Determining the feasibility of automatically translating SMILE to a Java framework

by

Said Aspen

LIU-IDA/LITH-EX-A--08/054--SE

2008-11-27
Final Thesis

Determining the feasibility of automatically translating SMILE to a Java framework

by

Said Aspen

LIU-IDA/LITH-EX-A--08/054--SE

2008-11-27

Supervisor: Niklas Lanzén, Ericsson AB
Examiner: Christoph Kessler, Linköping University
Determining the feasibility of automatically translating SMILE to a Java framework

Said Aspen

MTsim (Mobile Traffic Simulator) is an Ericsson AB internal software application that is part of 2Gsim. It is used to simulate elements of a GSM (Global System for Mobile communications) network for feature testing and automated testing. It is written in the programming language TSS Language, also known as SMILE which is a proprietary Ericsson programming language. SMILE is based on the principles of state matrix programming which in essence means that each program is on its own a finite state machine. The language is old and was originally intended as a macro language for smaller test programs, not for applications the size of MTsim.

It is of interest to evaluate the feasibility of performing an automatic conversion of applications written in SMILE, with special interest in converting MTsim, to a Java framework since Java has many advantages compared to SMILE. Java, as a language, is well suited for larger applications, there are numerous well supported tools and there is a much wider spread competence than there is for SMILE.

It is clear that in order to do a full conversion of a SMILE program to a Java framework two applications must be implemented. First a Java framework, which acts as a run time environment, must be designed which can host the translated programs. The other part is an actual translator which takes a SMILE program as input and outputs a translated Java program. A more sophisticated framework is preferred since it makes the actual translated programs more light weight and easy to read which means higher degree of maintainability.

There are different ways to implement state machines in Java but the most flexible and versatile is to implement it as a black-box framework in an object oriented way where the framework has sophisticated mechanisms for message and event handling which is central to any state machine framework.

The translation for SMILE can easily be done by using a AST (abstract syntax tree) representation, which is a full representation of the SMILE program in tree-form. The AST is obtained from an intermediate state of the SMILE program compiler.

State-machines, software translation, Java, software conversion, MTsim, TSS Language, state matrix programming, Java framework, framework design, object oriented implementation of state machines, abstract syntax tree
Abstract

MTsim (Mobile Traffic Simulator) is an Ericsson AB internal software application that is part of 2Gsim. It is used to simulate elements of a GSM (Global System for Mobile communications) network for feature testing and automated testing. It is written in the programming language TSS Language also known as SMILE which is a proprietary Ericsson programming language. SMILE is based on the principles of state matrix programming which in essence means that each program is on its own a finite state machine. The language is old and was originally intended as a macro language for smaller test programs, not for applications the size of MTsim.

It is of interest to evaluate the feasibility of performing an automatic conversion of applications written in SMILE, with special interest in converting MTsim, to a Java framework since Java has many advantages compared to SMILE. Java, as a language, is well suited for larger applications, there are numerous well supported tools and there is a much wider spread competence than there is for SMILE.

It is clear that in order to do a full conversion of a SMILE program to a Java framework two applications must be implemented. First a Java framework, which acts as a run time environment, must be designed which can host the translated programs. The other part is an actual translator which takes a SMILE program as input and outputs a translated Java program. A more sophisticated framework is preferred since it makes the actual translated programs more light weight and easy to read which means higher degree of maintainability.

There are different ways to implement state machines in Java but the most flexible and versatile is to implement it as a black-box framework in an object oriented way where the framework has sophisticated mechanisms for message and event handling which is central to any state machine framework.

The translation for SMILE can easily be done by using a AST (abstract syntax tree) representation, which is a full representation of the SMILE program in tree-form. The AST is obtained from an intermediate state of the SMILE program compiler.

The concept of automatically translating SMILE programs to Java is very much feasible. There are however some problems, data types of the two languages does not match, comments from the original code will be hard to transfer to the target code, scalability to a multithreaded environment will be difficult, there is a risk of introducing new logical bugs and the biggest problem of all is the risk of reducing the quality and readability of the code which means that future maintainability and extendibility will be limited. The actual cost of a total translation is high and it is yet to be determined if such a translation, even if feasible, would prove worthwhile.
Acknowledgement
First and foremost I thank my supervisor Niklas Lanzén, Fredrik Ljung, Jan Eriksson and Anders Holmstrand at Ericsson AB for their support, availability and technical knowledge and experience. I am also grateful for all the valuable technical discussions I had with others at Ericsson, everyone showed me such hospitality and was always available and eager to help.

My thanks also go out to Jonas Ljunggren and Henrik Karlsson for help with reviewing my work.

I also thank Prof. Dr. Christoph Kessler, Linköping University, examiner for this thesis project
# Table of contents

1. **INTRODUCTION** ........................................................................................................................................... 1
   1.1. Typographic conventions ............................................................................................................................. 1
   1.2. Background of thesis ....................................................................................................................................... 1
   1.3. Purpose ......................................................................................................................................................... 2
   1.4. Problem statement .......................................................................................................................................... 2
       1.4.1. Requirements .......................................................................................................................................... 3
   1.5. Methodology ................................................................................................................................................... 3
   1.6. Limitation of thesis ......................................................................................................................................... 4
   1.7. Structure of thesis ......................................................................................................................................... 4

2. **BACKGROUND** ............................................................................................................................................. 6
   2.1. The finite state machine ................................................................................................................................. 6
       2.1.1. Ways of implementing the finite state machine ..................................................................................... 7
       2.1.2. Procedural ................................................................................................................................................. 8
       2.1.3. Object oriented ........................................................................................................................................ 9
       2.1.4. Issues when implementing FSMs ........................................................................................................ 10

3. **SMILE** .......................................................................................................................................................... 13
   3.1. Structure of a TSS test application ................................................................................................................. 13
   3.2. Test programs .................................................................................................................................................. 14
       3.2.1. TSS Test programs as FSMs .................................................................................................................. 15
   3.3. Basic language elements ............................................................................................................................... 16
       3.3.1. Literals ..................................................................................................................................................... 16
       3.3.2. Identifiers ............................................................................................................................................... 17
       3.3.3. Data types .............................................................................................................................................. 17
       3.3.4. Operators ............................................................................................................................................... 18
       3.3.5. Conversions ........................................................................................................................................... 19
       3.3.6. Variables ............................................................................................................................................... 19
       3.3.7. Control flow ............................................................................................................................................ 20
       3.3.8. Messages .............................................................................................................................................. 20
       3.3.9. Timers .................................................................................................................................................... 21
       3.3.10. Procedures and functions .................................................................................................................... 21
       3.3.11. Events .................................................................................................................................................. 22
   3.4. Test Program Compiler ................................................................................................................................. 23
       3.4.1. Dataflow ............................................................................................................................................... 24
       3.4.2. Abstract syntax tree ............................................................................................................................. 24
   3.5. TSS Interpreter ................................................................................................................................................ 26

4. **BASIC LANGUAGE TRANSLATION OF SMILE TO JAVA** ............................................................................ 27
   4.1. Language comparison ................................................................................................................................. 27
   4.2. Translating literals ......................................................................................................................................... 28
       4.2.1. Occurrence of literals in MTsim ............................................................................................................ 28
       4.2.2. Integer literals ........................................................................................................................................ 28
       4.2.3. String literals ......................................................................................................................................... 29
       4.2.4. BCD literals ......................................................................................................................................... 30
5. DESIGN OF A COMMON JAVA FRAMEWORK ............................................. 43
5.1. Hollywood principle .............................................................................. 45
5.2. Responsibilities of the framework and run-time environment .............................................. 45
5.3. General decision about the framework ....................................................... 46
5.4. Communicating with external software ...................................................... 47
5.5. Event kernel ................................................................................................. 47
5.6. Message decoding ....................................................................................... 48
5.6.1. Messages .................................................................................................. 48
5.6.2. Message matching ..................................................................................... 49
5.7. Multiple threads ........................................................................................... 50
5.8. Existing frameworks ..................................................................................... 52
5.8.1. UniMod ...................................................................................................... 53
5.8.2. jfsm ........................................................................................................... 53
5.9. Overview of the Java framework runtime environment ........................................... 54
5.10. Implement a prototype of the Java framework ............................................ 55
5.10.1. Test case ................................................................................................. 55
5.10.2. Test programs ......................................................................................... 56
5.10.3. Test program example ......................................................................... 59
5.10.4. Modeling variable scope ....................................................................... 61
5.11. Error handling ......................................................................................... 62
5.11.1. Run-time errors ..................................................................................... 62
5.11.2. Run-time environment errors ................................................................. 62
5.11.3. System errors ........................................................................................ 63

6. TRANSLATING SMILE TO A JAVA FRAMEWORK .................................. 64
6.1. Translating MTsim to the Java framework .................................................. 64
6.2. Different approaches .................................................................................... 64
6.2.1. Direct source code translation ................................................................ 64
6.2.2. Translation via abstraction and reimplementation ................................... 64
6.2.3. Translation via intermediate language ..................................................... 65
6.2.4. Chosen approach for translation ............................................................. 65
List of figures

Figure 1 UML State chart for simplified TCP network connection ................................. 6
Figure 2 TCP network connection implemented with State pattern .............................. 10
Figure 3. Structure of TSS test application .................................................................. 13
Figure 4 Structure of a Test program ............................................................................ 15
Figure 5 Incoming message events [19] ......................................................................... 21
Figure 6 Function block for the Test Program Compiler (TCP) [3] .............................. 23
Figure 7 Data flow of the TPC [3] ................................................................................. 24
Figure 8 Simple AST for assign statement ................................................................... 25
Figure 9 Two representations of a TPC AST for assign statement .............................. 25
Figure 10 Relationship of complexity for the translator and the target framework ........ 44
Figure 11. Mapping test program instances to Java threads ......................................... 52
Figure 12. Java framework run time environment function blocks ............................... 54
Figure 13. Flow of message passing in the Java framework run time environment ......... 55
Figure 14. Simple model of the implementation of FSM .............................................. 57
Figure 15 Structure and placement of variables ........................................................... 62
Figure 16. Dataflow of translation of SMILE test program to Java test program ........... 65
Figure 17 Structure of the State pattern as described by GoF ...................................... 82
Figure 18 Basic structure of the visitor design pattern .................................................. 83
Figure 19 Structure of the Command pattern as described by GoF .............................. 84
Figure 20 Basic structure of the Singleton design pattern ............................................. 84
Figure 21 Structure of the Flyweight pattern as described by GoF ............................... 86
List of tables
Table 1. Transition table for simplified TCP network connection ........................................ 7
Table 2 Common changes made to FSMs .............................................................................. 11
Table 3 State matrix for simplified TCP connection ............................................................ 14
Table 4 Operator precedence in SMILE ............................................................................ 18
Table 5 Restrictions on data conversion in SMILE ............................................................ 19
Table 6 Multiple variables with the same identifier ............................................................. 20
Table 7 Summary of SMILE and Java language comparison .............................................. 27
Table 8 Occurrences of some types of SMILE literals in MTsim ....................................... 28
Table 9 Integer literal comparison in SMILE and Java ...................................................... 29
Table 10 Summary of literal translation ............................................................................. 31
Table 11 Reserved words that are not allowed as identifiers in Java ................................ 32
Table 12 Unsigned operator problems in Java .................................................................... 36
Table 13 Operator precedence in Java ................................................................................ 37
Table 14 Converting mixes of numerical and boolean values .......................................... 37
Table 15 Structure of INTERNAL_REQ message primitive .......................................... 49
Table 16 Packed representation of INTERNAL_REQ message primitive ...................... 50
Table 17 Number of registered SourceForge projects for some popular programming languages .................................................................................................................. 88
Table 18 Top 10 rated programming languages from TIOBE ........................................... 88
Introduction

1. Introduction
This chapter gives an introduction to the final thesis project. It presents the background, purpose and methodology for the thesis.

