state machine diagrams from source code. Furthermore, our approach works on C/C++ source code instead of Java. Our approach requires more restrictions on the source code that needs to be analysed, as opposed to the Bandera approach.

Another proposed method for extracting state machines is described in [Mos07] and [MH02]. Here, they try to extract state machines from PLEX (Programming Language for EXchanges) source code for telecommunications systems. The approach is to look

for certain patterns inside the source code, and resembles our approach. They defined a heuristic that allows to determine whether a program contains a state machine or not. If a program contains a state machine, an algorithm is run that allows to extract and re-assemble the original state machine on 'architecture level'.
Moslers system, however, only works with PLEX source code, which has a different structure than regular programming languages. Hence, this static analysis technique is not suitable for the reverse engineering of state machines from our the legacy software system of our industrial partner.

Knor et al. proposed a technique to reverse engineer state machine from C/C++ source via pattern searching, and then forward engineer these state machines into state machines with C++ generic components [KTW98]. Their approach consists of manually searching for state machines in the source code. With the help of the Enhanced String Pattern Recognition Tool (ESPaRT) they search for state machines in the source code. The tool needs manually defined patterns that ESPaRT uses. These patterns need to hand-fed to ESPaRT. ESPaRT's pattern language is based on strings, but it is enhanced by commands, that overcome the restrictions of regular expressions by adding features of syntactic tools. ESPaRT is invoked by specifying a pattern file and the source code files. When a pattern is found, the line numbers and names of the files containing the match will be printed.
The drawback of their approach is that it does not automatically look for these state machine patterns. It requires that source code patterns are provided to the tool. Our approach differs from theirs in that we automatically search for these state machine patterns. We do this by having a fixed set of patterns to look for, which could be automatically recognized in the source code. The ESPaRT-tool does not allow us to encode the specific state machine pattern that is used in FSC, because it is not possible to dynamically adjust the search-patterns. This adjustment is needed for checking whether conditions of the switch are used as assignments in the inner-case blocks.

All the above approaches use static analysis of the source code to extract state machines from source code. However, all the approaches have drawbacks or other issues that make them unsuitable for the reverse engineering of state machines from our legacy software system. Therefore we have chosen to develop our own tool that reverse engineers state machine diagrams from this legacy C/C++ source code.

2.3. Parsers Generators

We have chosen to build our own tool that reverse engineers state machine diagrams form legacy C/C++ source code. In order to do this we first need to process the source code and find state machine patterns. One way of doing this is to use a parser.
A parser generator is a program that produces a parser for a certain language. The language is defined by its grammar, which is given as input to the parser generator. The main advantage is that you can define your own parser, and that it can be determined which information is going to be extracted. We will discuss some parser generator. More

parser generators are discussed in Section 3.3.1 of [KPvdBM06].

The ASF+SDF Meta-Environment [vdBHdJ+01] can be used for writing formal specifications for some problem, developing your own language, transforming programs in some existing programming language into other languages. In the interactive development environment you can enter these specifications, which can then be used to analyze for instance source code. The ASF+SDF Meta-Environment includes a parser generator and has a formalism to specify transformation rules to produce UML diagrams in XMI format. Unfortunately, there was no sufficiently complete grammar for C or C++ at the time we started the project. Hence it was not possible to use the parser generator from the ASF+SDF Meta-Environment to generate a parser that could process our source code, and helps us in the reverse engineering process.

One of the most popular lexical analyzer and parser generators is JavaCC (Java Compiler Compiler) /citeKODA04. JavaCC can create parsers for any programming language. JavaCC needs a grammar specification to provide Java classes that can parse any source that matches to that grammar. We do not use JavaCC because of its minimal documentation on how to write grammars.

So there are still few complete grammars available for C and C++, and developing our own grammars would take up too much time. Hence, we have chosen to use complete and ready tools, that somehow can generate data that we can use as a basis for finding state machine patterns. Fact extractors could fulfill these requirements. These fact extractors will be discussed in the next section.

2.4. Fact Extractors

Building our own parser would take up too much time. Therefore we decided to use tools that are available, for instance by using existing parsers or fact extractor. Fact extractors extract certain facts from source code and put them into an intermediate format that could function as input for our tool. The disadvantage is that these tools only work for a particular language. However, it saves us the time of defining our own language. In this section we will discuss some well known C/C++ fact extractors. More fact extractors are discussed in Section 3.3.2 in [KPvdBM06] and Section 2.2 of [BJ06].

ELSA is a C and C++ parser [McP]. It is based on the Elkhound parser generator. It lexes and parses input C/C++ code into an Abstract Syntax Tree (AST). Furthermore it can do some type checking. ELSA parses C/C++ source code by using a Generalize Left-Reduce parser (GLR). It cleanly separates source code parsing from type-checking and disambiguation.
We choose not to integrate the ELSA-parser into our tool, because ELSA cannot handle parse errors. Thus, potential incorrect code stops the parsing process, and hence the reverse engineering of models from source code fails.

SolidFX [TV08] is a C++ fact extractor for C/C++, and is built on top of the ELSA parser. SolidFX extends the ELSA parser to handle parse errors by skipping from parsing the deepest scope where these errors occur. SolidFX outputs full Annotated Syntax Graphs (ASGs), complete with preprocessor data and line numbers. It is even possible to reconstruct the original source code from these ASGs. SolidFX provides an open query system that answers queries questions about the source code. The questions are constructed into a query tree that is applied to the fact-database, yielding a selection of the matches ASG nodes. SolidFX provides an ASG and an open query API to their fact database. This means that we could integrate solidFX into our tool.
However we choose not to do this, because of the licensing costs on the one hand, and because the output format of SolidFX does not suit our needs. Furthermore, we have a better alternative, to which we will come back later in this section.

EFES [BJ06], standing for ELSA Fact Extractor System, is a fact extractor constructed on the ELSA parser. However, the EFES improves the weaknesses of the ELSA parser, as can be read in Chapter 3 of [BJ06]. The EFES Fact Extractor works in four phases. In the first phase, the preprocessor reads and processes the input files and outputs a token stream. The parser reads this stream and builds up the Abstract Syntax Graph (ASG). In the second phase, all disambiguities are removed from the ASG. Furthermore the ASG is type checked. In the third phase the ASG is filtered. All information that is not of interest is removed, and a smaller ASG remains that contains all the needed information. In the last phase output is generated and written to a file.
Drawback of EFES is that it only supports a subset of the C grammar. Therefore we cannot use EFES in our tool.

Rigi is a system for understanding large information spaces such as code bases. The Rigi reverse engineering system [SWM97] is an interactive visualization tool of software structures. It analyzes the dependencies between software artifacts from the source code and extracts facts from a system and organizes these facts into higher level abstractions. The relations of procedures, procedure calls, data accesses, variables and among others data and control-flows are discovered and stored in Rigi graph model (Rigi Standard Format - RSF), and subsequently visualized in scalable hierarchical graph diagrams. RSF is an intermediate data format and processed by many reverse engineering tools. The Rigi reverse engineering system has the option to choose from different parsers.
However no parser that is capable to parse a mixture of C and C++ is available, hence we cannot use Rigi for our tool. We do need a parser that can handle the mixture of C and C++ source code, since FSC consist of mixture of C and C++ source code. Furthermore the RSF-format looses to much information about the source code. Therefore we cannot extract state machine diagrams from the RSF-format.

Columbus/CAN [RFMT02] is a fact extractor for C/C++, developed by the University of Szeged, Hungary. Currently, it is owned by the company FrontEndART. Columbus is a reverse engineering framework application that parses, analyzes, filters and exports information extracted from C/C++ source files into various formats, including the Abstract

Syntax Tree into an XML-like format called C++ Markup Language (CppML). Columbus/CAN comes with a C++ preprocessor (CANPP), a C++ Analyzer (CAN), the linker (CANLINK), and the exporter (ExportCPP). Columbus parses most C and C++ source code as valid constructs, including particular compiler dialects and slight deviations from the standard. It has an error recovery mechanism, which implies that incomplete source code, or syntax errors do not stop the parsing process. Columbus outputs various formats, such as CppML, XMI, GXL, HTML, RSF. The information about the source code structure is preserved in CppML. Columbus comes with good documentation, making it easy to understand and use. For an overview of the main advantages and drawbacks we refer to Section 4.3.2.1 of [KPvdBM06]. The major drawback is the amount of time and memory needed to parse source code. One of the major advantages is that it can extract the AST into an XML-like format, called CppML. This CppML could function as input into our tool.

We concluded that the Columbus parser could fulfill the requirements we have for a fact extractor, i.e. it deals with incomplete and/or incorrect source code. Hence it is fault-tolerant. It resolves cross-references in the source code correctly and completely. Columbus is able to preprocess source code of arbitrary complexity. Furthermore, Columbus generates enough information into the CppML format, like construct source code locations and types.
We decided to use the Columbus parser as a basis for our model extractor, which we will be discussing in the next section.

2.5. Columbus as a basis for a Model Extractor

In the previous section we have specified our choice for the Columbus parser. Especially the CppML output format, and the fault-tolerance of the Columbus parser are interesting features.
The next step in our reverse engineering process is to extract models from this intermediate format, called CppML. There are several type of models that are of interest, but as stated in Section 1.4, we are specifically interested in extracting state machines from source code. Our approach is to extract these models by finding state machine patterns in source code. What these patterns look like, and how we can detect them, is described in Chapter 3.
During our research we came across a tool called Cpp2XMI [KPvdBM06]. Cpp2XMI is a reverse engineering tool that is capable of extracting some UML diagrams from C++ source code. Cpp2XMI can extract Class, Sequence and Activity diagrams from C++ source code. It is also based upon the Columbus/Can framework. Cpp2XMI already integrates the preprocessing and parsing of source code and generating and exporting of models to XMI.
Cpp2XMI was developed by Elena Korshunova during her final project at the 'Ontwerpers Opleiding Technische Informatica' (OOTI) for the Laboratory for Quality Software (LaQuSo), Technical University of Eindhoven. Cpp2XMI was integrated into the analy-

sis, verification and validation toolset called Software Quality Analysis & Design Toolset (SQuADT). LaQuSo uses this toolset to analyze source codes provided by clients.
These above mentioned facts render Cpp2XMI to be an ideal candidate that serves as a basis for our state machine extractor. With this reverse engineering we want to capture the system behavior that emerges from the interactions occurring between the objects in the system. Furthermore, we could use the extracted diagrams for correspondence checking, which implies that the extracted models could be used to compare the implementation with the original design. Since Cpp2XMI already extracts multiple kinds of documentation, we could also check this generated documentation for similarities and differences with the original documentation.
Thus, with Cpp2XMI it is possible to derive the static structure and extract dynamic behavior of a system from potentially incomplete source code and represent them in XML Metadata Interchange (XMI) format. The tool is able to transform source code to UML diagrams in the XMI format automatically. Cpp2XMI is capable of extracting UML class, sequence and activity diagrams from C++ source code. We will be extending Cpp2XMI with the option to extract state machine diagrams from source code. In Chapter 4 we will further examine this extension of Cpp2XMI with state machine diagrams.

2.6. Summary

In this chapter we focused on related work concerning state machines extraction for software systems. We first looked at dynamic analysis techniques for extracting state machine diagrams. However, we concluded that dynamic analysis is not possible due to the fact that we are dealing with a realtime system. Hence we cannot merely add extra code to extract log-information. Furthermore, it can never be checked if we covered all possible states and transitions.
Furthermore, we discussed the static analysis techniques. All of these approaches are not suitable for the software system of our industrial partner. They either work on totally different programming languages, or they cannot deal with legacy C code and are only capable of extracting information from object-oriented languages (C++ / Java / C# / etc.). Therefore we decided to develop our own tool.
For the development of our tool we basically need to transform source code into models. For the first step in this process we need some type of parser. Due to various reasons we choose not to generate our own parser. We therefore opted to look at standard parsers and fact extractor. We concluded that the Columbus framework fits our needs.
We ended this chapter with information about Cpp2XMI. Cpp2XMI is a tool that also uses the Columbus framework. Hence we have chosen it as a basis for our state machine reverse engineering tool.

3. Detecting State Machines via Pattern Matching

In many realtime control systems we can identify state machines [Sim94]. The same is true for the FSC-software developed by Vanderlande Industries (VI). State machines can be identified at several levels of the automated material handling systems that FSC can control.
In this chapter we will discuss several aspects of the approach that was taken to extract state machine diagrams from source code. We will start with a general overview of the reverse engineering process. In Sections 3.2, 3.3 and 3.4 we will focus on the patterns that can be used to implement state machines. Section 3.3 focuses on the most often used pattern in FSC, i.e. the nested-choice patterns. In Section 3.5 we will describe step-by-step how the 'switch-within-switch'-pattern is constructed, and how we can extract the corresponding state machine from the source code. In Section 3.6 we will discuss the possibilities of adding more information to the state machine diagram. We will end this chapter with a summary.

3.1. General overview of methodology

As mentioned before, FSC contains several state machines. There are multiple abstraction levels at which we can identify these state machines. At the lowest level we can identify state machines of the devices of automated material handling systems, for instance the state of motors and electrical actuators. By combining these devices we create components like sorting machines, merging machines and conveyor belts. These components of material handling systems are state machine based. The highest level describes the overall state of a system and is basically the combination of state machines from the lower levels. Note that this leads to a state space explosion that is not preferred.
Together with the help of domain experts we identified state machines at the component-level. The experts indicated which source files belong to which component. Furthermore, they identified the components that are state machine based. In these instances we took a closer look into the source code belonging to that component. Via code inspection we identified a part of the source code that implements this state machine. This way we identified several nested-choice patterns in the source code of FSC that are used for implementing state machines.
The mentioned observations guided the reverse engineering of state machine diagrams. At first we try to extract states from source code. We do this by basically looking for

'switch-within-switch'-constructs in the source code, and by testing whether the case labels of the outer switch correspond to the elements of an declared enumerate, i.e. the state-enum. Then we try to reverse engineer the transitions between these states. This is done by looking at all assignments within the nested switches. Only the assignments to the state-variable are of interest (slicing). If we put these transitions between the correct states we get our basic state machine diagram.
However, we identified more source code constructs in FSC that indicate that a state machine could be implemented in the source code, i.e. state transitions inside functions, and multiple events that are handled the same way. Thus the next step was to capture the state machine of these special constructs. If we extend the basic state machine extraction with these special source code constructs, we get all the 'switch-within-switch' nested-choice pattern state machines from the FSC source code.
These extracted state machines look quite promising. On top of that we saw an opportunity to put more information into the state machine diagrams, like conditional transitions. To extract information about conditional transitions we detect the conditions that need to hold for a transitions to be able to perform. We do that by looking at the conditions of if-statements.

3.2. Patterns

We start by observing that, in their simplest form, UML state machines contain nothing but states and transitions connecting states. The transitions in these state machines are associated with events. At each moment in time the system can be in exactly one state. This implies that transitions are instantaneous. When an event occurs, the system should move to a new state. Note, that implementing a state machine behavior involves, therefore, a two-phase decision making:

- What is the current state of the system?
- What is the event to be handled?

Based on this simple observation, our approach consists of looking for nested-choice patterns, specifically the "switch-within-switch"-pattern. We will focus on this pattern because the domain experts indicated that this is the pattern that is most frequently used in the source code of FSC.

3.3. Nested choice patterns

For these simple state machines we can identify several nested-choice patterns. These nested-choice patterns have some common characteristics. In the next section we will

discuss them thoroughly.
One of these characteristics is that the states of the state machine are explicitly defined. This is being done in an enumerate structure in source code. The same holds for all possible events that could occur in the source code. Events are also explicitly defined and enumerated in the source code. We will call these enumerates the state- and event-enumerate respectively. The state machines defined by the nested-choice pattern can only consist of states and transitions from these enumerates.
Another characteristic of the nested choice patterns is that the system continuously handles incoming events. The events are coded by a separate variable, which could be passed into functions as a parameter.
Another trademark of this pattern is that the current state of the state machine is stored into a variable. This variable could be a member of some sort of structure/class. Assigning a new state to this variable is done by explicitly coding the new state to this variable. This is done by assigning an element of the state-enumerate to this variable, and not using other constructs, like `++` or `--` operations.
Last characteristic of this patterns is that the code that needs to execute in which state and on what event, is determined by the choice-statements. These could either be if-else or switch-case statements.
Earlier we mentioned the 'switch-within-switch'-pattern, which can be used in all situations, but it will mainly be used with state machines containing a large variety of states and a large number of different events that could occur within these states. Next to this pattern we can identify the related 'if-within-switch'-pattern. This pattern is mainly used for state machines that consist of a large diversity of states, and a small number of different events that could occur in these states. A third type of pattern is the 'switch-within-if'-pattern, that is used in state machines with a small number of states and a large number of events that could occur in these states. The last nested-choice pattern is that of the 'if-within-if'-pattern. This pattern can be used in situations with limited number of states and a limited number of events.
Note that it is possible to have a mixture of these nested-choice patterns. Furthermore we emphasize that it is possible to rewrite each of these patterns to another nested-choice pattern. The last remark we have to make is that the 'if-within-if'-construction is a general pattern in source code. Hence, we can find this pattern within source code frequently, but this does not imply that each occurrence of this pattern implements a state machine.

3.4. Other state machine patterns

There are more ways to implement state machines into source code. We have identified the following patterns and methods for implementing state machines.