1.1. Typographic conventions
Program code will be written in pseudo code inside a shaded box. The segment below demonstrates how code segments are presented.

```java
// This is a comment
class exampleClass{
    int variableName = 0;
    void methodName(int firstArgument, char secondArgument){
        return equals(variableName, "This is a string");
    }
}
```

<table>
<thead>
<tr>
<th>Style</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bold</td>
<td>Bold text indicates language control flow structures, types or reserved words in the language.</td>
</tr>
<tr>
<td>Italic</td>
<td>Text in italics represents methods, functions or procedures. Italics are also used for comments.</td>
</tr>
</tbody>
</table>

1.2. Background of thesis
2Gsim is an Ericsson AB product which is used for simulations of second generation wireless telephone technology networks (2G networks). It is a tool used to test, verify and support the development of all of the 2G products that Ericsson AB maintains. The main purpose of 2Gsim is to simulate realistic traffic cases in order to verify functionality of a BSC (Base Station Controller) under load.

The application has a wide range of uses including validation tests, protocol conformance tests, traffic tests, function tests and system tests. The SUT (System Under Test) is usually a BSC.

MTsim (Mobile Traffic Simulator) is an Ericsson AB internal software application that is part of 2Gsim. MTsim is used to simulate elements of a GSM (Global System for Mobile communications) network and is mainly used for feature testing and automated testing. It is written in the programming language TSS Language, also known as SMILE which is a proprietary Ericsson programming language. Since the language was created internally at Ericsson all competence, tools and experience is isolated to Ericsson AB. The programming language in itself is old and was originally intended as a macro language for
smaller test programs, not for applications the size of MTsim. If MTsim would have had been written in Java, or some other more common language, there would be a lot more competence, better tools and a broader base of experience available.

1.3. Purpose
The purpose of this thesis is to evaluate the feasibility of performing an automatic conversion of applications written in SMILE, with special interest in converting MTsim, to a Java framework. Java has many advantages compared to SMILE. Java, as a language, is well suited for larger applications, there are numerous well supported tools and there is much wider spread competence than there is for SMILE. This is the motivation behind the thesis investigation.

Even if the thesis investigation finds that a fully automated conversion of MTsim to Java is possible, this in itself does not necessarily mean that the thesis is successful. Since the main aim is to investigate the feasibility, some measure of cost and correctness must be evaluated. The applications that are generated from translation should also be maintainable and extendible in order for a translation to be useful. This is important since the product, or application, is still maintained and developed, thus it is important that the end result of the translation can be used when going forward with the application.

1.4. Problem statement
This section introduces the different problems that the thesis is to investigate.

- Is it possible to automatically convert the application MTsim from SMILE into Java?
  - How much user interaction would be needed for such a conversion?
  - How much time would such a translation require?
  - What are the risks involved?

**Comment:** If the thesis shows that such a translation is not possible, or that it is possible but very expensive such a result is equally valuable as a result showing that it is possible and how it can be done. It should also be noted that this question depend on the next problem statement.
Introduction

- Can any arbitrary program written in SMILE be converted to a Java framework?
  - What parts would such a framework consist of?
  - How large and complex would such a framework be?
  - Will the code of the new Java framework be maintainable?

Comment: There is one very important aspect that needs to be taken into consideration. It is not possible to do a direct syntactical translation of the language since they are fundamentally different. Programs written in SMILE are written as multiple state machines which must be modeled into object oriented code in Java. This will be explored further later in the report.

1.4.1. Requirements
There are some requirements that are essential in order for the result to be interesting. These are requirements that need to be fulfilled in order for any solutions to the stated problems to be useful.

1. The execution speed performance of the suggested state machine framework shall not be worse than that of the current solution (the MTsim run-time environment).
2. Translating from SMILE to the Java framework should not require extensive human interaction.
3. The translation from SMILE to Java should not introduce new logical bugs.

1.5. Methodology
This thesis project was carried out during a placement at Ericsson AB in Linköping, Sweden, during the spring and summer of 2008. The thesis is a final thesis project at the Department of Computer and Information Science, Linköping University. Supervising the work at Ericsson was Niklas Lanzén, at the time Requirement Manager for TSS and 2Gsim Management. Prof. Dr. Christoph Kessler was the examiner at Linköping University.

The master thesis project is composed of three main parts:

- Research about different topics which needs to be addressed in the project; Finite state machines and their implementation in software, the SMILE specification and the conformance of applications written in SMILE to this specification, MTsim, design principles and approaches concerning creation of common software frameworks and different types software translation. This part of the project also aims to research related work. The research part of the project is mainly performed
as a study of literature where books, articles, conference papers and other thesis projects are used as source material.

- The second part of the thesis project is to implement a prototype which can handle simplified conceptual translation from SMILE to Java. This part of the project is conducted as a design and implementation project which results in software that contains all the functionality for a conceptual translation. It is important to note that since the prototype is of conceptual nature, it is not meant as base for further development.

- The last part of the thesis project is to analyze the results of the prototype and then discuss the possibility of a complete automated translation of SMILE with restricted resources. It also aims to answer the questions formulated in the purpose, problem statements and the requirements and then to give general recommendations or suggestion for future work.

1.6. Limitation of thesis

The thesis will be restricted in a number of ways. The thesis will only discuss translation to Java and only from SMILE as specified by the original thesis proposal. There will be a short discussion about other languages that could be of interest for translation.

Since SMILE is an internal language at Ericsson AB the reader might draw the conclusion that the number of possible interested readers is low and restricted to Ericsson AB. This is not necessarily the case since anyone interested in translating a programming language based on state machines could find this report useful. There are many general problems associated with such a translation that is discussed in this project.

During the thesis project second phase, the implementation of the translation software, the software itself is restricted. Only core functionality is implemented. Once again, it is important to remember that the implementation part of the project is to create a prototype which aims to illustrate a possible solution on a conceptual level, not to solve the problems themselves.

1.7. Structure of thesis

The thesis report is structured into ten main sections. The first section, Introduction, focus on introducing the reader to the thesis project, the background, the methodology and the limitation. Section 2 gives the user some background information about terminology and concepts that will be discussed or referred to later in the thesis. It discusses the concept of finite state machines and problems as they are implemented in programming. Most of this section can used to read up on topics that the reader is not familiar with.

Section 3 introduces the reader to the SMILE language. It gives a shorter description of the main characteristics of this programming language and how it relates to the previously
presented concept of the finite state machine. Language specific constructs are presented in preparation for the translation which will follow. Section 4 (Basic language translation of SMILE to Java) starts the presentation of the ideas concerning translating the language. This section presents how the basic SMILE language can be translated to Java in general. Section 5 (Design of a common Java Framework) describes a possible framework which will be used as function library and as a run time environment for the translated applications. This section is needed for the complete translation of SMILE to Java and leads up to Section 6 (Translating SMILE to a Java framework. In this section the translation application and the translation process is presented and discussed. The results are then analyzed and discussed in Section 7 (Discussion). It also includes discussions of the relevance of the results, other possible solutions and an evaluation of the project as a whole. Section 8 (Review) ends the report with conclusions and recommendations.

Appendix A holds information about the design patterns which are discussed in the thesis. Appendix B gives a short view of the Java programming language and its history. The last appendix, Appendix C contains a glossary with acronyms and technical words used in the thesis report.
2. Background

This section presents background for the thesis project. For further background material see the appendices at the very end of the report. The appendices present information about design patterns, software quality and the Java programming language and virtual machine.

2.1. The finite state machine

The Finite State Machine (FSM) is a model that can be applied to a wide variety of fields ranging from business administration and psychology to communications, software modeling and linguistics [5].

The model originates from system theory in the middle of the last century and emerged as a tool for analysis. It is defined by Gill as “a synchronous system with a finite input alphabet, a finite output alphabet and a finite number of states where each output symbol is a function of the input symbol and the state at that a given time” [5]. This is not the definition which will be adopted in this report. Gill’s definition is modified such that the output alphabet introduced is replaced by an action. We also extend the definition by introducing the concept of transitions. Thus our definition becomes:

DEFINITION 1. A Finite-State Machine is a synchronous system with a finite input alphabet called events, a finite number of actions and a finite number of states. An action is the direct consequence of a specific event and state. The process of changing the state of the system is called a transition. Each Finite State Machine has an initial state.

There are different ways of visualizing state machines, some more frequently used than others. Two different representations will be used in this report. The first way, and also the currently most common way, is by using UML diagrams [15].

To illustrate the concept of a FSM (and other concepts later in the thesis) an example of a TCP network connection will be used. The example is taken from the book Design Patterns: Elements of Reusable Object-Oriented Software by Gamma et al. [4]

A TCP connection can be in different states and its behavior depends on the state of the connection. A possible UML representation of the state machine model is shown in Figure 1.

![Figure 1 UML State chart for simplified TCP network connection](image)
In this example there are three states: Established, Listening and Closed. The arrows between the states in the diagram are the transitions which are triggered with events. The events are represented in the diagram as the text associated with the transition. Notice how the event Close can be handled by two different states; Listening and Established. In this example they are both doing the same thing, changing state to Closed. However it is possible for different states to execute different actions for the same event.

If an event occurs which is not associated with a transition to the current state the definition of the finite state machine itself does not declare what should be done. It is the choice of the designer to decide what action should be taken. One could prefer to discard events that are not applicable for the current state. Another alternative is to raise an error or an exception.

Another way of representing a state machine is by using a matrix called a transition table [5]. Table 1 is a transition table which corresponds to the previous UML-diagram.

<table>
<thead>
<tr>
<th>State</th>
<th>Acknowledge</th>
<th>Close</th>
<th>Open</th>
</tr>
</thead>
<tbody>
<tr>
<td>Established</td>
<td>-</td>
<td>Closed</td>
<td>-</td>
</tr>
<tr>
<td>Listening</td>
<td>Established</td>
<td>Closed</td>
<td>-</td>
</tr>
<tr>
<td>Closed</td>
<td>-</td>
<td>-</td>
<td>Listening</td>
</tr>
</tbody>
</table>

One important thing to note in both representations is that both of them hold information merely about the transition of states. They do not contain any information about underlying actions taken for the events. That is, they only show us the control flow of the state machine. Any other action, which does not affect state but can be equally important to the overall functionality of the application, is not represented.

2.1.1. **Ways of implementing the finite state machine**

There are numerous ways of implementing a finite state machine in program code, each with its advantages and disadvantages [11].
2.1.2. Procedural

The most common way to implement state machines using a procedural approach is to use two case statements, a while loop and a state variable [8]. A simple implementation of the TCP connection example is suggested on the next page, implemented in a procedural way with double case statements.

```
states = {Established, Closed, Listening};
begin
    state = Closed; // initial state
    while true do
        event = getPendingEvent();
        switch(state)
            case Established:
                switch(event)
                    case Close:
                        state = Closed;
                    end
                end
            case Listening
                switch(event)
                    case Establish:
                        state = Established;
                    case Close:
                        state = Closed;
                    end
                end
            case Closed
                switch(event)
                    case Open
                        state = Listening;
                    end
                end
        end switch;
end while;
end
```

This approach is quite straightforward and also has reasonably high performance. It is implemented as two levels of case statements. The first level is used to establish what state the system is currently in and the other to determine what event has been raised. The code might seem structured at a first glance but the control flow of the program is modeled by means of assignment to the state variable, which in many ways resembles a goto statement. This kind of programming is generally agreed to be bad practice.

There are other procedural ways of implementing FSMs with a procedural approach. One could for example put the loop inside the individual cases instead of outside the case statement. The result would be the same.

All procedural implementations of finite state machines suffer from a fundamental drawback; redundant code is sometime needed to create the case statements.
Background

Also the complexity of the case statement obviously increases when the number of states and transitions increase, this way of implementing the program would prove extremely hard to maintain for applications the size of MTsim. Despite of the disadvantages mentioned there are reasons when one would prefer a procedural approach. If the case statements very rarely change and the state machine itself is reasonably small, the procedural approach would likely give the best performance and also be the cheapest to implement.

2.1.3. Object oriented

Another way for implementing finite state machine is by means of the object oriented paradigm [1][10][11][17][18]. Just like the procedural the object oriented approach offers a variety of different ways for implementing a FSM.

One common way is by using a state variable and to let each transition correspond to a virtual function\(^1\) [10]. The virtual function then selects the proper action with regard to the state variable. Another way is by using the state pattern (described in Appendix A). The state pattern models the states as classes that are instantiated as objects [4]. The state machine then delegates the events to its current state. There are several advantages to this approach.