- Jump tables: a pointer to a function pointer array can be used as the state variable. The function pointer array is called jump table and contains references to the event handler functions. When an input event occurs, the system indexes into the jump table, and invokes one of the table's functions. The index into the jump table is

derived from the system state (and often is the state variable itself). This technique works, but is not without problems. Furthermore it is advised to not use pointers in realtime critical systems [MRR04].

- Object-oriented approach: Each system state can be encoded into a separate object, but only one state object needs to exist at any given time. The current state of a system will be indicated by the object, which is called the current state object. The role of the system in response to an input event will be to direct the event to the current state object. The state object is responsible for handling all the input events, so the system can unconditionally give all the events to the object. This object will process the event, and then return control back to the system. When an event causes a state transition, the current state object transforms itself into a new state object, but without knowledge or intervention from the system. The system need not know what class the current state object represents, only that the object will handle all input events passed to it. In the course of handling events, the current state object's type will determine the state of the system. This type will change dynamically, but not for all events. Many events in a system will cause no state transitions, in which case no type mutation will occur.
- GOTO-statements: We can use GOTO statements to implement state machines. Every GOTO statement represents a transition from one state to another state. An identifier is used as the statement label representing each state. For each state, a superfluous pair of `begin`- and `end`-statements or brackets is used to group statements of that state into single statement. A superfluous GOTO statement jumping to the initial state is added because the program needs to explicitly specify the initial state. Also GOTO-statements are not an ideal way of implementing state machines [Dij68].
- Long jumps: The standard C library implements the `setjmp()` and `longjmp()` functions. The `setjmp()` function saves the current execution state of a program into a structure. The `longjmp()` function transfers the state of the program to the point of the called `setjmp()`-state. With this technique we can jump from one state to another state in the program. Hence, making it possible to implement state machines with these library functions.

Note that this list is not complete, there exists other ways to implement state machines.

3.5. Basic state machine pattern

In the previous sections we have seen what type of patterns can be used in source code to implement state machines in programs. The 'switch-within-switch' nested-choice pattern can easily be identified within the source code of FSC. In the FSC source code there are several files that handle the actions of a component of an automated material handling system. In general, we will assume that all the actions that a component can perform are

being implemented in one file. Therefore it suffices to look for patterns on a file-base level, specifically 'switch-within-switch'-patterns. In this section we will take an in-depth look at the identification of this nested-choice pattern by showing how the pattern looks like in source code and how the corresponding diagram looks like. We will do this step-by-step. Throughout these subsections we will use and extend an example to specify the basic state machine pattern.

3.5.1. States

In Listing 3 we see an example of the basic pattern for states. On lines 2-6 in the source code we see an enumerate which indicates all possible states an object can have. The declaration of the enumerate-type could be in the same file as where the switch is in, but it could also be in a separate header file.
Furthermore, we see that the switch-statement is very specific for this pattern (lines 8-17). The switch will check the current state a system has. Thus, the variable inside the condition of the switch (line 8) maps to the current state of the system. Characteristic for this pattern is that the case labels of the switch (lines 9, 11, 13) will be the elements of the state-enumerate (lines 2-6). These different case labels represent the possible states of the system. The union of the case labels of the state-switch equals the set of the elements of the state-enumerate. The default case block generally implements some sort of error-handling code. We cannot determine in what state the system is. Hence, it does not map to a source state. Therefore we neglect the default case in the outer switch.
In the state diagram these states simply map to rounded boxes, as we can see in the example of Figure 3.1.

Listing 3 States defined in source code

```
1 /* State of the object */
2 enum OBJ_STATE {
3     STATE_A,
4     STATE_B,
5     STATE_C
6 };
7 ...
8 switch (...) {
9     case STATE_A:
10        ...
11    case STATE_B:
12        ...
13    case STATE_C:
14        ...
15    default:
16        ...
17 }
```

Figure 3.1.: States drawn in a diagram

3.5.2. Transitions

Note that with the previous pattern we are able to represent states. However, we also need to model transitions. The source code contains an enumerate that lists all events that are possible in the state machine diagram. We can see an example of this in Listing 4. The events-enumerate is declared on lines 2-6.
Inside each case block of the state-switch (line 8) we find another switch statement (line 10 and 18). This is the event-switch and checks which block of source code needs to be executed by comparing the event variable with the case-labels (lines 11, 13, 19 and 21) of the event-switch. The block of source code that will be executed takes care of the handling of events. The case labels are elements of the event-enumerate. Note that it is not necessary that all elements of the event-enumerate are handled inside the case-blocks of the state-switch. Hence, there can be events that are not handled in a certain state, and therefore these events will not be coded as cases inside the event-switch. Furthermore each event-switch can catch all events by a default event-handler (line 23). This way the unspecified events in the event-switch are handled by the default event-handler.
In the diagram shown in Figure 3.2 we see that the states are connected by arrows. Each arrow represents a transitions. Note that from the source in the example we cannot determine the state in which the system will end, therefore we added a state unknown in which all events end. Furthermore we do not know if there are any transitions from STATE_C. That's why we have drawn a dotted arrow from STATE_C.

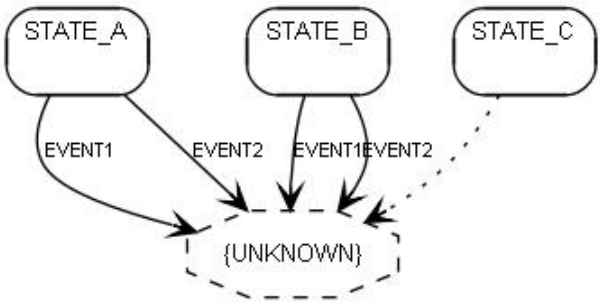

Figure 3.2.: Transitions drawn in a diagram

Listing 4 Transitions defined in source code

```
1 /* events to react on */
2 enum OBJ_EVENT {
3     EVENT1,
4     EVENT2,
5     EVENT3
6 };
7 ...
8 switch (...) {
9     case STATE_A:
10        switch (event) {
11            case EVENT1:
12                ...
13            case EVENT2:
14                ...
15        }
16        break;
17    case STATE_B:
18        switch (event) {
19            case EVENT1:
20                ...
21            case EVENT2:
22                ...
23            default:
24                ...
25        }
26        break;
27    ...
28 }
```

3.5.3. Object

Combining the detection of state and transition patterns forms the basis for the pattern recognition of state machines. However in FSC we further identified that the current state almost always belongs to an object. Hence in the source code we see objects containing a member variable to store the state. Note that this member variable stores a state of the state-enumerate type. In the legacy C code these objects are defined by means of structures (structs). In the newer object-oriented parts of the code of V5/V6 of FSC this is performed by means of a member variable which is stored in the class representing this object. We can basically map these structs to objects and vice versa for the FSC source code. An example of the legacy C code way is shown in Listing 5. Here we see that there is an OBJ-struct defined on line 6, which contains a member variable state on line 3 that stores the current state. In the source code we see that the condition of the state-switch on line 8 is the member variable of the struct or class. Furthermore we see that inside the case-blocks of the event-switch (lines 10-17 and 20-27) there are assignments to the state variable, of which we can see examples on lines 12, 15, 15 and 22. Hence this is the actual state transition. Note that this assignment can be surrounded by other code. Since the state variable is a value of type enumerate, we could think of other constructs to change the state, e.g. by using the `++` or `--` operator to switch between states. However in FSC this does not occur, and all state transitions are implemented in the source code

by explicitly writing an element from the enumerate type.
In the corresponding diagram that is shown in Figure 3.3, we see that the arrows now end in the corresponding states. Thus, if the systems current state is STATE_A and EVENT2 arrives, the systems is expected to enter STATE_C. In the diagram this transition between STATE_A and STATE_C is indicated by an arrow with label EVENT2. This is the mapping of the assignment on line 15.

Listing 5 Object structure in source code

```
1  typedef struct {
2      bool  initialised;
3      OBJ_STATE state;
4  } OBJ;
5  ...
6  OBJ obj;
7  ...
8  switch (obj->state) {
9      case STATE_A:
10         switch (event) {
11             case EVENT1:
12                 obj->state = STATE_B;
13                 break;
14             case EVENT2:
15                 obj->state = STATE_C;
16                 break;
17         }
18         break;
19     case STATE_B:
20         switch (event) {
21             case EVENT1:
22                 obj->state = STATE_B;
23                 break;
24             case EVENT2:
25                 obj->state = STATE_C;
26                 break;
27         }
28         break;
29     ...
30 }
```

3.5.4. Control Function

In FSC the state machine is normally handled by a control function. We refer to Listing 6 for an example. The control function, defined in the lines 1-25, must at least have one parameter, namely the event that needs to be handled, such as on line 2 in the example. However, in most cases in FSC there are at least two parameters. The event-handler parameter passes the event that has occurred to the control function. Inside the control function this parameter is checked by the event-switch that can be seen on lines 10 and 20 in the example. Thus this parameter determines which transition is performed in the state machine and is the same variable as the variable that is used in the condition of the event-switches.

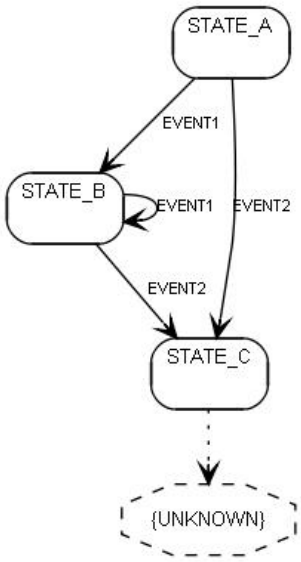

Figure 3.3.: Corresponding diagram of object structure in source code

Besides the event-handler parameter, there is also a parameter containing a reference to the object or a reference to some kind of variable that stores the current state of the system. An example of this is the first parameter of the control function. If it refers to an object, then this object has a state machine variable as a member. This object could be an object as is described in the section 3.5.3. Thus, it could be an instance of a struct-type like in the example in Listing 5 on line 6.
As mentioned in the introduction of this section, we focus on a file-based level and start to look for these kind of control functions. This implies that we are looking for local state machines as was discussed in Section 3.1. Due to these local patterns we cannot extract state machines from the overall system, or combined state machines. By using logging information, we could perhaps overcome this issue. For more information we refer to Section 6.3.
Note that this control function does not really influence our extracted diagrams. Hence the diagram remains the same as in Figure 3.3.

3.5.5. Multiple events are handled the same way

In all the previous examples, we assumed that several events have no overlapping code to get into a specific state. However we can think of two ways to introduce overlapping code to end up in the same state. The first method is to let the control function execute the same code for two or more events by leaving out the break statements inside the event-switch. An example of this method is shown in Listing 7 and the corresponding diagram in Figure 3.4. In this case when the system is in STATE_A, EVENT1 and EVENT2 are

Listing 6 Example: Control function

```
1  void OBJ_control(OBJ *obj,
2        OBJ_EVENT event) {
3     switch (obj->state) {
4        case STATE_A:
5        switch (event) {
6           case EVENT1:
7              obj->state = STATE_B;
8              break;
9           case EVENT2:
10             obj->state = STATE_C;
11             break;
12       }
13       break;
14    case STATE_B:
15       switch (event) {
16          case EVENT1:
17             obj->state = STATE_B;
18             break;
19          case EVENT2:
20             obj->state = STATE_C;
21             break;
22       }
23       break;
24    ...
25    }
26 }
```

handled the same way, because of the missing break or return statement between the cases of line 6 and 14. However, in STATE_B they execute different code. Hence we cannot merge EVENT1 and EVENT2, because EVENT1 exits due to a break statement on line 16. The break statement causes an immediate exit from the switch. Because cases just serve as labels, after the code for one case is done, execution falls through to the next unless explicit action to escape is taken. Break and return statements are the most common ways to leave a switch. Falling through allows several cases to be attached to a single action, like in the case of EVENT1 and EVENT2 in STATE_A on lines 4-10.

3.5.6. State transitions inside functions

Another way to introduce overlapping code to end in a specific state is by using functions. An example of this kind of overlapping code can be seen in Listing 8 and the corresponding diagram in Figure 3.5. Inside a case-block of the event-switch there is a call to a function, like on line 13. Somewhere in this function there is an assignment to the state-variable (line 3). Note that the function has a parameter to the object as we can see on line 1 of Listing 8. This need not always be the case. The key aspect is that the called function also has an assignment to the state variable.
One could think of situations where we get nesting of functions, or recursion of functions. This could lead to infinite loops. However, it need not always be the case that infinite loops do occur. But to make sure we do not get into infinite loops, we need to do some

Listing 7 Source code of multiple events that are being handled the same way

```
1  void OBJ_control(OBJ *obj,
2        OBJ_EVENT event) {
3     switch (obj->state) {
4        case STATE_A:
5           switch (event) {
6              case EVENT1:
7              case EVENT2:
8                 obj->state = STATE_C;
9                 break;
10          }
11          break;
12       case STATE_B:
13          switch (event) {
14             case EVENT1:
15                obj->state = STATE_B;
16                break;
17             case EVENT2:
18                obj->state = STATE_C;
19                break;
20          }
21          break;
22       ...
23       }
24 }
```

bookkeeping, and keep information about the already inspected functions. Thus, we do not get stuck into infinite loops. At the moment, our tool only looks at the first level function calls. So, only the directly called functions from within the case-block of the event-switch are being checked for an assignment to the state variable. This way we are sure that we cannot get into infinite loops when checking for assignments to the state variable.

3.6. Extracting additional information from source code

Thus far we have seen all sorts of constructs that can be used to implement a basic state machine. These constructs are reported by the experts to be the most common ones that are being used in FSC. Hence it made sense to first focus our attention on these common constructs. So we basically looked for 'switch-within-switch' patterns, and somewhere in the case blocks of the even-switch there needs to be an assignment to a state variable, which could be a member of an object.
However, we saw an opportunity to add more information into the models and identified some possible extensions. These possible extensions are:

- Conditional transitions
- OnEntry / OnExit / DoAction values
- Line / file mapping

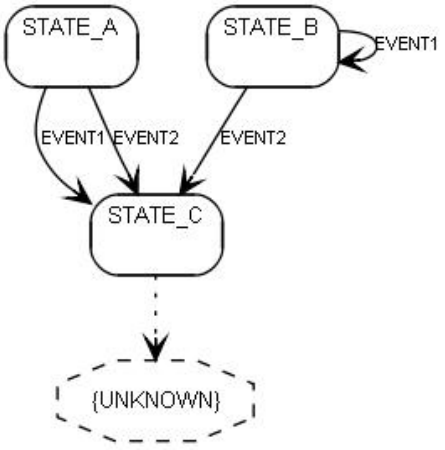

Figure 3.4.: Diagram of multiple events that are being handled the same way

- Overlap between state machines

From these we only implemented the conditional transitions into our tool.

Not all transitions that are coded in the source code do always have to take place. Some transitions will only take place if certain conditions are valid. Therefore our tool checks for the conditions which are needed for a transition to take place. See Listing 9 for an example of a conditional transition.
In the source code of that example we see an extra variable `y` on line 1. Inside the event-switch of EVENT1 of the event handler of STATE_A we see a conditional transition (lines 6-11). When EVENT1 occurs and the current state is STATE_A, it is unclear whether we end in STATE_B or STATE_C. This depends on the value of boolean `y` as is tested on line 6. Therefore we call the state transitions inside the case block of lines 5-12 conditional transitions: whether they occur depends on some factor, hence we cannot guarantee that this transition always occurs. It could be that, due to the value of the condition, the system does not execute a transition.
In the example we have a simple condition (line 6), namely a boolean `y`. However these conditions can consist of any condition that is allowed in C-code, so propositional equations consisting of a combination of booleans, comparisons with strings/integers, and function calls are allowed.
Currently, we only have implemented these conditional transitions in the picture export of the diagram. Due to the lack of time, these conditional transitions are not exported to the XMI-output yet. Hence, they cannot be visualized by CASE-tools. We can see the diagram corresponding to Listing 9 in Figure 3.6

More interesting additional information can be found in the use of OnEntry and OnExit values. This way we could add code preceding the assignment of the new state to the state variable to the OnExit value of the old state, and code that comes after this assignment

Listing 8 Source code of state transition inside a function

```
1  do_Function(OBJ *obj) {
2     ...
3     obj->state = STATE_B;
4     ...
5  }
6  ...
7  static void OBJ_control(OBJ *obj,
8        OBJ_EVENT event) {
9     switch (obj->state) {
10       case STATE_A:
11          switch (event) {
12             case EVENT1:
13                do_Function(obj);
14                break;
15             case EVENT2:
16                obj->state = STATE_C;
17                break;
18          }
19          break;
20       ...
21    }
22 }
```

Listing 9 Source code of conditional transitions

```
1  bool y = true;
2  switch (obj->state) {
3     case STATE_A:
4        switch (event) {
5           case EVENT1:
6              if (y) {
7                 obj->state = STATE_B;
8              }
9              else {
10                obj->state = STATE_C;
11             }
12             break;
13          case EVENT2:
14             obj->state = STATE_C;
15             break;
16       }
17       break;
18    case STATE_B:
19       switch (event) {
20          case EVENT1:
21             obj->state = STATE_B;
22             break;
23          case EVENT2:
24             obj->state = STATE_C;
25             break;
26       }
27       break;
28    ...
29 }
```

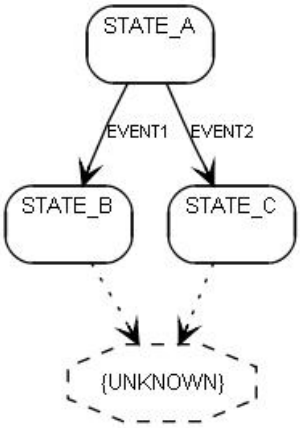

Figure 3.5.: Diagram of state transition inside a function

should be assigned to the OnEntry value of the new state. Furthermore, we could add the DoAction value to the transition. Again we could use the surrounding code of the assignment to the state variable as input for this DoAction value.