- It will be easier to handle changes to the state machine [18].
- Objects may be allocated in advance thus avoiding the need to instantiate objects and having to allocate memory at every state change [10].
- In Java (and other languages that allows nested classes) it will be possible to avoid having to pass a back pointer to the context class to the handling state if state classes are implemented as internal classes in the context [10].

For further explanation, background and justification of these advantages read Appendix A, [10] and [18].

---

\(^1\) A virtual function, or a virtual method, is determined, at run time, to be the function or method furthest down in the inheritance hierarchy of the object on which is was called. This is a very important construction in object oriented languages.
In Figure 2 the TCP network connection examples is designed with the State pattern. There are some important things to notice.

- States are modeled as classes.
- Transitions are modeled as method calls into the state.
- Every state needs to implement every possible event since they are all implementing the interface State.

The object oriented ways of implementing state machines generally makes for implementations that are more easily maintained. For smaller state machines the performance of the implementation can be slower with the object oriented approach, however the object oriented approach scale better which makes for faster programs when the state machines grow large. Object oriented implementations are not always slower than procedural implementations even for smaller FSMs, the reason is that the changing of a reference to a virtual function (the handling method of the state object) is usually faster than to use the lookup table that the case statement would be compiled into. However it is not easy to state in general which is faster since it depends on how the code is compiled and how it is executed.

The approach of using the state pattern for implementing a finite state machine can be extended with other patterns to create more flexibility and will be discussed later in the report.

### 2.1.4. Issues when implementing FSMs

Whichever approach one chooses for FSMs there are some common problems that need to be dealt with. One problem is that some parts of the code will be hard to reuse [17].
Background

There are other issues that need to be addressed, Gurp and Bosch [6] presents three of these and suggests solutions to them. They are discussed below.

2.1.4.1. Evolution of FSM

One of the challenges that software engineers will face when working with FSMs is that they tend to change over time, the problem being that making these changes is often difficult and might result in a cascade of other changes that will be needed as well. This problem is called Evolution of finite state machines by Gurp and Bosch.

Table 2 Common changes made to FSMs presents the most usual changes that can be required on a finite state machine and their implications.

<table>
<thead>
<tr>
<th>Change</th>
<th>Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding states</td>
<td>Creates need for adding or changing transitions.</td>
</tr>
<tr>
<td>Removing states</td>
<td>Will require changes made to transitions to or from the state.</td>
</tr>
<tr>
<td>Changing states</td>
<td>Can mean that changes need to be done for actions associated with the state.</td>
</tr>
<tr>
<td>Adding events</td>
<td>Will require changes made to transitions.</td>
</tr>
<tr>
<td>Adding transitions</td>
<td>It is possible that code should be reused from other transitions.</td>
</tr>
<tr>
<td>Removing transitions</td>
<td>Usually has no effect on other states or transitions.</td>
</tr>
<tr>
<td>Changing transitions</td>
<td>Changes to a transition will likely require changes to the source or target state. Also actions associated with the transition might need to be changed.</td>
</tr>
</tbody>
</table>

With a procedural implementation adding and changing states or transition requires a lot of changes to the code. It will also bring redundant code into the software since much code needs to be copied between different cases [6].

If we use a state pattern implementation it would mean less trouble adding states, it would however create some other problems [3]. In the case of the state pattern implementation several classes would need to be edited when we want to add events to the state machine and the context classes would need to be edited to support the new events. In some cases it could also require new states to be added. The reason for this chain reaction of changes is that the concept of transitions in state pattern is implemented as methods.

The only FSM concept that is properly represented in the state pattern is the concept of states. Transitions and events are not directly represented. Events could be seen as method headers and actions as the methods themselves. This is not good news for the designer of
the FSM, since it means that many of the changes that will be needed will also result in a lot of source code to be edited. Changes to transitions or event would also need a recompilation of the software.

### 2.1.4.2. Finite state machine instantiation

Another problem that arises with applications using state machines is something Gurp and Bosch call instantiation [6]. It refers to the challenge of handling multiple instances of state machines in a system. The example that Gurp and Bosch use is a TCP protocol that needs to handle 30000 connections on one system, one for each port. If each of these connections is modeled as a state machine, instantiation becomes very important.

To solve this problem we need to identify the parts of the state machine which are unique for each instance and what parts that are not. The unique parts of each instance are usually the current state and the context data associated with it. These will be referred to as *extrinsic data*. The rest of the FSM is not unique to the instance and could therefore be shared among all the instances. These parts will be referred to as *intrinsic data*.

It is not desirable to make a copy of the entire state machine for each instance. This would be a very ineffective use of memory [6]. Instead the Flyweight pattern (see Appendix A) could be used to handle the sharing of the extrinsic parts of the FSM.

### 2.1.4.3. Data management

Another challenge discussed by Gurp and Bosch is the problem of data storage. Without support for FSM instantiation, data must be separated from transitions to allow transitions to be shared among instances.

There are two places where data is stored when using the state pattern. Either data is stored in the context or within the state. Data stored in the state is only accessible locally within the state. Data from other states is not accessible. This means that data would instead be stored in the context. However this is against one of the basic principles of object oriented programming, data stored in a central place should be avoided. Also this would mean tighter coupling of the context and state, making it impossible to reuse states for different contexts since the state is dependent on the context. Tight coupling of classes and objects should, if possible, be avoided in object oriented programming.
3. SMILE

TSS Language is a proprietary Ericsson programming language commonly referred to as SMILE. SMILE is based on the principles of state matrix programming [19]. This means that the each program written in SMILE is in essence a FSM.

3.1. Structure of a TSS test application

![Figure 3. Structure of TSS test application](image)

Figure 3 describes what parts constitute a TSS test application. In the middle is the test case [20]. The test case is the component that binds everything together. It contains references to the other parts of the application. The test case also contains information about how many instances should be created for each test program (see section 3.2. for details).

- **Test input files.** The SMILE language supports a special kind of variable, the so called parameter. Parameters are available for assigning values from outside the actual application through set input files [20]. If a test input file containing parameters is connected to a test program, the values of the parameters specified will be assigned as initial values in the test program. The test input files are optional.

- **Log settings file.** Another kind of file that is optional to bind to the application is a Log settings file. This file is used for logging debug information or protocol data. Log settings is then enabled or disabled globally in the test case.
• **Global include file.** The global include file is a file containing variables that are global, which means they are accessible from all test program instances. The global include file is also optional.

• **Message database.** The message database is a collection of files containing the structure of the messages that can be used in the application. They are essentially text files strictly formatted in accordance to message file syntax.

### 3.2. Test programs

The TSS test programs are often represented as a state matrix where each cell contains *action code* [19]. Action code is the actions that are executed when an event is raised. Action code is thus associated with an event and a state. The states are always denoted by positive integers. Furthermore at the launch of a new instance of a test program a special section of code named *Init* (for initialization) is executed. Once this is done the test program is automatically set to state 0 and a reserved event called *start* is invoked.

To illustrate the concept of the state matrix let us denote the state Closed from our TCP network connection example as state 0, the Listening as state 1 and the Established state as 2. The state matrix might look like Table 3.

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acknowledge</td>
<td>NEXT_STATE(2);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Close</td>
<td>NEXT_STATE(0);</td>
<td>NEXT_STATE(0);</td>
<td></td>
</tr>
<tr>
<td>Open</td>
<td>NEXT_STATE(1);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In this example the state matrix is almost identical to the transition table presented in Table 1. However, whereas the cells of the transition table only contained the next state of the system, the cells of the state matrix allow program code, action code. In this example however they are equivalent since the only thing the action code does is to change the state.

Notice that we introduced an event called Start, the reserved event that is always called when the initialization of the test program is done. The test program is also always set to state 0 initially and that is the reason why the Closed state was mapped to state 0 in this example.
Figure 4 shows the basic structural layout of a Test program.

![Test program structure](image_url)

- **Declare section.** The declare section of a program is used for declaration of variables, constants, events, procedures and functions [19].

- **Init Section.** The init section is the part of the program that is first executed when a test program is started. It is instance specific and does not allow the use of local variables [19].

- **Action code section.** The action code section of the program contains the code which will be executed when an event is invoked [19]. A small sample action code section is displayed below. This example only prints a message to the console and then changes the state to state 1. The action code will be executed when the Test program is in state 0 and the event called open is received.

```plaintext
BEGIN 0, open
   DISPLAY("TCP connection is set to open.
   ");
   NEXTSTATE(1);
END;
```

### 3.2.1. TSS Test programs as FSMs

As mentioned earlier each TSS test program instance is essentially each a FSM. This means that each instance of a test program will more or less execute independently and only interact with other test program instances through message passing and shared variables. They have no other connection which means that different instances of the same test program can be in different states at the same time and thus act differently and independently to incoming events.
The TSS test programs are modeled such that action code can only be associated with state and event. FSM implementations could allow state-independent action code but this is not the case for TSS test programs and it will not be the case for the translated code.

3.3. Basic language elements

3.3.1. Literals

Literals are a source code representation of a value, a nameless constant, which can be used in the source code without prior declaration or computation. In both Java and SMILE literals are used to assign values to primitive data types.

3.3.1.1. Integer Literals

In SMILE integer literals can be written in three different radices: binary (base 2), decimal (base 10) and hexadecimal (base 16) with the decimal being default [19]. They cannot be larger than what can be represented by 32 bits.

3.3.1.2. String literals

String literals are in SMILE represented by a sequence of characters from the International Alphabet No 5 (IA5, 7-bit ASCII) enclosed in double quotes. The maximum length of a string literal is 255 characters.

Two special characters exist, the sequence ‘\n’ (backslash character together with an n character) represents newline and ‘\\’ (two following backslash characters) is used to represent the backslash character. Both of these special sequences are regarded as one character each.

3.3.1.3. BCD literals

SMILE has support for BCD (Binary Coded Decimal) literals which are sequences of characters preceded by the prefix BCD’. The sequence is allowed to be a maximum of 50 characters and the characters must be digits from the range 0-9 or the character F (or f).

The F (or f) character denotes the filler which can be specified to a value from B’0000 (H’0) to B’1111 (H’F). The default value for the filler, if not specified, is H’F. The command BCD_SET_FILLER procedure is used to set the value of the filler.

3.3.1.4. Hexadecimal string literals

SMILE support strings of hexadecimals. They work similar to BCD literals but can contain up to 500 characters each from the normal hexadecimal alphabet (0-9, a,b,c,d,e and f). Hexadecimal strings are prefixed with HS’.

* Java supports automatic boxing of some classes. This means that some classes (for example Integer, String and Float) can be instantiated with literals without using the new keyword which is necessary for all other classes.
3.3.1.5. Array literals
SMILE support three different types of array literals; decimal array, string array and BCD array. The arrays can contain up to $16777216 (2^{24})$ elements and the notation for them are the same:

\{<\text{literal 1}>, <\text{literal 2}>, ..., <\text{literal n}>\}

3.3.2. Identifiers
Identifiers in SMILE must start with a character, can be up to 32 characters long and contain only letters, numbers and the underscore character (_).
Identifiers can be written in the source code to have names longer than 32 characters, but in that case it is truncated to the first 32 characters at compilation.
SMILE is not case sensitive; therefore IDENTIFIER_1 is regarded equal to identifier_1.

3.3.3. Data types
There are only three data types used in SMILE [19].

- integer
- string
- BCD

It is also possible to construct structures and arrays of these data types.

For the integer type it is possible to define what length the integer should have, how many bits to use. Programmers can specify this during declaration of the variable. It is also possible to use one of the three predefined alternatives; INT (32 bit integer), SHORT (16 bit integer) or BYTE (8 bit integer). The integers of SMILE are always unsigned. It is not possible to use integers with more than 32 bits; the only way to do this is by creating structures of smaller integers.

The BCD data type is a string of binary coded decimals.
3.3.4. Operators

Table 4 presents the operators in SMILE.