Other types of information we could add are references to files and line numbers. A final type of information that could be added to the diagram is a visual way to indicate overlap between state machine diagrams. This could be performed by looking for states with equal names, and common names of transitions. A remark should be made that if either state names or transition names do not differ a lot, this would result in large amount of overlaps.

3.7. Summary

In this chapter we have taken a closer look to the patterns that can be used to implement state machines. Furthermore, a closer look was taken to the pattern that is mainly used in the FSC source code. We discussed how the 'switch-within-switch'-patterns could be mapped onto state machine diagrams. We concluded with additional information that could be added to the extracted diagrams.

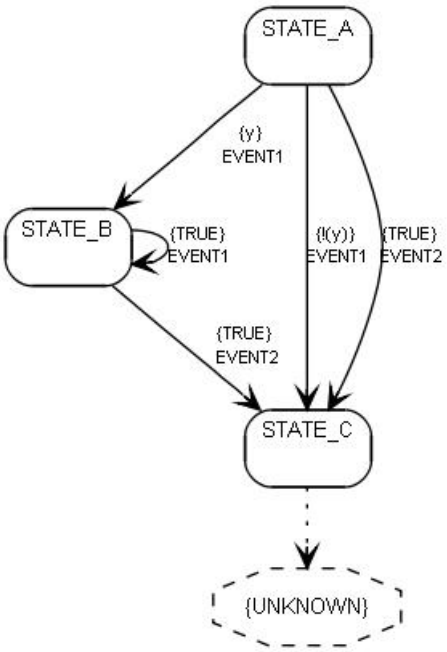

Figure 3.6.: Diagram showing conditional transitions

4. Architecture

The original tool created by Elena Korshunova [KPvdBM06] was capable to reverse engineer class, sequence and activity diagrams from C++ source code, and export these into the XMI-format. We added the option to reverse engineer state machine diagrams from C/C++ source code, using pattern matching. For a detailed description of these patterns we refer to Chapter 3.
In this chapter we discuss the architecture of Cpp2XMI. We will focus on the conceptual design that defines the structure and/or behavior of Cpp2XMI. We will discuss all the modules that are needed to reverse engineer state machine diagrams. However, our main focus in this chapter will be on the additional and extended modules of Cpp2XMI. For an overview of the rest of the unmodified modules we refer to Chapter 4 of [KPvdBM06].

4.1. General Design

In this section we will present a general design of the reverse engineering tool. The new version of Cpp2XMI uses the same basis of the general design as the original tool. This basis is described in Section 4.1 of [KPvdBM06]. In this design, we see several different modules, each having different functionalities. Thus, we achieve a decomposition of the main task into manageable, well-defined subtasks. The original version of the tool uses the Pipes and Filters architectural pattern. An advantage is that it allows us to easily extend the tool. The Pipes and Filters architectural pattern divides the task of a system into several sequential processing steps. In those kinds of systems, the flow of data through a system connects all these steps. *Filter* components implement processing steps. Every filter enriches, refines or transforms its input data.
The processing steps performed by the main modules composing a reverse engineering tool are shown in Figure 4.1. We will now take an in-depth look into these main modules.

4.1.1. Pre-processor

The pre-processor is a helper module that takes the C/C++ source code as an input and transforms it. For instance, it removes macros and transforms single backslashes in the `#include` statements into slashes. The standard Columbus/CAN pre-processsor (CANPP) cannot handle certain C/C++ constructs correctly. Therefore, we modify the source code in such a way that the pre-processor of the Columbus framework does accept it. We must remark that Columbus allows us to use different pre-processors. This module

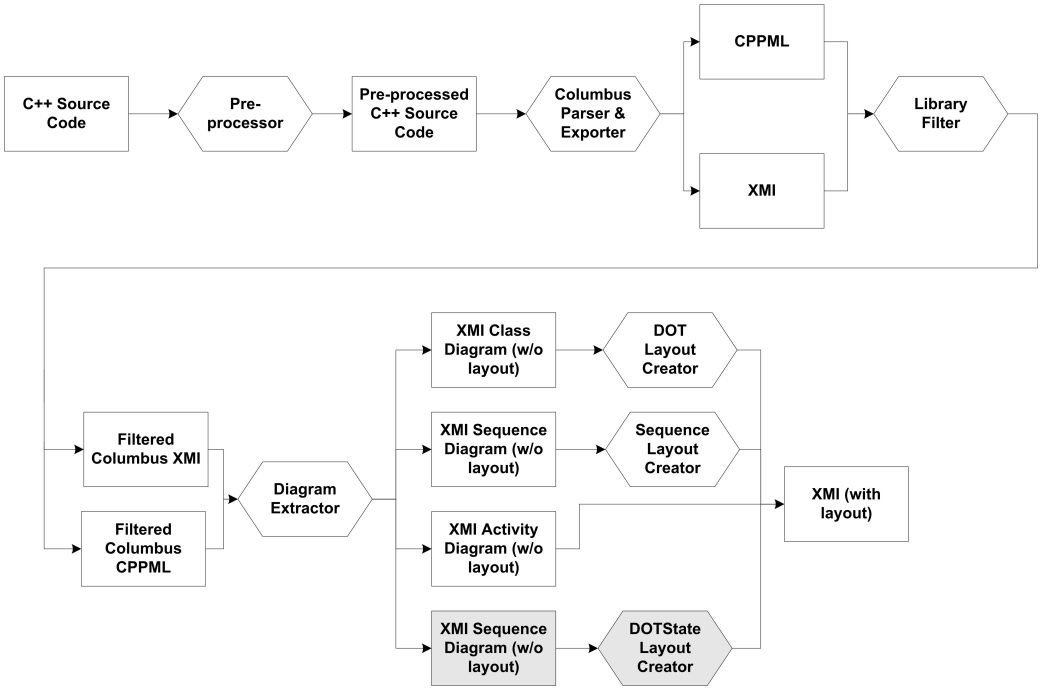

Figure 4.1.: Elaborated Design of the new version of Cpp2XMI

is implemented in the *Preprocessor*-package.
In conclusion we can note that the pre-processor transforms the source code into a form that can be processed by the Columbus framework.

4.1.2. Columbus parser and exporter

The Columbus parser and exporter parses the C/C++ source code. CAN is the analyzer of the Columbus framework. It analyses the preprocessed C/C++ source files and creates the internal representation file (object file). It builds up the Abstract Syntax Tree (AST) corresponding to this parsed source code. This all is stored in *.cst* files that contains all of the extracted information. This file is the equivalent of the object file used by compilers. The purpose of CANLink is to merge the separate internal representation files created by CAN into an *.mst* file. The purpose of the ExportCPP is to load the linker internal representation file created by CANLink and to export it into various formats. One of these formats is the C++ Markup Language, called CppML. The exporter outputs the AST into this CppML-format which basically is a XML-like format. Furthermore the exporter also produces class diagrams in the XML Metadata Interchange (XMI) format.

4.1.3. Library filter

The library filter removes all information concerning the standard libraries from the CppML and XMI outputs produced by the Columbus exporter. This way we can reduce the Columbus CppML output. This output has exploded due to the information obtained from standard C/C++ libraries. The information about the libraries is not relevant for the creation of the UML diagrams. Removing the information about the libraries reduces the CppML output significantly. This, in turn, improves the performance of the rest of our tool and improves the readability and understandability of the generated UML diagrams.
The Columbus framework does not have a filter option to remove this extra information. Therefore a standard filter mechanism was developed. This mechanism implements the Library Filter module and is coded in the *Filter*-package.

4.1.4. Diagram extractor

The main purpose of the diagram extractor module is to transform the filtered CppML output of the parser into a suitable format to represent UML diagrams, in this case a XMI document. It expands the XMI-document produced by the Columbus exporter with additional information about sequence and activity diagrams.
It first extracts the complete AST from the CppML file, and stores it into an internal structure. For instance information about objects allocated in the program and function calls are extracted from the Columbus CppML output. This information is needed for the generation of XMI tags for the sequence diagram.
Additionally, information about conditional and iterative statements is extracted from the internal structure that represents the AST. This information is needed to create XMI elements for the activity diagrams.
Note that this module also corrects the Columbus XMI output, because it has certain defects. These are further described in Appendix A of [KPvdBM06].
During this research we extended this module. Now this module is also capable of extracting state machine diagrams from source code via pattern matching. More information about these patterns can be found in Chapter 3. More information about the extension can be found in Sections 4.3 and 4.4 of this chapter.
Lastly, this module creates XMI tags for sequence, activity and state machine diagrams. In order to comprehend the transformation between source code and the UML class, sequence and activity diagrams, it is necessary to understand the correspondence between UML diagrams and object-oriented programming language concepts. In Section 4.2.5.1 till 4.2.5.3 of [KPvdBM06] the mapping between these are described.

4.1.5. Layout creator

The last modules in our reverse engineering process are the layout creators. After XMI with UML class and sequence diagrams is created, it is necessary to generate the layout

for these diagrams. This way we can visualize them in various CASE tools. The main goal of the layout creator module is to automatically generate layout for UML diagrams and store this position information in the *Diagram* element of XMI. The layout creators apply several layout algorithms to the UML diagrams in the XMI format.

If we compare the elaborated design of the extended version of Cpp2XMI with the elaborated design of the original version of Cpp2XMI (Figure 3 in [KPvdBM06]), we see that the diagram extractor has one more outgoing component, and that the new Cpp2XMI produces more output. In general, we added extra logic to the diagram extractor. This extra logic extracts state machine diagrams from the source code, and generates it as XMI state machine diagrams (without layout). Subsequently, the tool calls some other functions that add position information to the diagram, and output this all as XMI with layout.

In the next section, the logical view of our reverse engineering tool is presented. Furthermore we will go deeper into the extended and additional modules.

4.2. Logical View

In this section, the static structure of the system (classes, packages, operations, and their responsibilities) is explained. For the logical view of the original version of Cpp2XMI we refer to Section 4.2.1 of [KPvdBM06].
The reverse engineering system consisted of nine packages. We extended it with another package, i.e. the StateOrganizer. These packages were created based on the functionality of the classes. Thus, modification and further additions can be done easily. A brief description of these packages is given in Table 4.1.

Main classes and their relations are shown in Figures 5 and 6 in [KPvdBM06] and Figure 4.2. Only the most important entities are shown (for space reasons), with no indication of their properties and merely with methods for those entities that are of interest for state machine extraction. The `Store` class is an interface that allows storing or retrieving data from the internal structure. The main requirement on the internal structure is that it should contain all the information necessary for the reverse engineering algorithms to work. Thus, it plays a central role in the design of the reverse engineering tool. Furthermore the `App` class plays an important role too. The `App` class basically controls the reverse engineering process. It acts as the controller that sends the data through the filters of the reverse engineering process, as was discussed in Section 4.1.

4.3. The diagram extractor module

As stated before, we adapted several modules of the original Cpp2XMI tool. The biggest and most important changes were made to the diagram extractor module. A detailed

Package	Description
Main	• Contains the root class to initialize and run the system (Main class). • Depending on the parameters given by the user, the Main package initializes Preprocessor, ExecColumbus, SAXFilter, Parser, Writer, CollaborationOrganizer, StateOrganizer, Store, ClassLayout, DOTLayout, and SequenceLayout. If the user wants to generate only class diagrams, no other organizers not need to be created. And if users wants to generate only state machine diagrams, no other organizers are needed as well.
Columbus	• Contains the ExecColumbus class, which is responsible for the execution of Columbus command-line tools.
Preprocessor	• Changes C++ source code for the needs of Columbus. Also makes changes to anonymous structs.
Filter	• Filters irrelevant information from CppML (CppMLFilter) and XMI (XMLFilter) outputs of Columbus.
Parser	• Extracts necessary information from Columbus XMI/CppML files. • Encapsulates the extracted information into the Data class instance and passes it to the Store module. • Searches for state machine patterns and passes the states and events to the Store module.
Store	• Stores all data in the internal object structure.
Collaboration Organizer	• Gets data about attributes and function calls from the Store module. • Implements algorithms for the creation of objects and their messages. • Stores created objects and messages in the Store module.
Activity Organizer	• Gets data about various types of expressions (e.g. function calls, control statements, binary operations and variables) from the Store module. • Implements algorithms for the creation of activity nodes (actions and decisions) and transitions between them. • Stores created activity nodes and transitions in the Store module.
State Organizer	• Gets data about states and events from the Store module. • Implements algorithms for the creation of states and transitions. • Stores created state nodes and event transitions.
Writer	• Writes data from the Store module into the XMI file.
Layout	• Applies the ranking layout algorithm (ClassLayout) to create a layout for class diagrams. • Applies the DOT layout algorithm (DOTLayout) to create a more optimal layout for class diagrams. • Applies SequenceLayout algorithm to create a layout for sequence diagrams. • Applies the StateDot layout algorithm (StateDotLayout) to create a more optimal layout for state machine diagrams.

Table 4.1.: Package Responsibilities Description

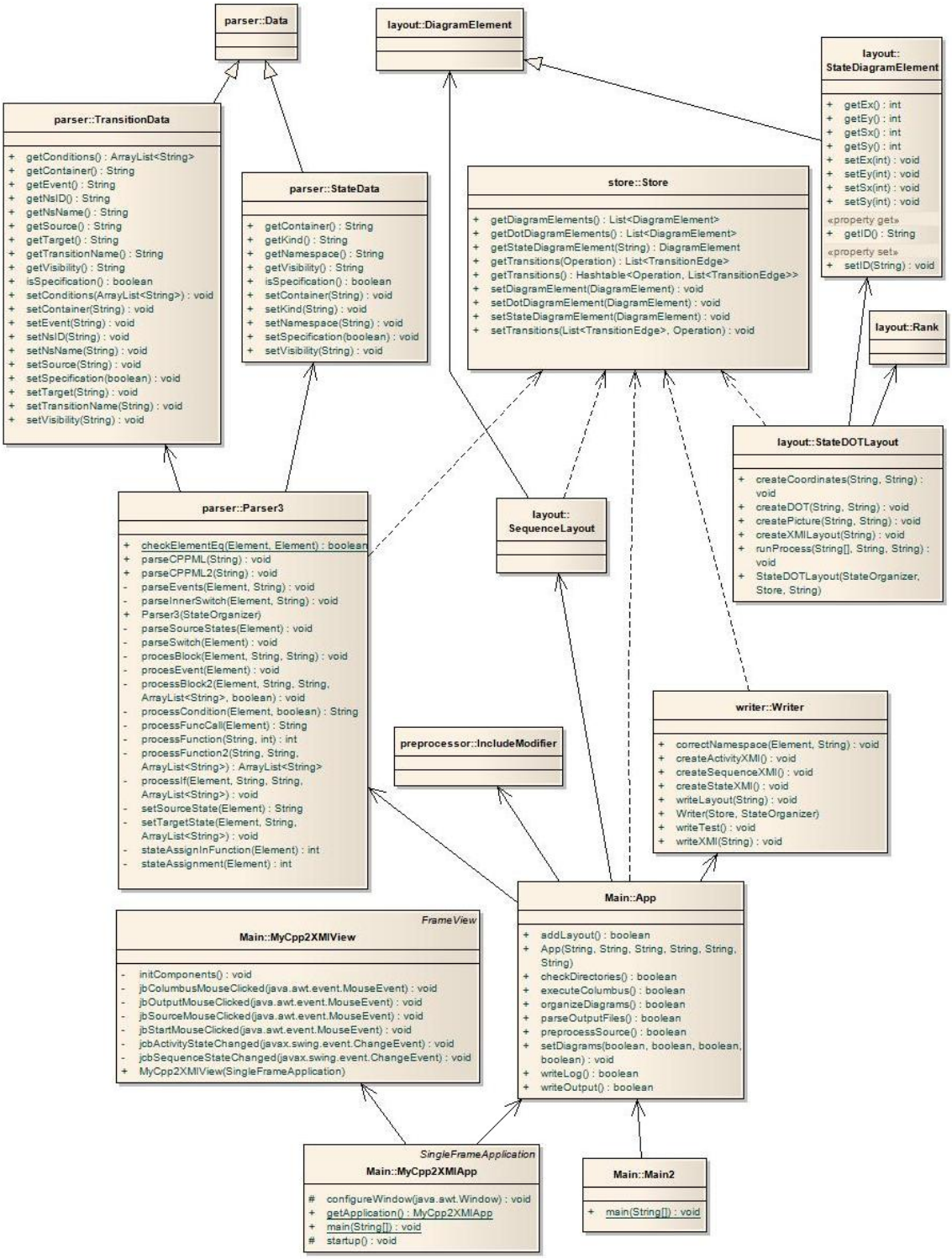

Figure 4.2.: Class Diagram

overview of the design of the diagram extractor is shown in Figure 4.3. We will focus on the adaptions and extensions in this reverse engineering process. In Figure 4.3 these are displayed by gray (rounded) boxes.