**Table 4 Operator precedence in SMILE**

<table>
<thead>
<tr>
<th>Id</th>
<th>Priority</th>
<th>Symbol</th>
<th>Description</th>
<th>Operands</th>
<th>input data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>~</td>
<td>Bitwise not</td>
<td>1</td>
<td>integer</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>NOT</td>
<td>Logical not</td>
<td>1</td>
<td>logical expression</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>*</td>
<td>Multiplication</td>
<td>2</td>
<td>integer or BCD</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>/</td>
<td>Division</td>
<td>2</td>
<td>integer or BCD</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>%</td>
<td>Modulus</td>
<td>2</td>
<td>integer or BCD</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>+</td>
<td>Addition</td>
<td>2</td>
<td>integer or BCD</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>-</td>
<td>Subtraction</td>
<td>2</td>
<td>integer or BCD</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>=&gt;</td>
<td>Right shift</td>
<td>2</td>
<td>integer or BCD</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>&lt;=</td>
<td>Left shift</td>
<td>2</td>
<td>integer or BCD</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>&gt;</td>
<td>Greater than</td>
<td>2</td>
<td>integer or BCD</td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td>&lt;</td>
<td>Less than</td>
<td>2</td>
<td>integer or BCD</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>&lt;=</td>
<td>Less or equal to</td>
<td>2</td>
<td>integer or BCD</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>&gt;=</td>
<td>Greater or equal to</td>
<td>2</td>
<td>integer or BCD</td>
</tr>
<tr>
<td>14</td>
<td>7</td>
<td>=</td>
<td>Equal to</td>
<td>2</td>
<td>integer, BCD or string</td>
</tr>
<tr>
<td>15</td>
<td>7</td>
<td>/=</td>
<td>Not equal to</td>
<td>2</td>
<td>integer, BCD or string</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>&amp;</td>
<td>Bitwise and</td>
<td>2</td>
<td>integer</td>
</tr>
<tr>
<td>17</td>
<td>9</td>
<td></td>
<td></td>
<td>Bitwise or</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>10</td>
<td>AND</td>
<td>Logical and</td>
<td>2</td>
<td>logical expression</td>
</tr>
<tr>
<td>19</td>
<td>11</td>
<td>OR</td>
<td>Logical or</td>
<td>2</td>
<td>logical expression</td>
</tr>
</tbody>
</table>

All arithmetic operators can take mixed operand types; this means that it is possible to multiply a BCD value with an integer value. The result of such an operation is always considered to be a BCD value.

The *equal* and *not equal* operators compare two operands for equality; they cannot take operands with different types.
Another special case is the subtraction operand which is not allowed to result in a negative value. This is due to the restriction in the integer data type itself. Such a subtraction would not result in a runtime error but the result would be undefined.

### 3.3.5. Conversions

It is possible to convert from one data type to another. However there are, as Table 5 presents, some things to note regarding conversions.

<table>
<thead>
<tr>
<th>From</th>
<th>To integer</th>
<th>To string</th>
<th>To BCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>No restrictions</td>
<td>No restrictions</td>
<td>Characters of the string must be ‘0’- ‘9’, ‘F’ and ‘f’.</td>
</tr>
<tr>
<td>string</td>
<td>Must be possible to interpret the string characters as an integer.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCD</td>
<td>BCD value cannot be too large, must fit into a 32 bit integer.</td>
<td>No restrictions</td>
<td></td>
</tr>
</tbody>
</table>

### 3.3.6. Variables

SMILE has support for variables with different scope.

- **Global variables.** Global variables are specified in a global include file, or in the declaration section of a test program, and are accessible to all test program instances in the test case.

- **Common variables.** Common variables are specified in the declaration section (see section 3.2) of a test program and are accessible to all instances of a specific test program.

- **Parameters.** Parameters are integer variables that from within the test program look just like common variables. They can have a value specified in the test program code but can also have values assigned to them through the use of test input files. Test input files have a list of the parameters and values to assign to them.

- **Instance variables.** Instance variables are specified in the declaration section of a test program and are accessible only to one specific instance of a test program.
• **Local variables.** Local variables can be specified inside sections of action code and are accessible only from within that particular piece of action code.

3.3.6.1. **Variable scope**

It is not allowed to define common, instance and global variables with the same name. It is however possible to have local variables in action code and procedures that have the same name as a common, instance or global variable. If this is the case it is the local variable that will have priority.

It is possible to have local variables inside action code segments, procedures and functions with the same name as global, instance or common variables. If this is the case it is the local variable that will be used.

It is not allowed to have global, common or instance variables with the same name as other variables of these types. This will cause an error during SMILE compilation.

<table>
<thead>
<tr>
<th></th>
<th>Global</th>
<th>Common</th>
<th>Instance</th>
<th>Local</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Local is used</td>
</tr>
<tr>
<td>Common</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Local is used</td>
</tr>
<tr>
<td>Instance</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Local is used</td>
</tr>
<tr>
<td>Local</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
</tr>
</tbody>
</table>

3.3.7. **Control flow**

There are many possible control flow statements in SMILE [19]. Most of the control flow constructs that would normally be expected exist. It is possible to use IF-THEN-ELSE statements, normal loops with DO-WHILE statements, SWITCH statements, GOTO statements and RETURN statements. They all behave in the way one would expect from other programming languages.

3.3.8. **Messages**

Messages have a key role in the test application [19]. Messages generate the events which drive the flow of the programs. When a message is received the run time environment checks if it matches any defined event. If a match is found the action code associated with that event is executed. If there is no match, the message is discarded.
SMILE

The following figure is taken from the TSS User’s Manual [19] and shows the general process of messages and events.

![Figure 5 Incoming message events](image)

If there are matches for an event in several instances of a test program the event will be handled only by the first match found. This is not always the desired behavior and there is a special construct called TP_REFERENCE which makes it possible to send the event to multiple, or specific, instances.

### 3.3.9. Timers

Timers can be used to control the behavior of a test program. Timers can also be used for error handling, for example to register timed-out network requests. Timers are declared in the declaration block. It is possible to set an expiration time for the timer at declaration; if this is not done the time must be supplied when the timer is started. When the timer expires it generates an event which in turn decides what action code should be executed.

### 3.3.10. Procedures and functions

SMILE has support for procedures and functions. Both must be declared with a body in the declare section of a test program. Procedures are macros, which means that any resource available at the procedure call in the code is also available inside the procedure. Functions on the other hand work similar to functions in other programming languages. They work independently of outside conditions. There are other differences between procedures and functions; functions can return a value, which a procedure cannot, functions allow recursive calls, procedures do not. Also arguments passed to procedures must all be integers while arguments to functions can be of any SMILE data type.
3.3.11. Events

There are two main different types of events that can be declared in a test program:

- Message events
  - Ordinary
  - Uncorrelated
  - To-all-instances
  - Broadcast
- Timer events

3.3.11.1. Message events

Message events are sent to a test case, it is then matched against events that are declared in test programs. The event declaration works somewhat like a filter that compares criteria of the event with incoming messages in order to be able to identify the event and execute the right action code associated with it.

The declaration specifies what primitive is expected to be received, parameters for this primitive and what data the message should carry. This means that the way a message is handled is determined when it is received by a test program and not when it is sent.

As seen in the list above there are four types of message events possible to be declared. Uncorrelated events are different from ordinary message events. With uncorrelated events the receiving instance cannot be determined immediately. These events are buffered temporarily and later passed to the correct instance.

To-all-instance message events are sent to every instance of a test program.

Broadcast event is the last type of message events and they are sent to each instance of a test program which has the same broadcast channel radio signaling link as the broadcast event. There are no broadcast events in MTsim and this type of event will therefore not be implemented in the implementation phase of this thesis project.

3.3.11.2. Timer events

Timer events can be used to trigger events after a given time or at given intervals. Timer events are put in the same event queue as message events. This means that timer events are not always very exact since there can be other messages to be handled before the timer event.
3.4. Test Program Compiler

The TSS Test Program Compiler (TPC) compiles the TSS application into a binary form that is executable by the TSS interpreter. All programs that are going to be executed must first be compiled.

![Function block for the Test Program Compiler (TPC)](image)

Figure 6 shows an overview of the TPC. It consists of several function blocks that all have specific functionality in the compiler.

- **Main.** The main function block is responsible for handling all communication with the other function blocks and calls upon them when a test program shall be compiled.
- **Declare.** The declare unit parses the declaration block of a test program and from it generates a symbol table.
- **Parser.** The parser function block parses tokens from the scanner function block. The parser utilizes the test program language specification to build an Abstract Syntax Tree (AST) from the tokens from the parser. The output from the parser is a complete AST representing the complete test program.
- **Scanner.** The scanner takes the source code of a test program and translates into tokens where each token is in a format used by the parser.
• **Analyze.** The analyze function block checks the AST from the parser for semantic correctness. The analyzer also decorates the AST with further information that is needed to generate binary code.

• **Generate.** The generate function block uses the decorated AST, decorates it further with more information, and then generates the actual binary output from the resulting AST.

• **Support.** The support function block is a collection of functionality that is used by the other function blocks. It has functionality for handling memory, the AST and the symbol table.

• **Lister.** The lister has functionality for error handling and reporting. The functionality from this block is used in all other function blocks.

3.4.1. **Dataflow**

Figure 7 shows the basic data flow of the TPC. The TPC takes a test program as input and in the end generates a binary file.

3.4.2. **Abstract syntax tree**

The TPC constructs an abstract syntax tree (AST). An AST is an intermediate stage in the process of translating the source code to a binary format. The AST is successively assembled when the TPC scans the source code and then further decorated in each step through the data flow of the TPC.

The AST is a binary tree where each parent node is an operation or other significant language construct, and the children are nodes representing operands or parameters.

The code below presents a simple assign statement; further down is the abstract syntax tree that would represent the same code.

```plaintext
a = b+5;
```
In the TPC the AST looks a little different.

There is no actual difference in the trees except the actual representation. There is another way of representing the AST, in textual form. It is possible to print the textual representation of the TPC AST during any phase of the compilation by using different input flags to the TPC. The code section on the next page is the textual representation of the same syntax tree extracted from the TPC.
3.5. TSS Interpreter

The TSS Interpreter is responsible for executing the binary code, compiled test programs, outputted by the TPC in a possibly very vast amount of concurrent instances [22]. It is the run time environment for all SMILE programs and handles everything from message decoding, scheduling, timer handling, run time error handling, communication and I/O. No further details are needed about the TSS Interpreter since its implementation will not be directly adapted to the implementation done in this thesis project.
Basic language translation of SMILE to Java

4. Basic language translation of SMILE to Java

One should keep in mind that the main reason for the interest in automatically converting SMILE code to Java code is to be able to translate the application MTsim to Java. Thus some constructions that are specified as legal or possible in the SMILE language will not be implemented in the Java conversion if it is very rarely, or never, used in MTsim. SMILE has been around for very long time and if some parts of the language have not been used up until this point it probably means that programmers do not need those constructions. This might be a dangerous assumption to make, but it will be evident that it is an acceptable assumption in most cases. When the issue arises it will be discussed for each individual case.

4.1. Language comparison

In many aspects SMILE and Java are quite similar. They both, for example, use strong static typing. Both are also imperative and well structured. The main difference is that SMILE is a state machine based language, where code is executed inside action code sections. Fortunately this maps very easily to the object-oriented paradigm which Java is based on.

The conversion of the basic syntax of SMILE to Java will not pose a major problem. The challenge lies in handling the more complex constructions of SMILE; i.e. the functionality that today lies in the interpreter. This has to be implemented as a framework in Java and the translation to this framework will be further discussed in Section 5. Table 7 Summary of SMILE and Java language comparison

<table>
<thead>
<tr>
<th>SMILE</th>
<th>Java (J2SE6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradigm</td>
<td>Automata-based/State-machine based</td>
</tr>
<tr>
<td>Execution</td>
<td>Compiled into intermediate language and interpreted</td>
</tr>
<tr>
<td></td>
<td>Compiled into Java bytecode and executed on virtual machine</td>
</tr>
<tr>
<td>Strong typing</td>
<td>yes</td>
</tr>
<tr>
<td>Static typing</td>
<td>yes</td>
</tr>
<tr>
<td>Reserved keywords</td>
<td>yes</td>
</tr>
<tr>
<td>Parameter passing</td>
<td>by value</td>
</tr>
<tr>
<td>Platform independent</td>
<td>yes</td>
</tr>
<tr>
<td>Platform dependent</td>
<td>yes</td>
</tr>
</tbody>
</table>

* However objects in Java are treated as pointers, this can lead to unexpected behavior for SMILE data types are translated into Java classes. See section 4.7.1 Parameter passing for further discussion of this.
4.2. Translating literals

4.2.1. Occurrence of literals in MTsim

Table 8 shows the number of occurrences of some of the more uncommon literals in MTsim.

Table 8 Occurrences of some types of SMILE literals in MTsim

<table>
<thead>
<tr>
<th>Type of literal</th>
<th>Occurrences</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCD literal</td>
<td>317</td>
<td>35</td>
</tr>
<tr>
<td>Binary Integer literal</td>
<td>1132</td>
<td>47</td>
</tr>
<tr>
<td>Hexadecimal Integer Literal</td>
<td>4540</td>
<td>109</td>
</tr>
<tr>
<td>Hexadecimal String literal</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hexadecimal Array literal</td>
<td>507</td>
<td>23</td>
</tr>
<tr>
<td>BCD Array literal</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

As the table shows the more exotic literal types Hexadecimal string and BCD array literals are never used in MTsim. BCD array literals can be constructed in Java the same way as String array literals are constructed so this will not require much effort in implementation. Hexadecimal string literals is however a quite obscure concept. On the one hand it is a string, on the other a hexadecimal value. It is a string which is only allowed to have certain characters. This literal type will probably never be used and if it is needed it is easy for the designer to use normal strings instead. Thus Hexadecimal String Literals will not be supported by the designed Java framework.

**DESIGN CHOICE 1.** Hexadecimal String literals will not be supported by the Java framework and they will not be supported by the translator.

4.2.2. Integer literals

Java has a language primitive data type called `int` which is always 32 bit (4 byte) large, Java’s support for different radices is a bit more restricted than in SMILE, the notation for both languages are presented in Table 9.
As the table shows the decimal representation of integer literals are the same in SMILE and Java and could thus be translated directly. The hexadecimal representation does not pose much of a problem as the only difference is the prefix.

The challenge comes when translating binary integer literals from SMILE to Java since this representation is not supported by the Java language.

There are two solutions to this problem

- Use the static method `parseInt` from the Java Integer class. This method returns a Java primitive `int` from a string representing the digits of the integer and an `int` representing the radix.

  - First convert the binary literal into hexadecimal in the translator and then directly translate this so the Java equivalent hexadecimal representation of an integer literal.

Both of the solutions offer advantages. The advantage of using the first solution is that the actual representation of the bits would be intact in the Java source code thus making it easy to directly see what specific bits are set.

The advantage of the second solution is that, after the translation to hexadecimal, translation of all integer literals from SMILE would translate into literals in Java.

It will have to be decided at translation time which of the two options is most suitable.

### 4.2.3. String literals

String literals are represented the same way in SMILE and Java. Java strings can be of greater length than 255 characters and this constraint on SMILE strings does not affect the translation. The newline and backslash characters in SMILE (`\n` and `\\`) are
represented the same way in Java. Also all of the 7-bit ASCII characters available in SMILE can also be used in Java strings.

There is no primitive data type for strings in Java, the string literals are therefore all references to the Java class String.

4.2.4. BCD literals
Java does not support any kind of BCD literals. It is clear that this type of literal from SMILE cannot be used in Java as literals and must be modeled into objects

DESIGN CHOICE 2. BCD values will be modeled into a Java class.

4.2.5. Array literals
Array literals from SMILE can all be translated directly into the Java equivalence where the internal literals from SMILE is translated individually as presented above.
Basic language translation of SMILE to Java

### 4.2.6. Summary of literal translation

Table 10  Summary of literal translation

<table>
<thead>
<tr>
<th>Literal</th>
<th>SMILE source code</th>
<th>Java source code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal integer</td>
<td>1023</td>
<td>1023</td>
</tr>
<tr>
<td>Hexadecimal integer</td>
<td>H´2A3F</td>
<td>0x2A3F</td>
</tr>
<tr>
<td>Binary integer</td>
<td>B´10011110</td>
<td>Integer.parseInt(“10011110”,2) or 0x9E via translation to hexadecimal literal.</td>
</tr>
<tr>
<td>String</td>
<td>“String with backslash \ and a newline \n.”</td>
<td>“String with backslash \ and a newline \n.”</td>
</tr>
<tr>
<td>BCD</td>
<td>BCD´0123FF</td>
<td>Must be modeled into object, for example: new BCDObject(“0123FF”)</td>
</tr>
<tr>
<td>Hexadecimal string</td>
<td>HS´01abef</td>
<td>Not translated to Java</td>
</tr>
<tr>
<td>Decimal array</td>
<td>(1,912,12,0)</td>
<td>(1,912,12,0)</td>
</tr>
<tr>
<td>String array</td>
<td>(“first”, “second”)</td>
<td>(“first”, “second”)</td>
</tr>
<tr>
<td>BCD array</td>
<td>{BCD´0123FF, BCD´99}</td>
<td>{new BCDObject(“0123FF”), new BCDObject(“99”)}</td>
</tr>
</tbody>
</table>

### 4.3. Translating identifiers

Identifiers in Java can be of arbitrary length and can contain any letter, digit or underscore. There are two problems that need to be addressed when translating identifiers.

1. Two different names can be used for the same identifier in SMILE if the names are more than 32 characters long and only differ after the first 32 characters.
2. There are names that cannot be used for Java identifiers which are allowed in SMILE.
Table 11  Reserved words that are not allowed as identifiers in Java

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract</td>
<td>assert</td>
<td>boolean</td>
<td>break</td>
<td>byte</td>
<td></td>
</tr>
<tr>
<td>case</td>
<td>catch</td>
<td>char</td>
<td>class</td>
<td>const</td>
<td></td>
</tr>
<tr>
<td>continue</td>
<td>default</td>
<td>do</td>
<td>double</td>
<td>else</td>
<td></td>
</tr>
<tr>
<td>enum</td>
<td>extends</td>
<td>false</td>
<td>final</td>
<td>finally</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>for</td>
<td>if</td>
<td>goto</td>
<td>implements</td>
<td></td>
</tr>
<tr>
<td>import</td>
<td>instanceof</td>
<td>int</td>
<td>interface</td>
<td>long</td>
<td></td>
</tr>
<tr>
<td>native</td>
<td>new</td>
<td>null</td>
<td>package</td>
<td>private</td>
<td></td>
</tr>
<tr>
<td>protected</td>
<td>public</td>
<td>return</td>
<td>short</td>
<td>static</td>
<td></td>
</tr>
<tr>
<td>strictfp</td>
<td>super</td>
<td>switch</td>
<td>synchronized</td>
<td>this</td>
<td></td>
</tr>
<tr>
<td>throw</td>
<td>throws</td>
<td>transient</td>
<td>true</td>
<td>try</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>volatile</td>
<td>while</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The solutions to these two problems are:

1. Truncate any identifier in SMILE to a maximum 32 characters before translation.
2. Change the name of any SMILE identifier that would not be allowed as a Java identifier.

**DESIGN CHOICE 3.** SMILE identifiers must first be truncated to 32 characters and then checked against the Java reserved words and changed appropriately.

### 4.4. Translating data types
As has already been decided the BCD data type of SMILE must be represented by some kind of class in Java.

#### 4.4.1. Translating unsigned integers to Java
There is reason to argue that the integer data type of SMILE also should be boxed inside a class. The reason for this is that integers in SMILE are always unsigned, but unsigned integers are not supported by Java. In the Oak Language specification [14], which is the specification of the language which eventually evolved into Java, a note declares “unsigned isn’t implemented yet; it might never be.”, and indeed it never was.

This poses a major problem for our translation to Java. The unsigned/signed dilemma does not affect all of our operations. It does not affect subtraction, addition for example, since these are same in both signed and unsigned arithmetic (assuming we do not have
Basic language translation of SMILE to Java

overflow or underflow). Multiplication would not pose a problem since Java discards the upper half of the product. Bit shifts to the right and left would not change since Java has support for an unsigned version of right shift. The equality checking operations (== and !=) would all be unaffected since they work as bit comparisons.

Division and modulo (remainder) would not be translatable since they differ in unsigned and signed arithmetic. The ordered comparison operators (>, <, >= and <=) would also not act the same if the integers are regarded as signed or unsigned.

Table 12  Unsigned operator problems in Java

<table>
<thead>
<tr>
<th>Operator</th>
<th>32 bit integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>store</td>
</tr>
<tr>
<td>+ -</td>
<td>addition, subtraction</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td>%</td>
<td>modulo / remainder</td>
</tr>
<tr>
<td>==, !=</td>
<td>equality</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;=, &gt;</td>
<td>comparison</td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;, &lt;&lt;, &gt;&gt;</td>
<td>shift</td>
</tr>
</tbody>
</table>

There are some points, or solutions, that could be of interest with regard to this problem.

1. Promote all SMILE integers to a Java int type of greater size (type long).

Comment: This solution is feasible but seems unattractive. It is possible since Java support 64 bit integers (long) and the largest integer types in SMILE is 32 bit. It would therefore always be possible to convert the SMILE integer to a Java long.

This would mean that we would, after the translation, support 64 bit integers but all integers would be signed. The obvious drawback is that all our integers, which are most of our variables, will require twice the space in memory.
The value of the old integers and the operations on them would still be functioning as expected, unless the old SMILE code made use of the modular nature of the computer arithmetic\(^2\).

Another reason for not doing this is that the Java Virtual Machine is optimized for using normal Java int. The reason is that this primitive data types is used a lot more frequently by programmers than the type long.

2. Promote the integer to a bigger version of the int data type and mask the bits which are not needed.

Comment: As with the previous solution it would be possible since Java support 64 bit integers (long) and the largest integer types in SMILE is 32 bit. We would also have the same memory problem as with the previous solution.

There would have to be additional code inserted every time any of the affected operations was going to be used. This would lead to unattractive code which would be hard to read. Alternatively we could implement our own versions of the problematic operators.

3. Create a class which handles the unsigned integers.

Comment: This solution is also possible but might seem just as unattractive. We could create a class which holds an integer value and also a flag indicating if it is signed or unsigned. Another way is by not using the flag indicator for signed and unsigned and instead let every instance be treated as signed and then instead force the operations on this class to handle the problem.

The third alternative has many advantages. First of all we protect the developer using the Java framework from the actual problem; we hide the problem inside the class. Also it would give us flexibility regarding the size of the integers. As described earlier SMILE integer variables can have different sizes which are declared at declaration time. This will be much simpler to handle with a class than directly from the Java source code. Another advantage is that we can create other operations or functionality for this data type by designing new methods for the class. One example where this might be needed is if we ourselves need to serialize variables into bit or byte streams to be sent over network connections.

\(^2\)In most statically typed languages, as in SMILE and Java, arithmetic on integers are modulo, which means that the number line is enclosed wrapping the end to the beginning. If a program kept adding 1 to an integer it would eventually get back to the start, 0 in SMILE and \(-2^{31}\) in Java.
Basic language translation of SMILE to Java

There is one major drawback with this solution though; it will seriously decrease maintainability and readability. Using objects for simple things such as integers will create extra lines of source code, code that will most likely not be needed since the unsigned/signed problem will probably only pose a real problems in very few situations. Since the framework prototype should be created with further development and maintainability in mind this solution should, if possible, be avoided.