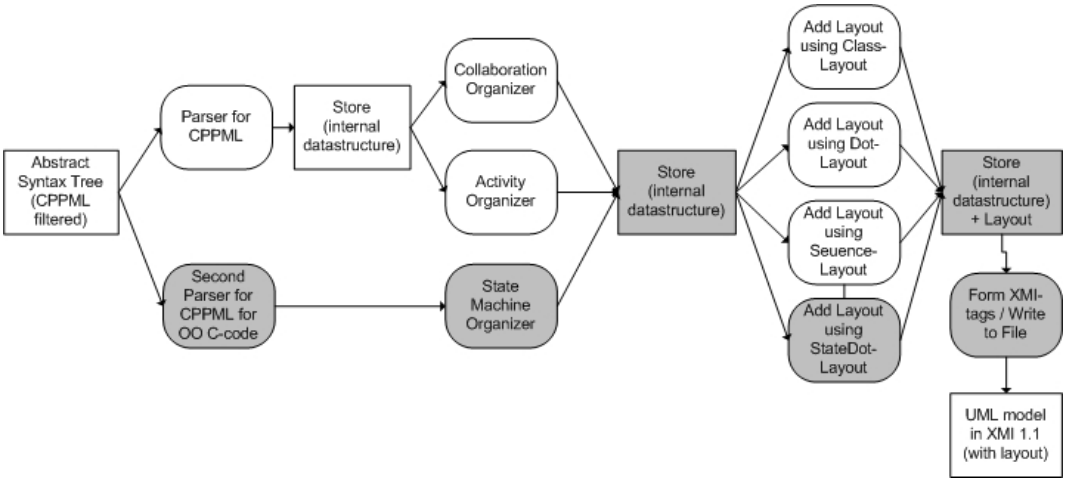

Figure 4.3.: Detailed view of the diagram extractor and layout generator

4.3.1. CppML Parser

The most important change to the design of Cpp2XMI is the addition of an extra parser. The original tool already had a CppML parser. However, this parser does not completely extract the information that we need to extract state machine diagrams. Therefore, we implemented our own parser for CppML. We could have chosen to adjust the CppML parser, but this would potentially break the reverse engineering of class, sequence and activity diagrams. This second CppML parser further abstracts itself from the C++ structure. Thus, this new parser does not look for object-oriented characteristics of C++, but it mainly focuses on finding state machine patterns in the AST. This parser traverses the AST and checks whether the branches of the AST can be mapped to the state machine patterns. If this is the case, state names are extracted from the AST. Furthermore we extract which transitions go from one state to another state and what event causes this transition to take place. All this information is stored in an internal data structure. First, we will take a closer look to the CppML format. Subsequently we will discuss the new CppML-parser more thoroughly.

CppML

Input for the diagram extractor module is a CppML file, which contains the AST of the C/C++ source code that is being reverse engineered. CppML is an XML format that contains all the information from this AST. CppML is produced by the ExportCpp tool, which is part of the Columbus/Can toolset. It has a structure according to the Columbus Schema. The original version of Cpp2XMI uses this as a basis to generate sequence and activity diagrams. The new version of Cpp2XMI also generates state machine diagrams using CppML as a basis. So with CppML we can represent the abstract syntax tree of C/C++ source code in a XML-like way. This representation was developed with the following goals in mind:

- Human readable: The representation must be easy to understand just by quickly glancing over the file.
- Easy to parse and process without the need for proprietary software
- Maybe a first step towards a C++ inter exchange format. Using CppML as a common format for C/C++ source code will enable independent analysis tools to be integrated and to exchange analysis information. A lot of discussion among researchers across numerous universities has made it clear that there is a need for a common format.

A smaller piece of C-code together with its CppML representation are shown in Listings 10 and 11. For a C++-example we refer to Figure 7 in [KPvdBM06].

Listing 10 Example C-program

```
1  typedef struct
2  {
3     char name[20];
4     int identifier;
5  } object;
6
7  void main (int argc, int *argv)
8  {
9     object o1;
10
11    strcpy(o1.name,"Object 1");
12    o1.id = 1;
13
14    printf ("Id of \%s is \%d",
15    o1.name, o1.id);
16 }
```

Additional Parser

The `Parser3` class is a new CppML parser, that is build to look for the state machine patterns. It traverses the AST and inspects it for possible state machines by examining

Listing 11 The CppML code that maps to the C-program

```
<struc:Class id="id101" name="#AnonStruct#">
   <struc:Object id="id109" name="identifier" />
<struc:Class>
<struc:Function id="id110" name="main">
   <struc:Parameter id="id111" name="argc" />
   <struc:Parameter id="id112" name="argv" />
   <struc:hasBody>
      <statm:Block id="id116">
         <expr:Assignment id="id128">
            <expr:MemberSelection id="id129">
               <expr:Id name="o1" id="id130" />
               <expr:Id name="id" id="id131" />
            </expr:MemberSelection>
            <expr:IntegerLiteral id="id132" value="1"/>
         </expr:Assignment>
         <expr:FunctionCall id="id133">
            <expr:Id name="printf" id="id134" />
            <expr:hasArguments>
               <expr:ExpressionList id="id135">
                  <expr:StringLiteral id="id136"
                           value="&quot;Id␣is␣\%d&quot;" />
                  <expr:MemberSelection id="id140">
                     <expr:Id name="o1" id="id141" />
                     <expr:Id name="id" id="id142" />
                  </expr:MemberSelection>
               </expr:ExpressionList>
            </expr:hasArguments>
         </expr:FunctionCall>
      </statm:Block>
   </struc:hasBody>
</struc:Function>
```

the branches of the tree for similarities to the state machine patterns as defined in Chapter 3.
For a global overview of the pattern finding process we refer to the pseudo code listed in Listing 12. This pseudo code describes the general algorithm that is used to traverse the AST and extract the state machine from source code.

Listing 12 Pseudo code of state machine finding process

```
1 Traverse AST and store all enumerates and structs
2 Traverse AST and store all function-branches
3 FOR all Control Functions with event parameter
4   IF there are Switch-statements inside these branches
5     FOR EACH Switch-statement {state-switch}
6       IF condition of the switch == variable of type enumerate (state variable)
7         FOR EACH case-block
8           IF there are Switch-statements inside these case-block
9             FOR EACH Switch-statement {event-switch}
10              IF condition of the switch == variable of type enumerate (event)
11                FOR EACH case-block
12                  get all assignments (including in first level functions)
13                  FOR EACH assignment
14                    IF left-hand-side of the assignment == state variable
15                      IF right-hand-side == element of an enumerate (state)
16                        store state and transition
```

4.3.2. State Organizer

The state machine organizer has functionality to store, and retrieve states. Furthermore, it checks for duplicate states, duplicate events between states, and basically performs all the logic that is needed for storing, building and checking of state machines. The state organizer class also implements a function that can check if states are end- or start-states. Thus, it can check for a certain state if it has incoming/outgoing transitions. If a state does not have outgoing transitions, we call it an end-state. If a state has no incoming transitions, we call it a start-state.
Additionally, this organizer transforms this internal state machine representation into a UML diagram by creating XMI tags without layout information. All this information is placed in the adjusted internal data structure store.

4.3.3. Storage

As stated before, we have adjusted several pieces of the original reverse engineering tool to extend Cpp2XMI with the option of reverse engineering state machine diagrams from legacy source code. We also modified the internal storage structure. In this section we will go deeper into these changes.
The new feature of Cpp2XMI is to extract state machine from source code by looking for state machine patterns. When we find such a pattern, we need to store it in this internal structure. Therefore we expanded the structure with functions for storing, retrieving and

searching of state machines, states and transitions.
We implemented two extra objects. One represents States, the other represents transitions. Furthermore we made an extra class `StateMachine` that holds these states and transitions and has methods for retrieval of these state machines. The instances of this StateMachine object represent the state machines that are found in the source code. With each state machine we store a name, which refers to the control function and file name in which the state machine is implemented.

4.4. The layout creator module

The new feature of Cpp2XMI is to extract state machines from source code. Like described in previous sections, we added logic to our tool that searches for state machine patterns. Furthermore we added extra storage options, to store states and transitions per state machine. The state machines are stored in the internal structure without layout information. Because there is no layout information available, all diagrams look messy when importing them into CASE tools. Therefore we extended the layout creator module with the option to add layout to the exported state machine diagrams.
First we need to export all state machines into an intermediate format, i.e. the DOT-format. Then we call the DOT-tool, which calculates position information. Next, we extract this position information, and store this in the internal storage. The last step of the layout creator module is to export all the information from the internal storage to a format suitable for importing into CASE tools.

The implemented layout-algorithms were inspired by the Class layout generator. Hence, we adopted the same approach for calculating position information as was done for calculating position information for the class diagrams via the DOT algorithm. Therefore we implemented a State-DOT-layout algorithm that is implemented in the `StateDOTLayout` class.
First, we output all state and transition data into a file in DOT format. To export these state machine diagrams we need to modify our writer class. At first we needed an extra method that outputs the data from the internal storage to a file in the DOT-layout format. For this step we iteratively get all state machines from a internal list and create separate DOT-files for each state machine. Each DOT-file contains the states and transitions between these states. We map each state to a node in the DOT-format, and we map each transition to an edge. For these steps we created `createPicture` method inside the `StateDOTLayout` class.
DOT is a diagram specification in a simple text format as can be seen in the example of Listing 13. With this format we can describe diagrams. DOT-files act as input for the DOT-tool, which generates diagrams in various output formats. The DOT-tool is part of the Grahpviz framework [EGK+03].
Subsequently Cpp2XMI feeds these output files to the DOT-tool to produce layout information in the PLAIN format. An example of the PLAIN output can be seen in Listing 14. From this PLAIN output we can extract the coordinates for the corresponding elements

of the diagram. The coordinates of nodes represent the coordinates of states, and edges are transitions between those states. The coordinates are extracted for all state elements of all state machine diagrams and are stored into the internal data structure. These coordinates can then be used to create the layout in the XMI format. We have to remark that this layout is CASE tool dependent and hence whether or not it is correctly interpreted depends on if the CASE tool understands the XMI-extension.
Furthermore we use the DOT-tool to generate pictures of the extracted state machines. Figure 4.4 shows the diagram that corresponds to the DOT file of Listing 13. When extracting state machine diagrams from source code the nodes in the diagram represent states, and the edges in the diagram represent transitions between these states.

Listing 13 Example of a DOT file

```
1 digraph G {
2   edge [fontname="Helvetica",fontsize=10];
3   node [fontname="Helvetica",fontsize=10, shape=Mrecord];
4   STATE_A [label="{STATE_A}", fontname="Helvetica", fontsize=10.0];
5   STATE_B [label="{STATE_B}", fontname="Helvetica", fontsize=10.0];
6   STATE_C [label="{STATE_C}", fontname="Helvetica", fontsize=10.0];
7   STATE_A -> STATE_B  [label="EVENT1", dir=back, arrowtail=none];
8   STATE_A -> STATE_C  [label="EVENT2", dir=back, arrowtail=none];
9   STATE_B -> STATE_B  [label="EVENT1", dir=back, arrowtail=none];
10  STATE_B -> STATE_C  [label="EVENT2", dir=back, arrowtail=none];
11 }
```

Listing 14 Example of a PLAIN file

```
1 graph 1.00 3.09 4.44
2   node STATE_A  2.69 4.19 0.81 0.50 "{STATE_A}" solid Mrecord
3   node STATE_B  0.40 2.38 0.81 0.50 "{STATE_B}" solid Mrecord
4   node STATE_C  2.44 0.58 0.81 0.50 "{STATE_C}" solid Mrecord
5   edge STATE_A STATE_B 4 2.37 ... 2.72 "{TRUE} EVENT1" 1.98 3.29 solid
6   edge STATE_A STATE_C 4 2.68 ... 0.97 "{TRUE} EVENT2" 2.75 2.38 solid
7   edge STATE_B STATE_B 7 0.81 ... 2.23 "{TRUE} EVENT1" 2.01 2.38 solid
8   edge STATE_B STATE_C 4 0.68 ... 0.93 "{TRUE} EVENT2" 1.77 1.48 solid
```

4.4.1. XMI writer

Besides exporting the state and transition data into DOT-files, we also export the state machine, states, transition data and the corresponding layout information into the XMI-export file and into a picture. This is the last step of the reverse engineering process. XMI-tags with position information are written into a file, which could then be viewed and edited by a CASE tool that is capable of viewing/importing XMI-files. This way the state machine diagrams can be imported by CASE tools. We have to remark that layout is CASE tool dependent and hence whether it is correctly interpreted depends on if the CASE tool understands the XMI-extension. To overcome this layout issues, we also implemented a jpg-exporter, which exports the state machine diagrams into a picture format that can be viewed with any ordinary image-viewer.

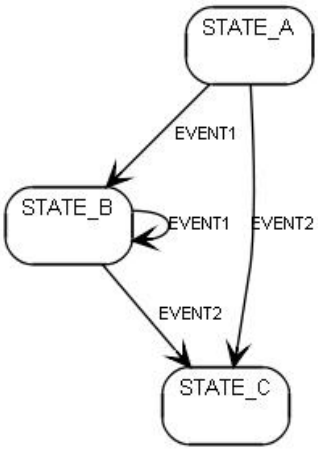

Figure 4.4.: Diagram after conversion of DOT file to a picture

To make this export possible we added the `createStateXMI` and `createPicture` methods and adjusted the `writeLayout` method. The `createStateXMI` method generates the internal structure into the XMI-format. The `createPicture` method generates the state machine diagram as a jpg-file. Now we will discuss the XMI-format.

XMI

XML Metadata Interchange (XMI) is a standard in which we can exchange metadata about models. It is a commonly used exchange format between several CASE tools. It is based upon eXtensible Markup Language (XML). If we dive into the XMI-code that Cpp2XMI produces, we can identify the state machine which is available in the source code. However, from this XMI-format we cannot determine what nested-choice pattern was used, and it is impossible to reconstruct the source code. In Listing 15 we see an example of the XMI which is produced by Cpp2XMI. It defines all the possible states of the state machine in the *<UML:SimpleState>* nodes. In the *<UML:Transition>* nodes the transitions between all states are defined.

A remark that we have to make is that the XMI specification does not implement position information of the elements of diagrams. This implies that we would need to manually position the states in a diagram. The transitions between states are then automatically placed by the CASE tool.
However the XMI-specification allows us to add extensions to XMI. These extensions can be CASE tool specific. Therefore we added the layout information of XMI which can be interpreted by Borland Together, of which we can see an example in Listing 16. Basically for every element that is part of a state machine diagram we store its position information in the geometry attribute. The position information is calculated by the layout algorithms of Cpp2XMI, as is described in the beginning of Section 4.4.

Listing 15 XMI-code, State Machine definition

```
<UML:Model name="StateMachineModel" xmi.id="myid">
    <UML:Namespace.ownedElement>
        <UML:StateMachine name="OBJ_control(example.i)" xmi.id="SM_OBJ_control"
            context="myid">
            <UML:StateMachine.top>
                <UML:CompositeState xmi.id="UMLCompositeState.0" name="TOP"
                    stateMachine="SM_OBJ_control">
                    <UML:CompositeState.subvertex>
                        <UML:SimpleState name="STATE_A" xmi.id="STATE_A"
                            container="UMLCompositeState.0" />
                        <UML:SimpleState name="STATE_B" xmi.id="STATE_B"
                            container="UMLCompositeState.0" />
                        <UML:SimpleState name="STATE_C" xmi.id="STATE_C"
                            container="UMLCompositeState.0" />
                    </UML:CompositeState.subvertex>
                </UML:CompositeState>
            </UML:StateMachine.top>
            <UML:StateMachine.transitions>
                <UML:Transition xmi.id="STATE_A->STATE_B"
                    stateMachine="SM_OBJ_control" source="STATE_A" target="STATE_B"
                    />
                <UML:Transition xmi.id="STATE_A->STATE_C"
                    stateMachine="SM_OBJ_control" source="STATE_A" target="STATE_C"
                    />
                <UML:Transition xmi.id="STATE_B->STATE_B"
                    stateMachine="SM_OBJ_control" source="STATE_B" target="STATE_B"
                    />
                <UML:Transition xmi.id="STATE_B->STATE_C"
                    stateMachine="SM_OBJ_control" source="STATE_B" target="STATE_C"
                    />
            </UML:StateMachine.transitions>
        </UML:StateMachine>
    </UML:Namespace.ownedElement>
</UML:Model>
```

Listing 16 XMI-code: State Machine layout information

```
1 <UML:Diagram xmlns:UML="//org.omg/UML/1.3" xmi.id="idOBJ_control"
      name="OBJ_control" diagramType="StateDiagram" style="">
2    <UML:Diagram.element>
3       <UML:DiagramElement xmi.id="OBJ_controlState0"
            geometry="Left=1347;Top=-2097;Right=409;Bottom=100;"
            subject="STATE_A">
4          <UML:PresentationElement.subject>
5             <Foundation.Core.ModelElement xmi.idref="STATE_A" />
6          </UML:PresentationElement.subject>
7       </UML:DiagramElement>
8       <UML:DiagramElement xmi.id="OBJ_controlState1"
            geometry="Left=201;Top=-1194;Right=409;Bottom=100;" subject="STATE_B">
9          <UML:PresentationElement.subject>
10            <Foundation.Core.ModelElement xmi.idref="STATE_B" />
11         </UML:PresentationElement.subject>
12      </UML:DiagramElement>
13      ...
14      <UML:DiagramElement xmi.id="OBJ_controlTransition0"
            geometry="Left=0;Top=0;Right=0;Bottom=0;" subject="STATE_A->STATE_B">
15         <UML:PresentationElement.subject>
16            <Foundation.Core.ModelElement xmi.idref="STATE_A->STATE_B" />
17         </UML:PresentationElement.subject>
18      </UML:DiagramElement>
19      ...
20   </UML:Diagram.element>
21 </UML:Diagram>
```

4.4.2. JPG writer

When we apply our tool to the source code, we eventually end up with two export-formats when reverse engineering state machine diagrams. One of them is XMI, which we discussed in Section 4.4.1. The other export format is a visual representation of the state machine diagram in jpg-format.
The XMI output can be used in several CASE tools, however it does not give us a picture of how the state machine diagram looks like. Therefore an jpg-export function was implement into Cpp2XMI. This function automatically exports the state machine diagrams into figures which can be viewed with an image-viewer. An example of a figure generated by Cpp2XMI can be seen in Figure 4.5. In this diagram-export, states are automatically positioned, which makes it an easy format to quickly view the results of the state machine extraction.