With these considerations in mind it is clear to see that a combination of the solutions will be required. This will increase the complexity of the translator and might also need user interaction during the translation process.
### 4.5. Translating operations

Table 13  Operator precedence in Java

<table>
<thead>
<tr>
<th>Id</th>
<th>Priority</th>
<th>Symbol</th>
<th>Description</th>
<th>Operands</th>
<th>data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>~</td>
<td>Bitwise not</td>
<td>1</td>
<td>int</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>!</td>
<td>Logical not</td>
<td>1</td>
<td>boolean</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>*</td>
<td>Multiplication</td>
<td>2</td>
<td>int</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>/</td>
<td>Division</td>
<td>2</td>
<td>int</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>%</td>
<td>Modulus</td>
<td>2</td>
<td>int</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>+</td>
<td>Addition</td>
<td>2</td>
<td>int</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>-</td>
<td>Subtraction</td>
<td>2</td>
<td>int</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>&gt;&gt;=</td>
<td>(signed) Right shift</td>
<td>2</td>
<td>int</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>&lt;&lt;=</td>
<td>(signed) Left shift</td>
<td>2</td>
<td>int</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>&gt;</td>
<td>Greater than</td>
<td>2</td>
<td>int</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>&lt;</td>
<td>Less than</td>
<td>2</td>
<td>int</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>&lt;=</td>
<td>Less or equal to</td>
<td>2</td>
<td>int</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>&gt;=</td>
<td>Greater or equal to</td>
<td>2</td>
<td>int</td>
</tr>
<tr>
<td>14</td>
<td>6</td>
<td>==</td>
<td>Equal to</td>
<td>2</td>
<td>int, boolean</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>!=</td>
<td>Not equal to</td>
<td>2</td>
<td>int, boolean</td>
</tr>
<tr>
<td>16</td>
<td>7</td>
<td>&amp;</td>
<td>Bitwise and</td>
<td>2</td>
<td>int</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>&amp;</td>
<td>Logical and</td>
<td>2</td>
<td>Boolean</td>
</tr>
<tr>
<td>18</td>
<td>8</td>
<td></td>
<td></td>
<td>2</td>
<td>int</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>boolean</td>
</tr>
</tbody>
</table>

When comparing the table of precedence in Java to the corresponding table for SMILE (Table 4 Operator precedence in SMILE) it is possible to identify one difference that could become a problem during translation. Bitwise operations have precedence over logical operations in SMILE but have the same priority in Java.

In fact it is not possible to mix integer and boolean operators in Java in the same way as can be done in SMILE. SMILE allows this mix since logical expressions are in fact the
Basic language translation of SMILE to Java

same as integers. The SMILE constant \texttt{TRUE} is the same as 1, and the constant \texttt{FALSE} is the same as 0. Therefore any expression that uses a mix of logical and numerical operations must be handled by the translator and cannot be translated directly. The translator must identify where a logical expression is used in a numerical context and where numerical values are used in a logical context. There is a simple solution for when a logical expression is used in a numerical context. The Java language has the so called ?: operator which does just this. It uses a boolean to determine what value should be evaluated.

<table>
<thead>
<tr>
<th>SMILE</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical value \texttt{a} used in a numerical context</td>
<td>\texttt{a?1:0}</td>
</tr>
<tr>
<td>numerical value \texttt{a} used in a logical context</td>
<td>\texttt{a&gt;0}</td>
</tr>
</tbody>
</table>

4.6. Translation of control structures
Control flow constructions in SMILE have almost exact equivalents in Java. What follows is a presentation of the control structures and how they will be translated into Java source code.

4.6.1. IF-THEN-ELSE statements
The use of IF-THEN-ELSE statements is specified in the TSS Language reference as:

```
IF <expression> THEN
<statements>
[ELSE <statements>]
[ELSEIF <expression> THEN
<statements>]
[ELSE IF <expression> THEN
<statements>
FI;]
FI;
```

To illustrate the use let us consider some examples:

```java
// Example 1
IF a < b THEN
    EXIT("Failure");
FI;

// Example 2
IF a <= b THEN
    DISPLAY("Failure!");
ELSE
```
Let us see how these examples would translate into Java:

```java
// Example 1
if (a < b) {
    exit("Failure");
}

// Example 2
if (a <= b) {
    display("Failure!");
} else {
    display("OK!");
    exit("OK");
}

// Example 3
if (a > 12) {
    display("Too much!");
} else if (a == 10) {
    exit("OK");
}
```

As we can see the syntax is almost identical and does not pose a problem for translation.

### 4.6.2. ON statements

The ON statement is the SMILE control flow construction normally named FOR. It is in the TSS Language reference specified as:

```
ON <loop_variable> FROM <start_value>
[UPTO | DOWNTO <end_value>] DO
<statements>
NO;
```
Basic language translation of SMILE to Java

This is translated into Java as the code below shows:

```java
// The equivalent Java code for an ON loop using UPTO
for (int <loop_variable> = <start_value>; 
     <loop_variable> <= <end_value>; 
     <loop_variable>++){
    <statements>
}

// The equivalent Java code for an ON loop using DOWNTO
for (int <loop_variable> = <start_value>; 
     <loop_variable> >= <end_value>; 
     <loop_variable>--){
    <statements>
}
```

### 4.6.3. DO-WHILE statements

The DO-WHILE statement in SMILE works just like the WHILE statement in Java. It is in SMILE specified as:

```smile
DO WHILE <expression>
<statements>
OD;
```

The same code would in Java look like:

```java
while (<expression>){
    <statements>
}
```
4.6.4. SWITCH-CASE statements

The SWITCH-CASE statements in SMILE can only switch on integer values. The statements have the following specification in SMILE:

```plaintext
SWITCH (<expression>) ON 
CASE (<int_val_1>)  
    <statements>
    [CASE (<int_val_2>)  
        <statements>
        .
        .
        CASE (<int_val_N>)  
        <statements>]
DEFAULT  
    <statements>...
NO;
```

It would translate into Java as the following code shows:

```java
switch (<expression>) {
    case <int_val_1>:  
        <statements>
        break;
    case <int_val_2>:  
        <statements>
        break;
        .
        .
        .  
        case <int_val_N>:  
        <statements>
        break;
    default:  
        <statements>
        break;
}
```
4.6.5. GOTO statements

GOTO statements are allowed in SMILE. It is generally considered to be bad practice to use GOTO statements when writing code since it decreases readability, the ease of debugging and testing. A simple search of the source code for MTsim tells us that GOTO is indeed used in MTsim. It is used 64 times across 5 files. This should be considered to be very restricted use of the construct.

GOTO statements will not be allowed in the Java framework and if they are found in the SMILE program they will require user interaction in order to change them into more appropriate code.

**DESIGN CHOICE 4.** The goto construction of SMILE will not be translated into Java.

4.7. Procedures and functions

4.7.1. Parameter passing

Parameter passing by value means that the actual parameter is fully evaluated and the resulting value is copied to a location used to hold arguments or parameters during function or method execution. Passing parameters by reference means that instead of copying the parameter value the internal variable in the method or function only acts as an alias for the parameter.

There is a common misconception that parameter are passed by reference for objects and by value for primitives in Java. There are some reasons why this conclusion is often made, it is however incorrect. Parameter passing is in Java strictly done by value, however since all variables holding objects in Java are in fact pointers, or references, it is these and not the actual objects that are passed. Thus in some cases it might look like Java uses parameter passing by reference and not by value when the parameter is an object. Java is strictly passed by value, exactly as C.

Arrays work the same way. Arrays themselves are passed as reference since they are not objects.

The fact that all variables are pointers makes for some interesting scenarios.

Consider the following example:

```java
public void foo(Dog d) {
    d = new Dog("Fifi");
}

Dog aDog = new Dog("Max");
foo(aDog);
```

In this case, since Java is using pass by value, the variable aDog is not affected by calling the foo method. The internal variable d is specified by the Dog keyword in the argument
list to be a pointer to a Dog object. The variable d is a copy of the pointer that points to the aDog object. However when we make a new object inside the method, we create a new object in a new memory space and it does not in any way affect aDog.

Now consider another example where we change the foo method to:

```java
public void foo(Dog d) {
    d.setName("Fifi");
}
```

In this case aDog will be affected since the pointer d also points to the same Dog object as the aDog pointer variable does.

If the equivalence of the examples above would have been written in SMILE, the aDog variable would have been unchanged in both examples. This obviously poses a problem for the translation.

The problem is quite restricted however. The problem can only arise if the arguments to a function or procedure translated from SMILE takes arguments that we have decided to translate into classes. If we decide to translate all SMILE integers to normal Java integers the problem is restricted to translated SMILE functions and not procedures. The reason is that SMILE procedures only allow arguments that are integers. Using the primitive Java data type integer will behave exactly the same as it would in SMILE.

With this in mind the problem only arises when we are passing strings or BCD values into functions, thus strictly limiting the number of occasions this problem can occur.

To identify where this problem can occur we look for variables that exist in the context that is calling the function and that are assigned inside the function. If this happens we need to make a copy ourselves of the inbound object which can be used inside the function.

We are in essence creating our own version of passing arguments by value for Java objects, this is not very good practice but will only be necessary for a restricted number of translated functions and will never be needed for future development of programs.

**DESIGN CHOICE 5.** All data types that are translated into Java classes must supply a method for making a new object which is an exact copy of itself.

**DESIGN CHOICE 6.** If an argument that is passed to a function is assigned a value inside the function it should first be copied and the copy of the argument should be used locally. This applies only to arguments of SMILE data types that have been translated into Java classes.
Design of a common Java Framework

5. Design of a common Java Framework

An object oriented framework is a reusable design that is defined as a set of abstract classes and interfaces in a way corresponding to how objects of these are collaborating [13]. The framework is often expensive to create and must be easy to learn in order to be useful. It should also comprise of functionality that is unlikely to change but must at the same time offer hooks for features that will change over time.

The definition that will be adapted in this thesis comes from Michael Mattson's book Object-Oriented Frameworks [12].

“A framework is a generative architecture designed for maximum reuse, represented as a collective set of abstract and concrete classes; encapsulated potential behavior and subclassed specializations.”

With this definition it is clear that the concept of a framework differs from other concepts that are used in software architecture. Object oriented design patterns, pattern languages, class libraries are for example not frameworks. The differences to some of these concepts will be discussed later in this report.

There are basically two types of frameworks;

- **White box frameworks**. White box frameworks are relying on inheritance as the main instantiation technique. The instances are made through subclassing the appropriate class from the framework. The main drawback of this approach is that the applications using the framework need to know about internal class details of the framework. It is this fact that the name white box framework refers to. As with white-box testing [2] the user (or tester in the case of testing) needs to have knowledge about the internals of the framework (or software under test).

- **Black box frameworks**. Black box frameworks differ from white box frameworks in the sense that the internals are not necessarily known to the user programs. Instead of using inheritance as the main technique for instantiation composition is used. This gives higher modularity as composition might, if needed, be changed during run time.

Generally a black box approach should be used if possible. There are two reasons, it hides the implementation of the framework from the user program and it makes for a higher degree of modularity. This follows from the basic object oriented principle which states that software architects should favor composition over inheritance [4].

A good framework will reduce the cost and difficulty of creating applications within the application domain [16]. The need for a good framework becomes greater the more applications that is to be created in this domain and the more functionality they need to share. If the aim is to create a single application it might not be very useful to first create a framework or function libraries to support it. If the aim however is to be able to create more applications in the same domain then spending time and resources to create a robust framework will pay off. There are difficulties in creating a good framework. The
framework must be simple enough for developers to easily understand it in order to take advantage of its functionality but at the same time the framework must have enough features for it to be useful.

In this thesis the aim is to translate SMILE code into Java code. This could be done without using a framework and instead directly translate every SMILE program into a large Java application. This would however require that many features would have to be implemented over and over again for every SMILE translation and would make the translator very complex.

Another alternative is to create a larger framework with all functionality needed. This would mean that the translation process would not be as costly. The end result would have higher maintainability and extendibility.

Figure 10 shows the expected complexity of the translator and the framework in relation to each other. If more functionality is put into the framework, thus increasing its complexity, we can expect a less complex translator and vice versa.
Design of a common Java Framework

5.1. Hollywood principle
There exists a distinction between two different kinds of frameworks, the calling framework and the called framework [13]. Calling frameworks are active; they invoke methods or actions into lower level modules, or modules using the framework. The called framework is passive; it offers methods and functionality to be called from the application that uses the framework. Generally frameworks should often be created as being calling frameworks. This is in accordance with the object oriented design principle known as Inversion of control. What this means is that control flow is inverted compared to normal imperative sequential programs. Instead of the programmer specifying when a specific action should be invoked the user program lets the framework know the desired response to specific happenings and offers the control to the framework. The user program implements resources or functionality that is used by the framework. The principle is sometimes also referred to as the Hollywood principle which in essence is the same concept. It takes its name from the clichéd response that actors seemingly often hear at auditions: “don’t call us, we will call you”.

Inversion of control is usually what makes the difference between a framework and a library. A library can be considered a passive, called framework, since it is only offering functionality to be called from the user program.

There is however problems with the Hollywood principle and inversion of control, one of them occurs when composing two or more calling frameworks [13]. When this is the scenario often all of the composing frameworks are expecting to be in control of the application which could lead to incorrect behavior.

5.2. Responsibilities of the framework and run-time environment
Having a common run time environment on which the test programs and test cases can run would be the minimal requirement for successfully running translated programs on the JVM. However there are reasons to aim at a more complex and complete framework and run-time environment, something similar to the current solution.

There are two main functionalities that need to be made available to the programmer and user of the system. The first is a function library which can be used by the Java test programs. This library of classes, functions and methods should be used to generalize the architecture of the test programs and test cases. It must also make common functionality available such as sending and receiving messages, data types and events.

The other part that needs to be made available is to supply a platform, a runtime environment framework for the test programs. This part must handle message decoding, scheduling of test programs, network handling and execution of test programs.

The library and the runtime environment framework will share a lot of functionality which means that they will be well integrated into each other. Therefore the name framework will from now on refer to both the passive part of the framework (library part)
and to the run time environment (the active, calling part of the framework) in the rest of
the report.

The choice, and aim, for this project is that test programs should be as simple as possible
and make use of a framework that supplies all common functionality. The translator
should also be kept as simple and general as possible.

The following list presents the responsibilities that need to be implemented and handled
by the Java framework.

**Passive part of framework available to the application designer**

1. Providing interfaces or abstract classes that will be used for test programs
2. Providing interfaces or abstract classes that will be used for test cases
3. Provide interfaces or abstract classes for messages
4. Provide interfaces or abstract classes for events
5. Provide support for data types
6. Provide support for structures

**Active part of the framework available to the application designer**

7. Handle protocol ports
8. Receiving messages
9. Sending messages
10. Decoding messages
11. Handle events
12. Providing support for data types similar to those in SMILE
13. Schedule execution of test program instances
14. Execute test program instances
15. Handle timers

5.3. **General decision about the framework**

With the last couple of sections in mind it is clear that the framework needs to be
implemented as a rather complete and large framework. It should contain all of the
common functionality which the test programs will need so that they can stay small,
general and agile. This will also lighten the load on the translator.

Furthermore the framework should be constructed using the black box approach, which
means that details about the internal implementation should not be needed by the test
Design of a common Java Framework

programs, only interfaces functionality from the framework should be known. Inversion of control is also important. The design should strive to confirm to the Hollywood principle, to call functionality from the test programs and not the other way around in order for the framework to keep the control of the program flow.

5.4. Communicating with external software
The Java framework and run time environment need to be able to communicate with external software and hardware over a network. It needs to be able to receive and send messages to outside systems. The reason for this is that MTsim is very rarely alone. Its main purpose is to test other systems, which it needs to be able to communicate with. In today’s application MTsim external communication is done through the use of calls into C/C++ libraries. This is not a solution that will be adopted in the created framework; instead all constructions for communication with external units will have to be implemented in Java as parts of the framework.

5.5. Event kernel
The event kernel is the heart of the run time environment. It is the event kernel that controls the flow of execution. The event kernel will have the following responsibilities:

- Handle incoming connections
- Handle outgoing connections
- Handle internal messaging
- Handle timers

Since the run time environment in itself is not the main focus of the thesis no event kernel will be created instead one which already exists will be used. The event kernel, known as EKFW is used in similar projects at Ericsson. It has support for timers, communication through sockets and an internal mail system and blocking operations. The event kernel itself is very small and easily adaptable to what is needed in the run time environment in this project. One drawback is that it only utilizes one thread, there is no possibility of multithreading.

There is one modification made to the EKFW event kernel. The EKFW main loop normally checks all messages to be handled then processes the sockets. If there is no work to be done with regards to the sockets the event kernel will wait till there are incoming events on a socket. This has been changed for the framework. With the original implementation if internal messages invoked actions that sent new internal messages, the new messages would have to wait to be handled until after the sockets had been processed. This meant that some internal messages could be waiting forever to be delivered if there was no communication on any of the sockets. This has been changed. Instead all internal
messages are handled after which the queue for internal messages is checked once more and new messages are handled. This is repeated until there are no more internal messages pending. This way there are no internal messages queued, allowing a chain of messages to be sent internally, before the sockets are processed. Thus the state machines will always have been processed and run to a point where they are either waiting for external events (over sockets) or have finished before sockets are processed.

5.6. Message decoding

Message decoding is a problem which must always be implemented in software that communicates via messages. This is something that can be done in endless different ways and with endless variation. Message decoding is very often very critical in regards to overall performance of systems since the process of identifying messages can be very time consuming. This is also the case for MTsim which has a very complex, but effective, way of decoding incoming messages. The implementation in MTsim is in fact so complex (and would also not work as well for an object oriented approach) that it could not be directly adapted by the framework developed in this thesis project. This meant that a completely new message decoder had to be developed which turned out to require much more time and effort than initially projected.

The aim of the message decoder is to handle incoming messages and find out if there is any event that matches this message. If an event matches a message the event is raised and handled by the correct test program instance.

Each event has a test program associated with it along with a match pattern which is used to match incoming messages.

5.6.1. Messages

Messages are implemented as a chain of fields. There are three types of fields implemented; string field, integer field and length field. These can be combined in any way to create a message structure with the only restriction being that string fields always have to be preceded by a length field which specifies the length of the string.

Messages that are created initially does not have any values assigned to the fields, they are in essence only specifying the structure of the message. The message is assigned values to its fields from the test program during run time.

Messages have two important functionalities; first of all they can be packed which means that they are converted into appropriate streams of bytes which can be sent over a socket. The other important functionality is to return a so-called match pattern. The match pattern for a specific message is used, as the name suggests, to match incoming streams of bytes to specific messages in order to identify what message has been received and where it should be handled.
5.6.2. Message matching

In order to match incoming messages the framework uses match patterns. The concept of match patterns is based on using an object, MatchPattern, which in essence is a complete message. In order to match incoming messages with these patterns one has to know the length of each expected field and the expected position in the byte stream of the data for that field. Position and length of fields can be either absolute or relative. Relative positions mean that the position of the field depends on the position and length of previous fields. Relative lengths mean that the length of a field depends on the value of some previous field, a length field. The reason for using relative positions and lengths is to be able to match messages which have string fields with lengths not known or not considered.

Let us consider a complete example where we look at a message that does not contain any user-data part, only containing a message primitive. Let us use a simplification of the MTsim message INTERNAL_REQ as an example. INTERNAL_REQ is used in MTsim to send data to a specific instance of a test program. It allows user data to be sent as payload but this will be disregarded in this example.

<table>
<thead>
<tr>
<th>Type of field</th>
<th>Name</th>
<th>Description</th>
<th>Size</th>
<th>Value assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>INTERNAL_REQ</td>
<td>The internal id for the message type. This will determine how the message is interpreted. It is however important to note that different systems, or different instances of the framework can interpret messages with the same id in different ways.</td>
<td>2 byte</td>
<td>automatically</td>
</tr>
<tr>
<td>Length</td>
<td>TPNameLength</td>
<td>The length of the following string field</td>
<td>2 byte</td>
<td>automatically</td>
</tr>
<tr>
<td>String</td>
<td>TestProgram</td>
<td>The name of the receiving test program</td>
<td>variable</td>
<td>user assigned</td>
</tr>
<tr>
<td>Integer</td>
<td>InstanceNumber</td>
<td>The instance number for the receiving instance</td>
<td>2 byte</td>
<td>user assigned</td>
</tr>
</tbody>
</table>

This means that the message, when packed into a stream of bytes, would have the structure as the table below shows.
Table 16. Packed representation of INTERNAL_REQ message primitive

<table>
<thead>
<tr>
<th>Field</th>
<th>Byte position</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 and 2</td>
<td>The id for the message primitive. In the case for INTERNAL_REQ it is the integer value 8.</td>
</tr>
<tr>
<td>2</td>
<td>3 and 4</td>
<td>The length of the string of field 3</td>
</tr>
<tr>
<td>3</td>
<td>5 through n</td>
<td>The name of the test program</td>
</tr>
<tr>
<td>4</td>
<td>n+1 through n+2</td>
<td>The instance number of the receiving test program instance</td>
</tr>
</tbody>
</table>

Thus the message could for example look like [8, 12, “TestProgram1”, 2].

Now assume that we, in the test program TestProgram1 instance 2 want to catch all incoming messages that are INTERNAL_REQ with the string field set to TestProgram1 and instance number set to 2. This is called tagging. We are tagging the fields 3 and 4 with the values that we want them to match for incoming messages. To do this we create a message and, instead of packing it into a byte stream, we create a MatchPattern from it. This is a set of conditions, linked together in a list, of how a byte stream should look in order for it to match the given MatchPattern. For this MatchPattern we set the tags TestProgram to the value “TestProgram1” and the field InstanceNumber to the value 2.

When this is done, we have a MatchPattern that can be used to match incoming messages and thus make sure that they are handled by the correct instance.

The byte buffer, which is the incoming message, is passed to the match pattern which then passes it along to the first condition in the list. If this condition is satisfied, it passes the byte buffer to the next condition in the list and so on. Each condition corresponds to one field of the message which has been declared to be of interest for matching when creating the match pattern. If the last condition in the match pattern is satisfied, this means that a match has been made and the associated event can then be raised.

The reason why message matching was chosen to be implemented this way is that a message can contain multiple fields that have relative lengths. What this means is that we cannot directly specify, in our pattern, which bytes to look at in the incoming byte buffer since the exact position of fields can be dependent on values of other fields.

5.7. Multiple threads

It is desired to be able to have the framework handle multiple threads. That is something which will not be implemented in this project but will be discussed for the future.

The reason why multithreading is important is scalability for the future. The trend in processor technology seems to be going in the direction of adding more processor cores instead of focusing on increasing clock frequency. What this means is that in order for the
Design of a common Java Framework

MTsim application to be scalable for the future it needs to support multiple threads that can be run in parallel on different processor cores.

The SMILE run time environment, the TSS Interpreter, had its own scheduler. The interpreter itself ran in one process but with the implemented scheduler it could in essence run test programs in parallel. This is some kind of virtual multithreading and might not be the best solution.

Java has support for concurrency and this could be utilized by the framework. What this means more specifically is that the framework could implement some kind of scheduling functionality which can schedule execution of test program instances.

How the actual concurrency should be implemented is not obvious. Multithreading in Java works differently on different systems. There are however some considerations that needs to be made. First of all, handling a large amount of threads can be expensive and can create large amount of overhead. Each Java thread has its own stack which would mean that memory usage would increase significantly. The size of the stack is different on different platforms but generally defaults to sizes between 128kB to 512kB. It is possible to specify how large stack you want your threads to use and it is often possible to get the stack down to 32-64kB. Another concern is the garbage collection in Java. Garbage collection is the process done by the Java Virtual Machine (JVM) that reclaims dynamically allocated memory. Garbage Collection (GC) has impacts on performance. Each thread has its own local part of the heap meaning that programs with a lot of threads will consume a large portion of the total heap. This would mean that garbage collection would run more often which could decrease performance. A third problem to take into consideration is the actual cap of number of possible threads in Java. The Java threads are often very closely coupled to the operating system threads meaning that if the operating system has a limit for maximum number of threads, this will also apply for the Java threads, thus will find that different platforms allow a different number of maximum running Java threads.

If every test program instance would run in its own Java thread it would mean a huge amount of individual threads. This is the main reason why this model should not be chosen. Another argument for not mapping test program instances directly to Java threads is to try not to have the behavior of the framework be dependent on the platform on which the JVM is running.
The framework should be able to handle a user defined number of threads and then use some kind of scheduling to map test program instances to these threads. This will allow running multiple test program instances in parallel and also the utilization of Java concurrency. It will however create new requirements on the Java framework.