4.5. Improvements to Cpp2XMI

Although we have extended Cpp2XMI with the option to extract state machines from source code, we still find things that need improvement to make the Cpp2XMI tool work better.
One of the things that could be improved to Cpp2XMI is the use of a different CppML

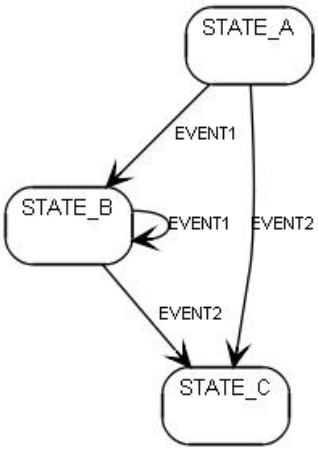

Figure 4.5.: Example: JPG Figure, State Machine figure with layout

parser for state machine extracting. The current CppML parser for state machine extracting is based on the SAX Parser. Our algorithms to detect state machine patterns traverse the AST multiple times. Due to the size and structure of the AST and due to the functioning of the SAX parser, this results in high memory consumption, and therefore this leads to performance issues. Multiple ways exist to overcome these issues. First of all we could use a different XML parser, which would traverse the AST more efficiently. Another option would be to modify our algorithms to traverse the AST less frequently. Third, transformation of the CppML into internal data structures, for instance by using the original CppML parser developed by Elena Korshunova, followed by the application of our algorithm to the internal data structure. We have chosen not to pursue this latter approach because not all source code is mapped to the internal data structure.
Another bug we have discovered during the case study is the inability of the SAX parser to cope with escape characters in the source code (hex 1B). This leads to a null-pointer exception raised in the SAX-parser class. The ideal solution would be to adjust the SAX parser so it can read strings with escape characters correctly. A workaround would be to transform all escape characters into other characters, or remove the escape characters completely from source code. We can safely apply this workaround solution for our state machine extraction method, as we do not expect escape characters to be used in the definitions of state machines.
Another improvement to Cpp2XMI is to correct the depth of the function call. As stated in Section 3.5.6 we currently only look for state transitions in functions called directly from within the switches. We do this to overcome the risk of potentially ending up in an infinite loop. We could implement some bookkeeping function to keep track of which functions were already checked. Hence reducing this risk to infinite loops to zero.

4.6. Summary

In this chapter we discussed the new architecture of Cpp2XMI. We focused on the conceptual designs of Cpp2XMI. We particularly went deeper into the changed and added modules of Cpp2XMI that implement the extraction of state machines from source code.

5. Case Study

In this chapter we describe the case studies that were used to test our reverse engineering tool. First we will briefly introduce the FSC-system that forms the basis for the two case studies.
For the first case study we applied our reverse engineering tool to the runtime system of FSC, a product that was provided by Vanderlande Industries. The runtime system consist of approximately 200K Source Lines of Code (SLoCs). From this system we try to extract state machines via pattern matching. Furthermore, we extract some metrics which indicate how many state machines could be implemented in the system. Subsequently, we will interpret these metrics.
For the second case study we applied our reverse engineering tool to the Gappex component of the runtime system of Version 5 of FSC. With this case study we show that our tool can generate up-to-date documentation that can replace the old out-of-date documentation.
We will end this chapter with conclusions that were drawn based upon this case study.

5.1. FSC

The FSC is a universal software product developed by Vanderlande Industries. Its main task is to control the automated material handling systems. The development of FSC started approximately 15 years ago. Over 100 man-years were put into the development of the current FSC versions. Though FSC was implemented in C, it is based on an object-oriented model. The software was initially written in C and some parts were rewritten to C++. More about the FSC software system can be found in Section 1.3.
Version 5 and 6 of FSC are currently used by customers all over the world. Therefore maintenance of the current versions is still a key task of the FSC-team. However, the documentation of these versions of FSC are out-of-date as several modifications in the design were made, e.g. by enhancements or refactorings. The modifications automatically have devaluated existing documentation. Since up-to-date documentation is missing, maintenance is hard because understanding the code is problematic.
As mentioned before, FSC contains several state machines. There are multiple abstraction levels at which we can identify these state machines, as was explained in Section 3.1. However, we will focus on the state machines of the components of material handling systems. These state machines are implemented in the runtime system of FSC, which is the core of the system. It communicates with all hardware. The runtime system handles all events, and stores and updates tracking information of packages. The runtime system is

the set of modules that can be used to control a material handling system. For instance, the Graphical User Interface does not belong to the runtime system. The size of this runtime system for version 6 of FSC is approximately 200K SLoC. The above mentioned reasons therefore make it an ideal candidate for our case study.

5.2. Application of Cpp2XMI to FSC V6

For a first case study, we have applied our reverse engineering tool to the runtime system of FSC Version 6. The source code for the runtime system consists of approximately 200K SLoC. The total size of the source code is approximately 11MB. The Abstract Syntax Tree (AST) of the source code of the runtime system in CppML format is approximately 170MB large.
At the moment 27 state machines have been extracted from the code of Version 6 of FSC. All these state machines are implemented by means of the 'switch-within-switch'-pattern, see Section 3.5. We have extracted some metrics from the source by applying the metrics-tool presented in Appendix B to the AST of this runtime system. These metrics can be found in Table 5.1. We discovered that this runtime systems has 36 'switch-within-switch'-constructs. From this we can conclude that ~75% of the 'switch-within-switch'-constructs are used for state machines. Approximately 11K SLoC are needed for implementing these 'switch-within-switch'-constructs. It is of course interesting to see why we missed these 9 'switch-within-switch'-occurrences.

Metric	#	Extra information
'Switch-switch' occurrences	36	11901 SLoCs
'Switch-if' occurrences	268	31423 SLoCs
'If-switch' occurrences	160	10075 SLoCs
Functions	4718	178142 SLoCs
Functions with 'control' in its name	95	13740 SLoCs
Enumerates	356	1748 elements
Enumerates with 'state' in its name	78	387 elements
Enumerates with 'event' in its name	34	275 elements
Case labels of switches originating from enumerate-type	3346	-
Case labels of switches not from enumerate-type	0	-

Table 5.1.: Metrics extracted for the runtime system of FSC V6

Further investigation has shown that only one of these 9 'switch-within-switch'-occurrences implements a state machine, i.e. the 'switch-within-switch'-occurrence in the *HostCioChannelManager.cpp* file. We missed this state machine implementation because this file is implemented in true C++ code. This state machine is therefore implemented in the object-oriented style. In the condition of the state-switch (outer-switch) there is a call to

a method of the object. This method returns the current state of the system. Instead of an assignment to the state-variable in the event-switch (inner-switch), there are calls to methods of the object, with the new state as a parameter of this method. Hence, this pattern implements a state machine, but it does not match with our state machine pattern. Therefore we cannot discover this state machine.
The other 8 detected 'switch-within-switch'-occurrences can be divided into two categories. Five of these detected 'switch-within-switch'-occurrences are in fact 'switch-within-switch-within-switch'-occurrences. Due to way our metrics are calculated these are counted as a 'switch-within-switch'-occurrence. In Listing 17 we see a small example of source code from this category. In that example we see a three levels deep nested switch. The outer-switch is still used for determining in which state the system currently is. The middle-switch is used for determining which event needs to be handled. And the third and most inner switch is used for determining some extra conditions. Hence, the third switch is used to make conditional transitions. Note that the assignment to the state variable is always performed in the most inner switch, but that this could be a second level or third level switch. This is another property of the 'switch-within-switch'-pattern.
However, our metrics-tool discovers the following switch-within-switch pairs:

- state-switch (line 4) / event-switch (line 7)
- state-switch (line 4) / condition-switch (line 12)
- event-switch (line 7) / condition-switch (line 12)

So our metrics-tool does let us believe that there would be three state machines implemented in the source code. However these three switch-switch-pairs in the example are implemented in one state machine. Our tool does extract the state machine for this example correctly. The number of switch-switch-pairs does not directly correspond to the number of implemented state machines. The number of state machines is less than or equal to the number of switch-switch-pairs.

The second category of missed 'switch-within-switch'-occurrences are those 'switch-within-switch'-occurrences that are used for additional debugging and logging information. These 3 'switch-within-switch'-occurrences have the same structure as the 'switch-within-switch'-nested-choice pattern. However, they lack an assignment to the state variable inside the inner-switches. Only debugging and logging information is produced in these inner-switches. Hence, these inner-switches do not implement state-machines. They are only used to check in which state the system is.
The size of the extracted state machines varied from 3 states up to 10 states. The number of transitions in the state machines varied from 3 up to 73 transitions. The average number of transitions in the extracted state machine diagrams is approximately 20. In the source code there are 356 enumerates implemented, of which 78 contains the word 'state', and 34 contains the word 'event' in their declaration. These latter two are likely to be used for defining the type of state-variables and defining the events that could occur. The complete source code of the runtime component of FSC contains ~7400 functions. These functions cover ~178K SLoC. However, only ~14K SLoC are needed for implementing approximately 100 functions with the string 'control' in the function-name. We

Listing 17 Switch-within-switch-within-switch occurrence

```
1  int CCodingScanner::Control( CModuleParcelData *pdata, TEvent event )
2  {
3     ...
4     switch(m_state)
5     {
6        case EState_NoParcel:
7           switch(event)
8           {
9              case EEvent_ParcelAtReadPos:
10                ...
11                result = ScheduleParcelLeft (pdata);
12                switch(result)
13                {
14                   ...
15                      case 1:
16                      m_state = EState_WaitForLengthKnown;
17                      break;
18                   case 0:
19                   case 2:
20                      m_state = EState_WaitForParcelLeft;
21                      break;
22                   default:
23                      break;
24                }
25                break;
26             ...
27             case EEvent_ParcelLeft:
28                m_state = EState_NoParcel;
29                ...
30          }
31          break;
32       ....
33    }
34 }
```

assume that not all of these ~100 functions implement the actual control-functions, but this number gives a reasonable estimation of the number of control-functions. One of the extracted state machines is shown in Figure 5.1, all transitions are decorated with conditional events. However, there are two transitions with *true*-labels, indicating that there are no condition for these transitions.
All the machines extracted were presented to the (software) engineers of the company and their correctness as well as importance were confirmed by them.

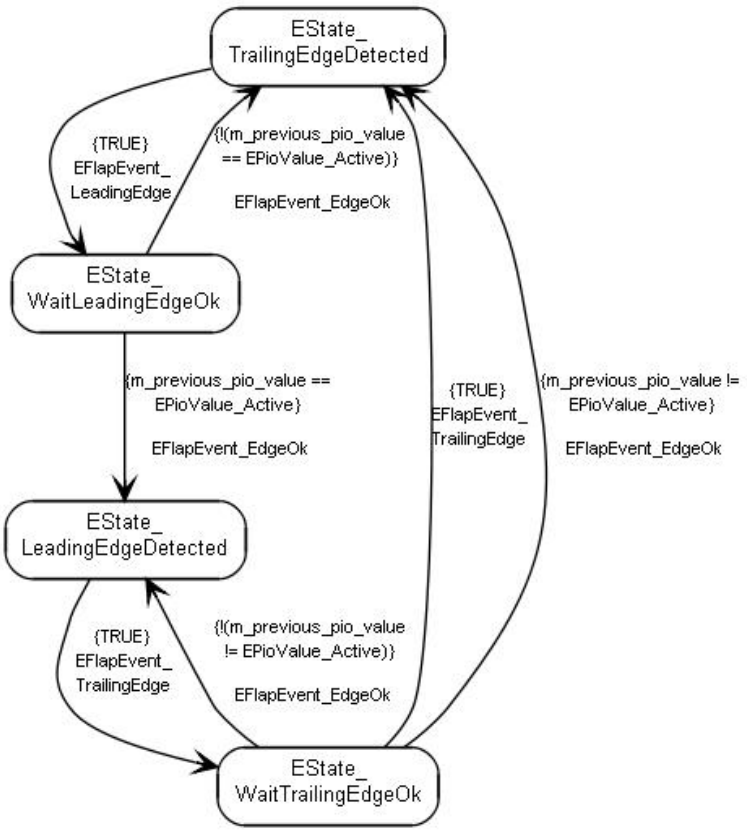

Figure 5.1.: Extracted state machine diagram for the FlapFilter component of FSC V6

From internal documentation [Ind], we believe that there should be at least 49 state machine diagrams in this runtime system of FSC Version 6. Hence, we believe that the other state machines are implemented via other state machine patterns. Therefore we extracted other metrics from this CppML file to expose whether these other state machines are indeed implement with other patterns. In the source code we identified 268 'if-within-switch'-constructs (covering ~30k SLoC) and 160 'switch-within-if'-constructs (covering ~10k SLoC). These numbers indicate the maximum number of events that are handled. The estimated number of state machines is expected to be much less.

5.3. Application of Cpp2XMI to Gappex V5

We will now focus on a certain component of the runtime system of FSC version 5. We shall take a closer look at the Gappex component. The Gappex is a piece of hardware that can control the gaps between bags/packages/boxes/etc. It is controlled by a separate module of the FSC software, which is being implemented by the *gappex_run.cpp* and *gappex_run.h* files (sources can be found in Appendix A). Figures 5.2 and 5.3 show the state machine diagrams of the Gappex component of the runtime system of FSC version 5. Figure 5.2 is the original state machine diagrams created by the FSC designers. This model is still being used in the current documentation. Figure 5.3 is the state machine diagram that was extracted by our reverse engineering tool. We can see that these diagrams more or less resemble each other. Together with the domain experts we tried to determine the origin of the differences.

In the new state machine diagram we see a new state, called `EState_InPowerSaving`. In 2007-2008 FSC was expanded with Power Savings options. This means that the automated material handling system puts itself into a standby-mode when no bags/packages/boxes/etc. have passed the system for a configured time. This way the system consumes less power. For this feature a new state was implemented, with corresponding transitions to other states.

Another big change in this Gappex component is the change in implementation of the automatic and manual mode of the Gappex. In the original design these states were split. However, in the actual implementation these states are merged into one state, called `EState_stopping`. This change automatically leads to changes in transitions to and from these automatic and manual states.

We can state that the original state machine diagram and the extracted state machine diagram are for $\sim 70\%$ the same. The differences are made by the previous observations. From these two state diagram we can conclude that the extracted diagram contains more specific information. Hence this actually represents the state machine diagram that is implemented in the code, while the original diagram specifies how the state machine originally should have been implemented. Furthermore, we can conclude that this component of the runtime system of FSC has changed compared to its original implementation. Thus, by using our reverse engineering tool we can generate up-to-date documentation, and identify flaws in the current documentation.

We have to remark that in the original state machine diagram the transitions are not labeled by their events. Hence it is impossible to make a complete comparison.

5.4. Conclusions

The case study showed that the recovered state machine diagrams are meaningful, but there are some differences due to a more abstract system representation of the diagrams made by the designer. While performing this case study, we found serious deficiencies of the reverse engineering tool. For example, there is a serious drawback to the CppML for-

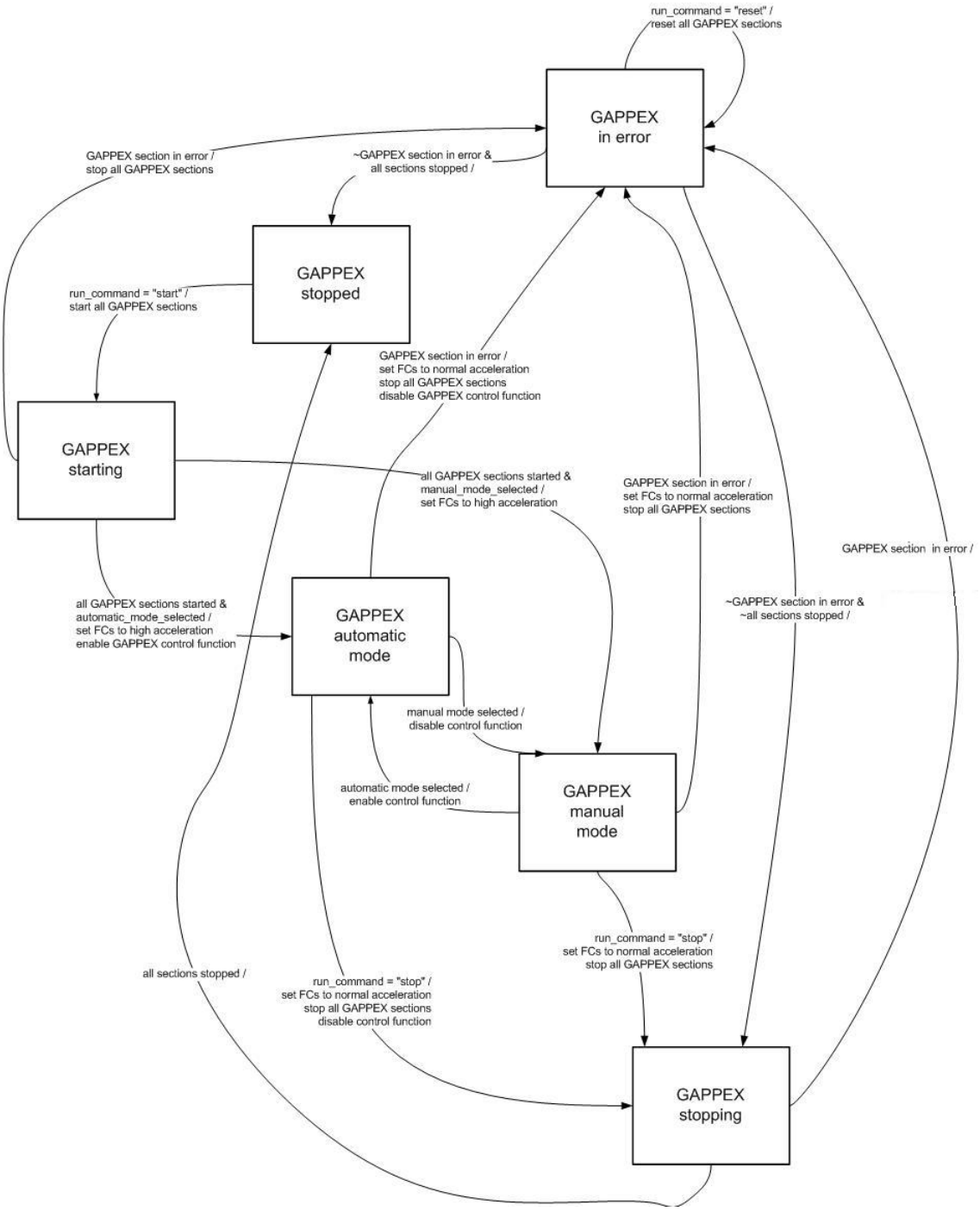

Figure 5.2.: Original design of the state machine diagram for the Gappex component of FSC V5

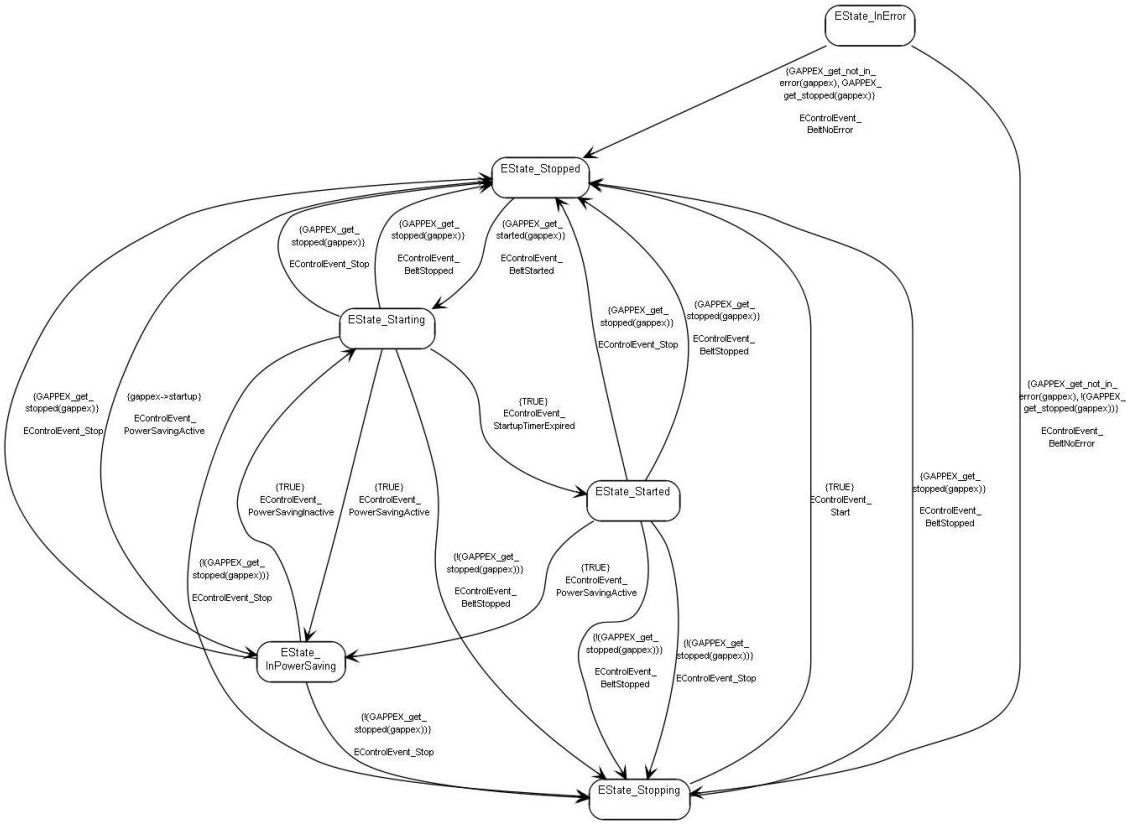

Figure 5.3.: Extracted state machine diagram for the Gappex component of FSC V5

mat. The size of the CppML-file explodes. For small amounts of source code, the CppML becomes rapidly large. Another example is that the SAX-parser, on which our CppML-parser is based, breaks when parsing escape characters (hex 0x1B). We implemented a workaround for this by replacing all escape characters by spaces. Furthermore we detected a drawback of our reverse engineering tool. Due to the size of the CppML-files, our tool uses a lot of resources (up to 1.8GB of RAM). However the normal 32-bit version of Java cannot reserve more than 1GB of RAM memory. Therefore to apply our tool to the complete FSC source it has to run on 64-bit versions of Java. Also the reverse engineering process consumes a lot of time when processing the complete runtime system in one go. We therefore recommend to apply the reverse engineering tool to smaller subsets of the source code of the runtime system.

5.5. Summary

In this chapter we discussed the case study. We applied our reverse engineering tool to Version 6 of FSC. With this case study we showed that our tool is capable of extracting state machine diagrams from source code. From the results of reverse engineering state machines from the source code of FSC version 6 we can conclude that our tool extracts all state machines that are implemented by the 'switch-within-switch'-pattern in the source code.

6. Conclusions

This chapter discusses the results of the project. First, we will briefly mention what the problem was and how it was solved. Subsequently, we will analyze the achieved goals. After that, suggestions for possible directions of future work are given.

6.1. Problem and solution

Like in most iterative software development processes evolution of the documentation does not keep up with the evolution of the software product itself. Design changes are forgotten to be reflected in the documentation. This makes the existing documentation less valuable. Especially during maintenance this can pose a problem. Maintainers of the software can experience difficulties when trying to understand how the software actually works. This problem also occurred with the FSC-software developed by Vanderlande Industries.
During this project we build a prototype tool that was capable of extracting this documentation for a software system. There are several types of documentation that can be reverse engineered for a software system. During this study we limited ourselves to the extraction of state machines from legacy source code. With this kind of documentation we tried to capture the dynamic behavior of the software system.
There are numerous way of extracting state machines for a software system. Our approach uses programming patterns that are generally used to implement state machines. We look for specific programming patterns in the source code, and automatically extract state machines from these patterns. By reverse engineering state machines from source code, we can extract up-to-date documentation that can be advantageous during maintenance.

6.2. Main Results

By means of this project we have shown that it is possible to extract state machine diagram from source code. We developed a prototype tool that allows extracting state machine diagrams from legacy C/C++ source code. The derived diagrams can be visualized by current CASE tools.
Our approach uses pattern finding to search for state machines implementation in the source code. There are several way of implementing state machines in source code. Together with domain experts we identified which pattern was mostly used by their software

system, and extended the Cpp2XMI tool to look for these patterns and extract state machines for this software system. We limited ourselves to the 'switch-within-switch'-pattern of the nested choice patterns family. For these patterns we can generate XMI-output and produce figures that correspond to the state machines implemented by the software system.
We extended Cpp2XMI with the option to extract state machines from source code. Cpp2XMI can now reverse engineer the following UML diagrams: class, sequence, activity and state machine diagrams. The latter only works for the state machines that were implemented by the 'switch-within-switch'-pattern. However, our theory could easily be applied to the other patterns as well.
We applied our reverse engineering tool to two case studies to validate our tool. The first case study was the runtime environment of Version 6 of the FSC software developed by Vanderlande Industries. We generated state machine diagrams for this system and hand-checked them to see if they matched the state machines that were implemented by the software. Metrics that were extracted from the source code of this runtime system indicate that there are 36 'switch-within-switch'-occurrences. But only 27 of them map to state machines.
Of these 9 missed 'switch-within-switch'-occurrences one implements a state machine, but it is implemented in C++. The other missed 'switch-within-switch'-occurrences are due to the fact that these switches are used for making conditional transitions, and due to the fact that these switches do not implement a state machine, but are merely used for producing debugging and logging information. From these extracted metrics we can further conclude that more state machines are implemented in this runtime system. However, these state machines are most likely implemented with other state machine patterns. Concluding, we extracted all state machines implemented by the 'switch-within-switch'-pattern.
The second case study was performed on the Gappex component of Version 5 of the runtime system of the FSC software. We compared the diagram made by the designers of the system with the extracted diagram. We saw that the system evolved, but the documentation did not. More details on these case studies can be found in Chapter 5.

6.3. Future work

Reverse engineering is an extensive topic that involves various directions. During the development of our tool we saw several opportunities to further expand our prototype tool.

The main thing that can be further investigated is the implementation of other patterns. In this prototype we only implemented the 'switch-within-switch' pattern of the nested choice patterns family. With relatively little work we think it may be possible to adapt the tool to also generate state machines that are implemented by other nested choice patterns, or even a mixture of these nested choice patterns.
Furthermore, we indicated other patterns that are commonly used to implement state machines in software systems. Future work could be to adapt the tool so it can also

extract state machines for these patterns.

Another interesting topic is the investigation of the possibilities to use the results of our state machine reverse engineering tool as input for model checkers. This requires expansion of the tool with an export format that can be used as input for one or more model checkers. Software developers can then check their (improved) design by properties they described in the language of the model checker. The properties are fed as input to the model checker. If certain properties do not hold, model checkers will display which paths you should execute to violate that property. Ideally, you would like to have the feedback of the model checkers exported back to the tool again, thus making visible which part of the source code is responsible for violating a certain property.

Further brainstorming on this topic results in thoughts about forward engineering models. Ideally it would be great to first reverse engineer the current software system, followed by adjusting these models to the new needs and finally regenerating new source code from these adjusted models via forward engineering.

More work can be carried out regarding the level of abstraction of the state machines. Our tool currently only extracts very local state machines. The reverse engineered state machines map to components of systems. However, during the project we saw opportunities to extract higher level state machines. This could be done by either combining state machines, i.e. by looking for states with common names, or transitions that have common names.
Another option to extract higher level state machines is by using logging information, and combining the logging information with the output of our tool. We have to remark that in some systems it is not always possible to generate extra logging, for instance in realtime systems (due to the fact that extra logging could lead to deadline misses).

Another feature that could be added to our tool is the option to extract OnEntry and OnExit values. The specification of UML state machines declares the option to store OnEntry and OnExit codes. We think that our tool could map the source code surrounding the actual assignment to the state variable to these OnEntry and OnExit values. This feature could lead higher storage of information inside the diagram. We have to remark that it is probably not wise to directly display this information in the diagram, since this would result in unreadable diagrams. However, storing this information in XMI-output would not pose a problem. Hence, this extra information could be useful in the earlier mentioned forward engineering option.

Further investigations can be performed in the fields of zooming and filtering. We can imagine that our tool could produce huge state machine diagrams. This problem can be overcome by either implementing some sort of filtering mechanism, or by implementing a 'zoom' function that makes it possible for state machines to consist of sub-state machines.

Finally, we would like to mention that future work may include the use of different parsers. Currently Cpp2XMI uses the Columbus/Can framework as a C/C++ parser. This has some serious drawbacks, which are addressed in [KPvdBM06]. Furthermore Columbus/-Can limits our tool to the C and C++ programming languages.

Not only the C/C++ parsers are not optimal. Also the internal CppML parser that looks for the patterns inside the CppML file is not optimal. We propose that changing this parser could lead to enormous performance gains.

6.4. Recommendations

Finally, we recommend to use one of the previous nested choice patterns to implement state machines in following version of the software system. Thus, it is always possible to reverse engineer the state machines from source code and extract up-to-date documentation form source code. Furthermore, we recommend to look at other state machine patterns when implementing state machines. The current implementation has deficiencies. In general, there is no separation of the state machine engine, the state transition information and the action code. This can limit a programmer in understanding the dynamic behavior modeled with the state machine. If the state-event space becomes large, this effect is strengthened. Readability and understandability are key aspects when modifying and enhancing the state machines.

A. Source code of Gappex Component

A.1. Gappex_run.h

```
1 ...
2 typedef void *GAPPEX_ID;
3 ...
4 /*============================== function prototypes ==============================*/
5
6 extern GAPPEX_ID GAPPEX_init( SHM_HANDLE shm_handle, GAPPEX_DATA *gappex_data );
7 extern int       GAPPEX_init_objects( GAPPEX_ID GAPPEX_id );
8 extern int       GAPPEX_exit_objects( GAPPEX_ID gappex_id );
9 extern int       GAPPEX_exit( GAPPEX_ID gappex_id );
10
11 extern int      GAPPEX_start( GAPPEX_ID gappex_id );
12 extern int      GAPPEX_stop( GAPPEX_ID gappex_id, unsigned ctrl_parameter );
13 extern int      GAPPEX_reset( GAPPEX_ID gappex_id );
14 extern int      GAPPEX_set_nom_speed( GAPPEX_ID gappex_id, unsigned
       perc_nom_speed );
15
16 extern int      GAPPEX_clear_tracking( GAPPEX_ID gappex_id );
17 extern int      GAPPEX_diag_init_all( GAPPEX_ID gappex_id,
18                                       FSC_DETAILED_OBJECT_CMD
                                             detailed_object_diag_cmd );
19 extern int      GAPPEX_diag_object( GAPPEX_ID                gappex_id,
20                                     const FSC_NAME           object_name,
21                                     FSC_DETAILED_OBJECT_CMD
                                           detailed_object_diag_cmd,
22                                     unsigned                 diag_parameter );
23
24 extern int      GAPPEX_set_belt_speed( GAPPEX_ID gappex_id,
25                                         unsigned beltno,
26                                         unsigned beltspeed );
27
28 extern int      GAPPEX_get_max_nr_parcels( SHM_HANDLE shm_handle, GAPPEX_DATA
       *data );
29 ...
```

A.2. Gappex_run.cpp

```
1 ...
2 enum TControlEvent
3 {
4     EControlEvent_Start,
5     EControlEvent_Stop,
6     EControlEvent_BeltNoError,
7     EControlEvent_BeltStarted,
8     EControlEvent_BeltStopped,
9     EControlEvent_StartupTimerExpired,
10    EControlEvent_PowerSavingActive,
```

```
11      EControlEvent_PowerSavingInactive
12  };
13
14  enum TState
15  {
16      EState_Stopped,
17      EState_Stopping,
18      EState_Starting,
19      EState_Started,
20      EState_InPowerSaving,
21      EState_InError
22  };
23
24  ...
25  struct GAPPEX_RTD
26  {
27      GAPPEX_DATA     *gappex_data;
28
29      int             nr_motors;
30      MTR_ID          motor_table[MAX_GAPPEX_MOTORS];
31
32      int             nr_ppis;
33      PPI_ID          ppi_table[MAX_GAPPEX_PPIS];
34
35      int               nr_sections;
36      SEC_ID            sec_owner_table[MAX_GAPPEX_SECTIONS];
37      TSectionInfo      sec_user_info_table[MAX_GAPPEX_SECTIONS];
38      TSectionContext   sec_context[MAX_GAPPEX_SECTIONS];
39
40      int             nr_sins;
41      SIN_ID          sin_table[MAX_GAPPEX_SINS];
42
43      int             nr_souts;
44      SOUT_ID         sout_table[MAX_GAPPEX_SOUTS];
45      SOUT_USER_ID    sout_user_id;
46
47      int             nr_updates;
48      CUpdate        *update_table[MAX_GAPPEX_UPDATES];
49
50      int             nr_scanners;
51      SCAN_ID         scanner_table[MAX_GAPPEX_SCANNERS];
52
53      int                   nr_pdexp_actions;
54      CParcelDataExport  *pdexp_table[MAX_GAPPEX_PDEXP_ACTIONS];
55
56      int             nr_pdchk_actions;
57      PDCHK_ID        pdchk_table[MAX_GAPPEX_PDCHK_ACTIONS];
58
59      int             nr_pstatechk_actions;
60      PSTATECHK_ID    pstatechk_table[MAX_GAPPEX_PSTATECHK_ACTIONS];
61
62      int             nr_hmps;
63      HMP_ID          hmp_table[MAX_GAPPEX_HMPS];
64
65      int             nr_bds;
66      BD_ID           bd_table[MAX_GAPPEX_BDS];
67
68      bool            startup;
69      bool            beltError;
70      FSC_NAME        main_sec_name;
71      SEC_USER_ID     main_sec_user_id;
72      int             nr_belts;
```

```
73      SEC_USER_ID     belts[MAX_GAPPEX_SECTIONS-1];
74      CUpdate        *sorted_updates[MAX_GAPPEX_UPDATES];
75
76      PIO_NAME        gappex_acceleration_selection_pio_name;
77      PIO             *gappex_acceleration_selection;
78      int             gappex_acceleration_value;
79
80      PIO_NAME        conveyor_mode_selection_pio_name;
81      PIO             *conveyor_mode_selection;
82      EVENT           conveyor_mode_selection_pio_event;
83      bool            conveyor_mode_selected;
84      int             conveyor_mode_value;
85
86      TIMER           startup_timer;
87      EVENT           startup_timer_event;
88
89      GAPCTRL_ID      gapctrl_id;
90
91      OVERALL_STATE_ID   overall_state_id;
92      OVERALL_STATE_VAL *overall_state_ptr;
93
94      int             perc_nom_speed;
95
96      TState  state;
97  };
98
99  /*====================  Structures, unions and enumerations =======*/
100
101 /*====================  Global function prototypes ================*/
102
103 /*====================  Static function prototypes ================*/
104
105 static int GAPPEX_StartBeltSections(GAPPEX_RTD *gappex);
106 static int GAPPEX_StopBeltSections(GAPPEX_RTD *gappex);
107 static int GAPPEX_StopAllSections(GAPPEX_RTD *gappex);
108 static bool GAPPEX_get_started(GAPPEX_RTD *gappex);
109 static bool GAPPEX_get_stopped(GAPPEX_RTD *gappex);
110 static bool GAPPEX_get_not_in_error(GAPPEX_RTD *gappex);
111 static void GAPPEX_set_gappex_acceleration(GAPPEX_RTD *gappex);
112 static void GAPPEX_reset_gappex_acceleration(GAPPEX_RTD *gappex);
113 static void GAPPEX_HandleMainSectionCommands(void *context, SEC_CMD cmd);
114 static void GAPPEX_HandleBeltSectionCommands(void *context, SEC_CMD cmd);
115 static int GAPPEX_attach_sout_state_handler(GAPPEX_RTD *gappex);
116 static int GAPPEX_detach_sout_state_handler(GAPPEX_RTD *gappex);
117 static void GAPPEX_sout_state_handler(void *context, bool     available);
118 static void GAPPEX_handle_conveyor_mode_selection_pio_event(void *context,
        EVENT event);
119 static int GAPPEX_control_start_stop(GAPPEX_RTD *gappex, TControlEvent event);
120 static void GAPPEX_init_structure(GAPPEX_RTD *gappex);
121 static int GAPPEX_init_gappex_acceleration_selection(GAPPEX_RTD *gappex);
122 static int GAPPEX_exit_gappex_acceleration_selection(GAPPEX_RTD *gappex);
123 static int GAPPEX_init_conveyor_mode_selection(GAPPEX_RTD *gappex);
124 static int GAPPEX_exit_conveyor_mode_selection(GAPPEX_RTD *gappex);
125 static int GAPPEX_attach_startup_timer(GAPPEX_RTD *gappex);
126 static int GAPPEX_detach_startup_timer(GAPPEX_RTD *gappex);
127 static int  GAPPEX_set_startup_timer(GAPPEX_RTD *gappex);
128 static void GAPPEX_reset_startup_timer(GAPPEX_RTD *gappex);
129 static void GAPPEX_handle_startup_timer_events(void *context, EVENT event);
130 static int GAPPEX_compare_sections(const void *op1, const void *op2);
131 static int GAPPEX_compare_updates(const void *op1, const void *op2);
132 static int GAPPEX_init_belts(GAPPEX_RTD *gappex);
133 static int GAPPEX_exit_belts(GAPPEX_RTD *gappex);
```

```
134 static int GAPPEX_init_sorted_updates(GAPPEX_RTD *gappex);
135 static int GAPPEX_exit_sorted_updates(GAPPEX_RTD *gappex);
136 static int GAPPEX_create_motor(GAPPEX_RTD *gappex, MTR_DATA *mtr_data);
137 static int GAPPEX_create_ppi(GAPPEX_RTD *gappex, PPI_DATA *ppi_data);
138 static int GAPPEX_create_section(GAPPEX_RTD *gappex, SEC_DATA *section_data);
139 static int GAPPEX_create_sin(GAPPEX_RTD *gappex, SIN_DATA *sin_data);
140 static int GAPPEX_create_sout(GAPPEX_RTD *gappex, SOUT_DATA *sout_data);
141 static int GAPPEX_create_update(GAPPEX_RTD *gappex, UPD_DATA *update_data);
142 static int GAPPEX_create_scanner(GAPPEX_RTD *gappex, SCAN_DATA *scanner_data);
143 static int GAPPEX_create_pdexp_action(GAPPEX_RTD *gappex, PDEXP_DATA
        *pdexp_data);
144 static int GAPPEX_create_pdchk_action(GAPPEX_RTD *gappex, PDCHK_DATA
        *pdchk_data);
145 static int GAPPEX_create_pstatechk_action(GAPPEX_RTD *gappex, PSTATECHK_DATA
        *pstatechk_data);
146 static int GAPPEX_create_hmp(GAPPEX_RTD *gappex, HMP_DATA *hmp_data);
147 static int GAPPEX_create_bd(GAPPEX_RTD *gappex, BD_DATA *bd_data);
148 static int GAPPEX_create_gapctrl(GAPPEX_RTD *gappex, GAPCTRL_DATA
        *gapctrl_data);
149 static GAPPEX_ID GAPPEX_create(SHM_HANDLE shm_handle, GAPPEX_DATA *gappex_data);
150 static int GAPPEX_remove(GAPPEX_ID gappex_id);
151
152 ...
153
154 /*****************************************************************************
155  *
156  * Function name: GAPPEX_start_stop_control
157  *
158  * Description:   Based on GAPPEX module STD in figure 3 of the ADD.
159  *                NOTE: State specific events are processed, others are ignored
160  *
161  * Parameters:    gappex  pointer to gappex runtime var
162  *                event   type of event to handle
163  *
164  * Returns:       0 on succes, -1 on error.
165  *
166  * On error:      When an error has occurred, errno contains a value
167  *                indicating the type of error that has been detected.
168  *                ENOSYS - unknown event.
169  *
170  * History:
171  *
172  * When:        Who:              What:
173  * 17-07-2000   J. Smeenk         Creation
174  * 26-04-2001   M. van Hoorn      Changed operation functionality/defines
175  * 08-05-2001   M. van Hoorn      Corrected state faults
176  * 16-05-2001   M. van Hoorn      Changed OVERALL_STATE use
177  * 21-05-2001   M. van Hoorn      Implemented the POWER_SAVING functionality
178  * 31-05-2001   M. van Hoorn      Speedup the function by removing unnecessary
       code
179  * 01-08-2001   M. van Hoorn      Reset timer also when a stop command is given
       during starting
180  * 29-01-2002   M. van Hoorn      Adapted start sequence
181  * 26-06-2003   M. van Hoorn      Only start the tracking section
182  *
183  *****************************************************************************/
184 static int GAPPEX_control_start_stop(GAPPEX_RTD *gappex, TControlEvent event)
185 {
186     int ret_val;
187
188     C_LogEnter("GAPPEX_control_start_stop");
189     C_LogParamPointer("gappex", gappex);
```

```
190     C_LogParamInt("event", event);
191     ret_val= 0;
192
193     switch (gappex->state)
194     {
195     case EState_Stopped:
196        switch (event)
197        {
198        case EControlEvent_Start:
199           gappex->startup = true;
200
201           /* Start main (tracking) section only */
202           if (SEC_start(gappex->main_sec_user_id) < 0)
203           {
204              fsc_perror("%s %d", __FILE__, __LINE__);
205              ret_val = -1;
206           }
207           break;
208
209        case EControlEvent_BeltStarted:
210           if (GAPPEX_get_started(gappex))
211           {
212              gappex->state = EState_Starting;
213              if (GAPPEX_set_startup_timer(gappex) < 0)
214              {
215                 fsc_perror("%s %d", __FILE__, __LINE__);
216                 ret_val = -1;
217              }
218           }
219           break;
220
221        case EControlEvent_PowerSavingActive:
222           if (gappex->startup)
223           {
224              gappex->state = EState_InPowerSaving;
225           }
226           break;
227
228        default:
229           break;
230        }
231        break;
232
233     case EState_Starting:
234        switch (event)
235        {
236        case EControlEvent_Stop:
237        case EControlEvent_BeltStopped:
238           if (GAPPEX_get_stopped(gappex))
239           {
240              gappex->state = EState_Stopped;
241           }
242           else
243           {
244              gappex->state = EState_Stopping;
245           }
246           GAPPEX_reset_startup_timer(gappex);
247           break;
248
249        case EControlEvent_StartupTimerExpired:
250           gappex->startup = false;
251           gappex->state   = EState_Started;
```

```
252             if (!gappex->conveyor_mode_selected)
253             {
254                GAPPEX_set_gappex_acceleration(gappex);
255                if (GAPCTRL_enable_controller(gappex->gapctrl_id) < 0)
256                {
257                   fsc_perror("%s_%d", __FILE__, __LINE__);
258                   ret_val = -1;
259                }
260             }
261             break;
262
263          case EControlEvent_PowerSavingActive:
264             gappex->state = EState_InPowerSaving;
265             GAPPEX_reset_startup_timer(gappex);
266             break;
267
268          default:
269             break;
270          }
271          break;
272
273       case EState_Stopping:
274          switch (event)
275          {
276          case EControlEvent_Start:
277             gappex->state   = EState_Stopped;
278             gappex->startup = true;
279
280             /* Start main (tracking) section only */
281             if (SEC_start(gappex->main_sec_user_id) < 0)
282             {
283                fsc_perror("%s_%d", __FILE__, __LINE__);
284                ret_val = -1;
285             }
286             break;
287
288          case EControlEvent_BeltStopped:
289             if (GAPPEX_get_stopped(gappex))
290             {
291                gappex->state = EState_Stopped;
292             }
293             break;
294
295          default:
296             break;
297          }
298          break;
299
300       case EState_Started:
301          switch (event)
302          {
303          case EControlEvent_Stop:
304          case EControlEvent_BeltStopped:
305             if (GAPPEX_get_stopped(gappex))
306             {
307                gappex->state = EState_Stopped;
308             }
309             else
310             {
311                gappex->state = EState_Stopping;
312             }
313
```

```
314             if (!gappex->conveyor_mode_selected)
315             {
316                 GAPPEX_reset_gappex_acceleration(gappex);
317                 if (GAPCTRL_disable_controller(gappex->gapctrl_id) < 0)
318                 {
319                     fsc_perror("%s_%d", __FILE__, __LINE__);
320                     ret_val = -1;
321                 }
322             }
323             break;
324 
325         case EControlEvent_PowerSavingActive:
326             gappex->state = EState_InPowerSaving;
327 
328             if (!gappex->conveyor_mode_selected)
329             {
330                 GAPPEX_reset_gappex_acceleration(gappex);
331                 if (GAPCTRL_disable_controller(gappex->gapctrl_id) < 0)
332                 {
333                     fsc_perror("%s_%d", __FILE__, __LINE__);
334                     ret_val = -1;
335                 }
336             }
337             break;
338 
339         default:
340             break;
341         }
342         break;
343 
344     case EState_InPowerSaving:
345         switch (event)
346         {
347         case EControlEvent_Stop:
348             if (GAPPEX_get_stopped(gappex))
349             {
350                 gappex->state = EState_Stopped;
351             }
352             else
353             {
354                 gappex->state = EState_Stopping;
355             }
356             break;
357 
358         case EControlEvent_PowerSavingInactive:
359             gappex->state = EState_Starting;
360 
361             if (GAPPEX_set_startup_timer(gappex) < 0)
362             {
363                 fsc_perror("%s_%d", __FILE__, __LINE__);
364                 ret_val = -1;
365             }
366             break;
367 
368         default:
369             break;
370         }
371         break;
372 
373     case EState_InError:
374         switch (event)
375         {
```

```
376         case EControlEvent_BeltNoError:
377            if (GAPPEX_get_not_in_error(gappex))
378            {
379               if (GAPPEX_get_stopped(gappex))
380               {
381                  gappex->state = EState_Stopped;
382               }
383               else
384               {
385                  gappex->state = EState_Stopping;
386               }
387
388               if (OVERALL_STATE_reset_error(gappex->overall_state_id,
                      gappex->overall_state_ptr) < 0)
389               {
390                  fsc_perror("%s_%d", __FILE__, __LINE__);
391                  ret_val = -1;
392               }
393            }
394            break;
395
396         default:
397            break;
398         }
399         break;
400
401      default:
402         break;
403      }
404
405      C_LogLeaveInt("GAPPEX_control_start_stop", (ret_val));
406      return(ret_val);
407   }
```

B. Verification tool

B.1. Files containing 'switch-within-switch'-occurrences extracted from the CppML-file

```
\fscrun\extlink\ESI_run.cpp(line: 416)
\fscrun\extlink\ESO_run.cpp(line: 369)
\fscrun\fws\COTR_run.cpp(line: 686)
\fscrun\hostman\HostCioChannelManager.cpp(line: 110)
\fscrun\infeed\LFA_run.cpp(line: 196)
\fscrun\infeed\LFA_run.cpp(line: 588)
\fscrun\infeed\FAINFEED_run.cpp(line: 1431)
\fscrun\merge\FMP_run.cpp(line: 514)
\fscrun\merge\MERGE_run.cpp(line: 408)
\fscrun\merge\MGEAREA_run.cpp(line: 515)
\fscrun\misc\PHOTO.cpp(line: 53)
\fscrun\misc\Balance.cpp(line: 1058)
\fscrun\misc\Balance.cpp(line: 4520)
\fscrun\misc\CodingKeyBoard.cpp(line: 1459)
\fscrun\misc\CodingKeyBoard.cpp(line: 1735)
\fscrun\misc\CodingKeyBoard.cpp(line: 1875)
\fscrun\misc\CodingKeyBoard.cpp(line: 2095)
\fscrun\misc\CodingScanner.cpp(line: 1085)
\fscrun\misc\CodingScanner.cpp(line: 1088)
\fscrun\misc\CodingScanner.cpp(line: 1213)
\fscrun\misc\HMP_run.cpp(line: 376)
\fscrun\misc\HSIND_run.cpp(line: 2575)
\fscrun\misc\ParcelSensor.cpp(line: 294)
\fscrun\misc\SSIND_run.cpp(line: 3064)
\fscrun\section\ENVCTRL_lib.cpp(line: 536)
\fscrun\sort\TRIPLEFUN_run.cpp(line: 2351)
\fscrun\sort\TRIPLESORT_run.cpp(line: 834)
\fscrun\sort\TRIPLESORT_run.cpp(line: 1338)
\fscrun\sort\TRIPLESWITCH_run.cpp(line: 1401)
\fscrun\sort\VERTIFUN_run.cpp(line: 2031)
\fscrun\sort\VERTISORT_run.cpp(line: 481)
\fscrun\sort\VERTISWITCH_run.cpp(line: 816)
\fscrun\spur\SPURFULL_run.cpp(line: 232)
\fscrun\spur\SPURINPUT_run.cpp(line: 470)
\fscrun\trans\GAPCTRL_run.cpp(line: 3747)
\fscrun\trans\GAPPEX_run.cpp(line: 1081)
```

B.2. ParserMetrics.java

```
1 package parser;
2 //import java.util.ArrayList;
3 import java.util.Iterator;
4 //import java.util.List;
5
```

```
6  //import org.jdom.Attribute;
7  import org.jdom.Element;
8  import org.jdom.Namespace;
9  import org.jdom.filter.ElementFilter;
10 import org.jdom.input.SAXBuilder;
11
12 import JDomFilter.AndFilter;
13 import JDomFilter.AttributeFilter;
14 import java.util.HashMap;
15
16 /**
17  * @author Dennie van Zeeland
18  *
19  *This class is responsible for parsing of the Columbus output files: CPPML and
        XMI
20  */
21
22 public class ParserMetrics {
23
24    static boolean showSelfLoops = false;
25    static private Element rootElement;
26
27    static private Namespace ns =
          Namespace.getNamespace("struc","columbus_cpp_schema/struc");
28    static private Namespace nsStatm =
          Namespace.getNamespace("statm","columbus_cpp_schema/statm");
29    static private Namespace nsType =
          Namespace.getNamespace("type","columbus_cpp_schema/type");
30    static private Namespace nsExpr =
          Namespace.getNamespace("expr","columbus_cpp_schema/expr");
31    static private Namespace nsUML = Namespace.getNamespace("UML",
          "//org.omg/UML/1.3");
32         static private SAXBuilder builder;
33
34
35          public static void main(String[] args) throws Exception{
36             builder = new SAXBuilder();
37             builder.setValidation(false);
38             builder.setIgnoringElementContentWhitespace(false);
39
40             org.jdom.Document document;
41             document = builder.build("D:\\FSC6\\filteredFSCRun.cppml");
42             rootElement = document.getRootElement();
43             enumMetrics();
44             enumSwitchMetrics();
45             switchIfMetrics();
46             ifSwitchMetrics();
47             functionMetrics();
48             switchMetrics2();
49          }
50
51
52     /*
53      * Determine Metrics about the switches
54     */
55     private static void switchMetrics() throws NumberFormatException {
56         int i = 0;
57         int j = 0;
58         int totalsize = 0;
59
60         // get all switches
```

```
61          Iterator<Element> it2 = rootElement.getDescendants(new
                ElementFilter("Switch", nsStatm));
62          while (it2.hasNext()) {
63              Element sSwitchElement = it2.next();
64              Iterator<Element> it3 = sSwitchElement.getDescendants(new
                    ElementFilter("Switch", nsStatm));
65              if (it3.hasNext()) { //only is this is a non inner-switch
66                  String path = sSwitchElement.getAttributeValue("path");
67                  String fileName = path.substring(path.lastIndexOf("\\") + 1,
                        path.lastIndexOf("."));
68                  i++;
69                  int start =
                        Integer.parseInt(sSwitchElement.getAttributeValue("line"));
70                  int end =
                        Integer.parseInt(sSwitchElement.getAttributeValue("endLine"));
71                  int size = end - start; //determine size of switch
72                  System.out.println("Size of state-Switch in " + fileName + ": "
                        + size + " lines");
73                  totalsize = totalsize + size;
74              }
75              while (it3.hasNext()) { //for all non-outer switches
76                  j++;
77                  Element sSwitchEl = it3.next();
78                  int start =
                        Integer.parseInt(sSwitchEl.getAttributeValue("line"));
79                  int end =
                        Integer.parseInt(sSwitchEl.getAttributeValue("endLine"));
80                  int size = end - start;
81                  String caseVal = "";
82                  System.out.println("Size of case " + caseVal + ": " + size + "
                        lines");
83              }
84          }
85          System.out.println("The number of 'Switch-Switch' occurences found: " +
                i);
86          System.out.println("The number of cases in the state-switch found (=the
                number of states): " + j);
87          System.out.println("Total Size: " + totalsize);
88      }
89
90      private static void enumMetrics() {
91          int nrEnum = 0;
92          int nrStateEnum = 0;
93          int nrEventEnum = 0;
94          int totalEnumSize = 0;
95          int totalStateEnumSize = 0;
96          int totalEventEnumSize = 0;
97          // get all enumerations
98          Iterator<Element> itEnum = rootElement.getDescendants(new
                ElementFilter("Enumeration", ns));
99          while (itEnum.hasNext()) {
100             Element sEnum = itEnum.next();
101             // get all elements of the enumeration
102             Iterator<Element> it3 = sEnum.getDescendants(new
                    ElementFilter("hasEnumerator", ns));
103             int enumSize = 0;
104             String name = sEnum.getAttributeValue("name");
105             while (it3.hasNext()) { // count the elements of the Enumeration
106                 enumSize++;
107                 it3.next();
108             }
109             totalEnumSize = totalEnumSize + enumSize;
```

```
110             //System.out.println("The size of Enum '" + name + "': " +
                    enumSize);
111             nrEnum++;
112             if (name.toLowerCase().indexOf("state") > -1) { //enums with
                    'state' in its name
113                 nrStateEnum++;
114                 totalStateEnumSize = totalStateEnumSize + enumSize;
115                 //System.out.println("STATE-ENUM: " + name);
116             }
117             if (name.toLowerCase().indexOf("event") > -1) { //enums with
                    'event' in its name
118                 nrEventEnum++;
119                 totalEventEnumSize = totalEventEnumSize + enumSize;
120                 //System.out.println("EVENT-ENUM: " + name);
121             }
122         }
123         System.out.println("The number of enum found: " + nrEnum);
124         System.out.println("The size of all enum found: " + totalEnumSize);
125         System.out.println("The number of 'state' enums found: " + nrStateEnum);
126         System.out.println("The size of all 'state' enums found: " +
                totalStateEnumSize);
127         System.out.println("The number of 'event' enums found: " + nrEventEnum);
128         System.out.println("The size of all 'event' enums found: " +
                totalEventEnumSize);
129     }
130
131
132
133     private static void enumSwitchMetrics() {
134         java.util.HashMap<String, String> enumeratorMap = new
                java.util.HashMap<String, String>();
135         java.util.TreeMap<String, String> enumMap = new
                java.util.TreeMap<String, String>();
136         int i = 0;
137         int j = 0;
138         int totalsize = 0;
139
140
141         // get all Enumerators and store them
142         Iterator<Element> myEnumerator = rootElement.getDescendants(new
                ElementFilter("Enumerator",ns));
143         while (myEnumerator.hasNext()){
144             Element enumerator = myEnumerator.next();
145             String key = enumerator.getAttributeValue("id");
146             String name = enumerator.getAttributeValue("name");
147             enumeratorMap.put(key, name);
148         }
149
150
151         // store all the elements of the enumerator with the enumeration
152         Iterator<Element> itEnum = rootElement.getDescendants(new
                ElementFilter("Enumeration", ns));
153         while (itEnum.hasNext()){
154             Element enumeration = itEnum.next();
155             Iterator<Element> enumerator = enumeration.getDescendants(new
                    ElementFilter("hasEnumerator", ns));
156             String enumerationName = enumeration.getAttributeValue("name");
157             while (enumerator.hasNext()){
158                 Element myEnum = enumerator.next();
159                 String ref = myEnum.getAttributeValue("ref");
160                 String enumeratorName = enumeratorMap.get(ref);
161                 enumMap.put(enumeratorName, enumerationName);
```

```
162             }
163         }
164
165         int caseValNotFound=0;
166         int caseValFound=0;
167         //get all the switches
168         Iterator<Element> it2 = rootElement.getDescendants(new
                ElementFilter("Switch", nsStatm));
169         while (it2.hasNext()) {
170             Element sSwitchElement = it2.next();
171             // get all the caselabels of the switches
172             Iterator<Element> itHasCaseLabel =
                    sSwitchElement.getDescendants(new ElementFilter("hasCaseLabel",
                    nsStatm));
173             while (itHasCaseLabel.hasNext()){
174                 Element hasCaseLabel = itHasCaseLabel.next();
175                 //get the identifier of the case label
176                 Iterator<Element> itId = hasCaseLabel.getDescendants(new
                        ElementFilter("Id", nsExpr));
177                 while (itId.hasNext()){
178                     // get the name of the identifier of the case label
179                     String idName = itId.next().getAttributeValue("name");
180                     if (enumMap.containsKey(idName)){ // check if the name is
                            from a enumerate-type
181 //                      System.out.println("Switch " +
    sSwitchElement.getAttributeValue("id") + ", " + idName + " was located in "
    + enumMap.get(idName));
182                         caseValFound++;
183                     }
184                     else{
185                         caseValNotFound++;
186                     }
187                 }
188             }
189
190             // get all inner switches
191             Iterator<Element> it3 = sSwitchElement.getDescendants(new
                    ElementFilter("Switch", nsStatm));
192             if (it3.hasNext()) { // this outer-switch has a inner-switch
193                 String path = sSwitchElement.getAttributeValue("path");
194                 String fileName = path.substring(path.lastIndexOf("\\") + 1,
                        path.lastIndexOf("."));
195                 i++;
196                 int start =
                        Integer.parseInt(sSwitchElement.getAttributeValue("line"));
197                 int end =
                        Integer.parseInt(sSwitchElement.getAttributeValue("endLine"));
198                 int size = end - start; //determine size of the outer-switch
199                 //System.out.println("Size of state-Switch in " + fileName + ":
                        " + size + " lines");
200                 totalsize = totalsize + size;
201             }
202             while (it3.hasNext()) { // for all non-outer switches
203                 j++;
204                 Element sSwitchEl = it3.next();
205                 int start =
                        Integer.parseInt(sSwitchEl.getAttributeValue("line"));
206                 int end =
                        Integer.parseInt(sSwitchEl.getAttributeValue("endLine"));
207                 int size = end - start; //determine the size of the non-outer
                        switches
208                 //System.out.println("Size of case: " + size + " lines");
```

```
209             }
210         }
211         System.out.println("En: Nr of Casevalues in enums defined: " +
                caseValFound);
212         System.out.println("En: Nr of Casevalues not in enums defined: " +
                caseValNotFound);
213         System.out.println("SS: The number of cases in the state-switch found
                (=the number of states): " + j);
214         System.out.println("SS: The number of 'Switch-Switch' occurences found:
                " + i);
215         System.out.println("SS: Total LOC of these Switch-Switches: " +
                totalsize);
216     }
217
218
219
220     private static void switchIfMetrics() throws NumberFormatException {
221         int i = 0;
222         int j = 0;
223         int totalsize = 0;
224
225         //get all switches
226         Iterator<Element> it2 = rootElement.getDescendants(new
                ElementFilter("Switch", nsStatm));
227         while (it2.hasNext()) {
228             Element sSwitchElement = it2.next();
229             //get all if-staments inside this switch
230             Iterator<Element> it3 = sSwitchElement.getDescendants(new
                    ElementFilter("If", nsStatm));
231             if (it3.hasNext()) { // this switch has a if-statement inside
232                 String path = sSwitchElement.getAttributeValue("path");
233                 String fileName = path.substring(path.lastIndexOf("\\") + 1,
                        path.lastIndexOf("."));
234                 i++;
235                 int start =
                        Integer.parseInt(sSwitchElement.getAttributeValue("line"));
236                 int end =
                        Integer.parseInt(sSwitchElement.getAttributeValue("endLine"));
237                 int size = end - start; //determine the size
238                 //System.out.println("SI: Size of state-Switch in " + fileName
                        + ": " + size + " lines");
239                 totalsize = totalsize + size;
240             }
241
242         }
243         System.out.println("SI: The number of 'Switch-If' occurences found: " +
                i);
244         System.out.println("SI: Total LOC of these Switch-If's: " + totalsize);
245     }
246
247
248     private static void ifSwitchMetrics() throws NumberFormatException {
249         int i = 0;
250         int j = 0;
251         int totalsize = 0;
252
253         // get all the if-statements
254         Iterator<Element> it2 = rootElement.getDescendants(new
                ElementFilter("If", nsStatm));
255         while (it2.hasNext()) {
256             Element sIf = it2.next();
257             // get all the switches inside this if
```

```
258             Iterator<Element> it3 = sIf.getDescendants(new
                    ElementFilter("Switch", nsStatm));
259             if (it3.hasNext()) { // this if has a switch inside
260                 String path = sIf.getAttributeValue("path");
261                 String fileName = path.substring(path.lastIndexOf("\\") + 1,
                        path.lastIndexOf("."));
262                 i++;
263                 int start = Integer.parseInt(sIf.getAttributeValue("line"));
264                 int end = Integer.parseInt(sIf.getAttributeValue("endLine"));
265                 int size = end - start; //determine size
266                 //System.out.println("IS: Size of state-Switch in " + fileName
                        + ": " + size + " lines");
267                 totalsize = totalsize + size;
268             }
269
270         }
271         System.out.println("IS: The number of 'If-Switch' occurences found: " +
                i);
272         System.out.println("IS: Total LOC of these If-Switches: " + totalsize);
273     }
274
275
276
277
278     private static void functionMetrics() {
279         int nrFunctions = 0;
280         int nrControlFunction = 0;
281         int totalFunctionSize = 0;
282         int totalControlFunctionSize = 0;
283         //get all Functions
284         Iterator<Element> itFunction = rootElement.getDescendants(new
                ElementFilter("Function", ns));
285         while (itFunction.hasNext()) {
286             Element sFunction = itFunction.next();
287             // ignore header files, or else we would count functions twice
288             if (!sFunction.getAttributeValue("path").endsWith(".h")){
289                 nrFunctions++;
290                 int start =
                        Integer.parseInt(sFunction.getAttributeValue("line"));
291                 int end =
                        Integer.parseInt(sFunction.getAttributeValue("endLine"));
292                 int size = end - start;
293                 //System.out.println("IS: Size of state-Switch in " + fileName
                        + ": " + size + " lines");
294                 totalFunctionSize = totalFunctionSize + size;
295                 // get all functions that have 'control' in its name
296                 if
                        (sFunction.getAttributeValue("name").toLowerCase().contains("control")){
297                     nrControlFunction++;
298                     totalControlFunctionSize = totalControlFunctionSize + size;
299                 }
300             }
301         }
302         System.out.println("FM: The number of Functions found: " + nrFunctions);
303         System.out.println("FM: The SLoC of all these Function: " +
                totalFunctionSize);
304         System.out.println("FM: The number of ControlFunctions found: " +
                nrControlFunction);
305         System.out.println("FM: The SLoC of all these ControlFunction: " +
                totalControlFunctionSize);
306     }
307
```

```
308
309
310
311
312
313
314
315
316
317         /*
318          * Determine Metrics about the switches
319         */
320         private static void switchMetrics2() throws NumberFormatException {
321             int i = 0;
322             int j = 0;
323
324             // get all switches
325             Iterator<Element> it2 = rootElement.getDescendants(new
                        ElementFilter("Switch", nsStatm));
326             while (it2.hasNext()) {
327                 Element sSwitchElement = it2.next();
328                 Iterator<Element> it3 = sSwitchElement.getDescendants(new
                            ElementFilter("Switch", nsStatm));
329                 if (it3.hasNext()) { //only is this is a non inner-switch
330                     String path = sSwitchElement.getAttributeValue("path");
331                     String line = sSwitchElement.getAttributeValue("line");
332 //                      String fileName = path.substring(path.lastIndexOf("\\") + 1,
        path.lastIndexOf("."));
333                     System.out.println(path + "(line: " + line + ")");
334                 }
335             }
336         }
337
338
339 }
```

Bibliography

[BJ06] F.J.A. Boerboom and A.A.M.G. Janssen. Fact extraction, querying and visualization of large c++ code bases design and implementation. Master's thesis, Technical University Eindhoven, 2006.

[CDH+00] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 439–448, New York, NY, USA, 2000. ACM.

[Dij68] E.W. Dijkstra. Go to statement considered harmful. *Comm. ACM*, 11(3):147–148, 1968. letter to the Editor.

[EGK+03] J. Ellson, E.R. Gansner, E. Koutsofios, S.C. North, and G. Woodhull. Graphviz and dynagraph static and dynamic graph drawing tools. In *Graph Drawing Software*, pages 127–148. Springer-Verlag, 2003.

[GZ05] Y.-G. Guéhéneuc and T. Ziadi. Automated reverse-engineering of uml v2.0 dynamic models. In *Proceedings of the 6th ECOOP Workshop on Object-Oriented Reengineering*, 2005.

[Ind] Vanderlande Industries. Internal document: States.doc.

[KPvdBM06] E. Korshunova, M. Petkovic, M.G.J. van den Brand, and M.R. Mousavi. Cpp2xmi: Reverse engineering of uml class, sequence, and activity diagrams from c++ source code. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 297–298, Washington, DC, USA, 2006. IEEE Computer Society.

[KTW98] R. Knor, G. Trausmuth, and J. Weidl. Reengineering c/c++ source code by transforming state machines. In *Proceedings of the Second International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families*, pages 97–105, London, UK, 1998. Springer-Verlag.

[McP] Scott McPeak. `http://www.cs.berkeley.edu/~smcpeak/elkhound/sources/elsa/`. Last visited: *17-03-2009*.

[MH02] A. Marburger and D. Herzberg. Reengineering of state machines in telecommunication systems. In *Proceedings of the 4. Workshop on Software Reengineering*, pages 21–23, 2002.

[Mos07] C. Mosler. Reengineering of state machines in telecommunication systems, 2007.

[MRR04] A. Milanova, A. Rountev, and B.G. Ryder. Precise call graphs for c programs with function pointers. *Automated Software Engineering*, 11:2004, 2004.

[RFMT02] Á. Beszédes R. Ferenc and T. Gyimóth M. Tarkiainen. Columbus - reverse engineering tool and schema for c++. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 172–181, Washington, DC, USA, 2002. IEEE Computer Society.

[Sim94] M.J. Simms. Using state machines as a design and coding tool, technical tutorial. *Hewlet-Packard Journal*, 45(6):27–32, 1994.

[SWM97] M.-A.D. Storey, K. Wong, and H.A. Muller. Rigi: A visualization environment for reverse engineering. In *In Proceedings of the 1997 international conference on Software engineering*, pages 606–607, 1997.

[TV08] A. Telea and L. Voinea. Solidfx: An integrated reverse engineering environment for c++. In *CSMR: 12th European Conference on Software Maintenance and Reengineering*, pages 320–322. IEEE, 2008.

[vdBHdJ+01] M. van den Brand, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The asf+sdf meta-environment: a component-based language development environment. In *Proceedings of Compiler Construction 2001 (CC 2001)*, LNCS. Springer, 2001.

[WBAH08] N. Walkinshaw, K. Bogdanov, S. Ali, and M. Holcombe. Automated discovery of state transitions and their functions in source code. *Software Testing, Verification and Reliability*, 18(2):99–121, 2008.

[WBHS07] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Reverse engineering state machines by interactive grammar inference. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 209–218, Washington, DC, USA, 2007. IEEE Computer Society.

[YXM06] H. Yuan, T. Xie, and E. Martin. Automatic extraction of abstract-object-state machines from unit-test executions. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 835–838, New York, NY, USA, 2006. ACM.

[ZHJ04] T. Ziadi, L. Helouet, and J.-M. Jezequel. Revisiting statechart synthesis with an algebraic approach. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 242–251, Washington, DC, USA, 2004. IEEE Computer Society.