5.8. Existing frameworks

There exist multiple frameworks that model FSMs. Some are more advanced than others. Some are expensive and some free. This section presents some of the frameworks that exist for Java today and discusses the possibility of using one of these for the future complete implementation of a conversion of MTsim to Java. There are many frameworks for FSMs that are not discussed here. Some frameworks are too general in nature but a larger portion of FSM frameworks existing today are too specific to their application.

The reason why it was chosen to create an entirely new framework for this thesis project, opposed to using an existing framework, was to keep control of the complexity and to be able to choose what functionality the framework had. This might not have had been possible if an existing framework was used.

Another reason for creating a new framework was to keep it simple. Only include functionality that is needed and not to be forced to work with constructs or functionality which will not be used in the conceptual prototype for this thesis.
Design of a common Java Framework

However, for the future, if it is decided that a full scale automated translation of MTsim from SMILE to Java is to be made it might be worthwhile looking into using one of the frameworks discussed below.

5.8.1. UniMod
UniMod stands for Unified Modeling and consist of a Java Finite State Machine Framework and an Eclipse plug in [21]. The project is open source under GNU Lesser General Public License (LGPL) and driven as a SourceForge project.

UniMod allows the creation of applications through a set of Class and state chart diagrams which in turn generate a XML description of a state machine which is executed by a run time environment. UniMod is built to execute UML state charts.

UniMod also has support for translation of the state machines into program code but currently only support Symbian C++.

UniMod is quite well tested and well supported. It has all of the features which one would like in a state machine framework and this is also why the software is used at other projects at Ericsson.

This might be an option for future implementations of the framework.

5.8.2. jfsm
jfsm is a Java.net project, its goal is to offer a FSM code base which has founding in research. It has many similarities to the framework that was developed for this project since both frameworks are based on the same type of research.[7]

The development of jfsm was test driven, the framework is design pattern focused and it was built to be very independent from other products and platform independent.

The drawback of this framework is that it is currently in early alpha stages, which means that it has not been very well tested and does not have all features that could be needed.
5.9. Overview of the Java framework runtime environment

The design of the framework, on a logical level, is quite simple. In the heart of the design is an event kernel which handles all events and reacts to them. The event kernel also handles selection of sockets and handling of timers. It handles messages, both internal and external.

The second major part of the design is the Message decoder which is discussed in section 5.6. The program handler keeps references to all test programs which are being executed, waiting to be executed and which has finished execution. The actual scheduling of execution is done by a scheduler which schedules the execution in the execution service.
5.10. Implement a prototype of the Java framework

In this section we present the prototype framework which was the result of the implementation part of the thesis project.

5.10.1. Test case

The test case, as we know, should hold information about what test programs should be started and references to what message database to use, any parameters to use and so on. In our prototype framework the only responsibility of the test case is to hold information of what test programs to start. Even though the test case implementation in the prototype is so thin it was decided to keep it since it might be used in the future for other reasons.

In our implementation we provide a TestCase interface for the developer to use.

There are two kinds of messages; messages which are sent over sockets and internal messages. Both internal message and messages which are received on sockets are routed to the right test program instance through the message decoder.
As we can see its only use is to return a table of test programs to launch, each test program is also identified by a name.

An example of how a simple implementation of a test case could look is presented in the code segment below.

```java
public interface TestCase {
    Hashtable<String, TestProgram> getTestPrograms();
}
```

The implemented test case above is an application which is a simple client for NTI messages. We will later see what the actual test program, NtiClient, is.

### 5.10.2. Test programs

The discussions in section 2.1.1 and in section 2.1.4 lead to some conclusions. First of all the procedural approach has too many problems associated with it when dealing with large FSMs. The procedural approach is not an option for SMILE applications the size of MTsim which will require multiple very large FSMs to be handled.

The object oriented approach seems viable and will be the approach chosen.

The state pattern alone is not enough when seen in the light of evolution of FSM, FSM instantiation and data management. It seems that some kind of extended model using the State pattern, the Flyweight pattern and possible other patterns will be needed.

We use the Flyweight pattern in order to avoid an explosion of number of states that will be instantiated thus hopefully having better usage of memory. This however requires us to
find what data is intrinsic to the states and what data are extrinsic, what data should be handled by the states and what data should be kept by the context of the state machine.

The context of the state is the actual test program, the actual FSM. Therefore the context will know what state it is currently in and also be able to receive events via a FIFO queue. The context will also hold and handle the extrinsic state data.

Another important choice is to use the command pattern for transitions. As mentioned earlier the normal State pattern models the transitions as method invocations. This is not desirable for our framework since we do not know the actual name of the transition before the implementation. Also we saw when discussing the State pattern that all implementations of the state must have the same interface, thus we would need to interface all transitions in all states. The actions that are associated with events and the actions that are executed inside the states will therefore be implemented using the Command pattern.

In SMILE all actions that are executed are action code associated with a state and an event. There are no actions taken during transitions, all are done inside a state. Thus we can safely choose to only allow actions to be taken inside of states. The events then become objects which are used for data storage.

The simple diagram below describes the basics of the FSM implementation that is chosen. It makes use of the State, flyweight and the command design patterns.

![Diagram of FSM implementation](image-url)
- **State.** This class models the states of the FSM. In order to have multiple states in the state machine there must be multiple instances of the state. The state however does not in itself contain any large amounts of data; it is only used to handle the incoming events. In order to do this it offers a public method with which it is possible to associate actions to incoming events. The proper action is then executed when the public method dispatch is invoked along with an instance and an abstract event. The instance argument to this method contains all the intrinsic data that will be needed.

- **Instance.** The Instance class contains a reference to the current state (which can be changed from within executed action code with the public method setState). It is the Instance which gets a dispatch call invoked first; it gets the event as an argument and then passes this on to its current state.

- **IAction.** IAction is an interface which must be implemented by a class. The classes implementing this interface are then ready to run from a state instance. This way, classes that contain action code (implementing IAction) can be reused for different programs. Using IAction is actually a use of the Command Pattern.

- **TestProgram.** The test program is an abstract class that needs to be generalized by a concrete class. This class act as the central point for the FSM, it has the method for creating more instances which in essence creates a new FSM. The new FSM all share the same extrinsic data because of the separation of TestProgram and Instance.
5.10.3. Test program example

Let us look at an example. The example is a simple client which responds to incoming NTI messages.

```java
class NtiClient extends TestProgram {

    private class ConfReceived implements Action {
        public void execute(final Instance i) {
            final MessagePrimitive mp = i.getCurrentMp();
            i.nextState(establishedConnectionState);
            i.startTimer("ti", 3000);
            send(Net.dataReq(netHandle, 12,
                MyTestMessage.create("StringField", "Actual value"))) ;
        }
    }

    private class ErrorAction implements Action {
        public void execute(final Instance i) {
            display(i, "NTI error");
        }
    }

    private class InitAction implements Action {
        public void execute(final Instance i) {
            send(Net.connectReq("", "NET1", 3));
        }
    }

    private class TimeOutAction implements Action {
        public void execute(final Instance i) {
            display(i, "Timer event raised!");
        }
    }
}
```

The test program has a number of internal classes which all implement the interface Action. All classes that implement this interface can be regarded as action code since they all have a method called execute. It is up to the developer to choose if the action code class should be internal or external, in its own file. This has many advantages compared to how it was designed in SMILE. First of all, having action code blocks implemented as classes instead of methods means that they are much more easily reused. We can use the same action code for any number of test programs without ever duplicating code, something which was not possible in SMILE.

However by keeping the action code classes as internal classes we keep the same functionality as in SMILE where we, from within the action code class, can use test program variables.

Another thing to notice is that an object Instance is passed to the execute method. This handle to an Instance object is used to reach instance specific data, such as timers. When
we want to change state of the instance we also use the handle of the instance object as can be seen in the Java class.

```java
private final State initState, establishedConnectionState;
private Integer netHandle;
private final TimerEvent t1;
```

All test program implementations need to define states, which is done as private objects as shown above. All other objects or data that needs to be used on a test program level should also be defined here. In this example we have a TimerEvent t1, which we want to use on a test program level.

```java
public NtiClient() {
    super();
    initState = new State();
    establishedConnectionState = new State();
    t1 = new TimerEvent("t1");
    registerTimer(t1);
    final MessageEvent error = new MessageEvent("Error", this,
        Net.errorInd(0, 0));
    final MessageEvent connectConf = new MessageEvent("ConnectConf",
        this, Net.connectConf());
    initState.setAction(error, new ErrorAction());
    initState.setAction(connectConf, new ConfReceived());
    establishedConnectionState.setAction(t1, new TimeOutAction());
}
```

The actual constructor of the test program class is very interesting. It handles the initiation of the test program. In the constructor we setup states, events and actions. In this example we have two message events. Error and connectConf. Each message event is associated with a testprogram and a message. It also has a name as an identifier.

In order to react to incoming events we then set actions to the states, this is the actions that will be taken when the event is raised. This is where we instantiate our action classes which was described earlier.

```java
@Override
protected Action initAction() {
    return new InitAction();
}
```

The test program interface specifies that every test program should have a method called initAction(), the reason is that we want to force every test program to have a initial action
Design of a common Java Framework

code block. In this case we have an action code block called InitAction, however we could return any Action in this method.

```java
@override
protected State initState() {
    return initState;
}
```

Just as all test programs need to have an initial action, it also needs an initial state. That is the reason for the interface method initState().

5.10.4. Modeling variable scope

There are multiple levels of variables in SMILE. These must be modeled into the Java framework in such a way that the basic behavior is the same.

Local variables in SMILE do not pose any problems for the Java framework. These are modeled as normal Java local variables inside the Action object.

Instance variables are specific to the test program instance and must therefore somehow be specific for the context. They must however be available also from inside the individual states and actions. Thus we create a construct in the context that is able to handle requests for variables. States and actions then request variables of a specific name from the context and specify if they are requesting a global, common or instance variable.

Common variables should be available for all instances of a specific Test program. Just like with the instance variable we design an Object called TestProgram which keeps track of the multiple instances, the contexts, and which also handles requests for variables from the contexts. These can easily be implemented as private class variables in the concrete Test program implementation.

Global variables should be available from all instances of all Test programs. This however we leave to the client to implement with the requirement that the Test Program object should be able to request those variables.

One important thing to consider here is that where a program written in SMILE would use the same syntax to dereference global, common, instance and local variable we would in the Java framework need to have different syntax for them. This would mean that we need to know, at any given time, if a variable is global, common, instance or local before dereferencing it or accessing it. This should not pose a problem when compiling to the Java framework since SMILE declares the names of the variables once and they cannot be redeclared. The problem is therefore addressed by the translation process and not the actual framework.
5.11. Error handling
There are various errors that might occur. They are presented below along with a short description of how they are handled.

5.11.1. Run-time errors
Run-time errors are errors that exist in the user programs, the translated programs, and that are raised during execution of test programs. Run-time errors include out of index referencing of arrays, null reference when expecting an object or errors when sending and receiving messages. If the error occurs in a test program instance the error will be caught by the run-time environment which will then stop execution of that test program instance and try to continue with the other test program instances.

Even if the error is not directly raised inside a test program instance it might still be possible to identify which instance caused the error and again stop its execution and try to continue.

5.11.2. Run-time environment errors
Errors in the run-time environment of the framework are serious errors. There are different errors that can occur, some will be recoverable and some will not. Network timeout errors, or network write or read errors will not cause a very big problem since they are planned for and handled. However internal errors that occur, for example some run-time exceptions, will cause the program execution to stop.
5.11.3. System errors

Errors that are caused by the system, for example when system (JVM) routines fail will be handled by the application. The error will be printed or logged but the execution of the application will stop.
6. Translating SMILE to a Java framework

Since the main reason for this thesis is to investigate the feasibility and cost of a translation of programs written in SMILE to a Java framework this section could be considered the most important. In this section the construction of such a translator is presented.

6.1. Translating MTsim to the Java framework

MTsim is made up by approximately 650000 lines of code distributed over about 175 files. The average test program contains 3700 lines of code, but the variance is very large which means that there are a lot of very large files and many very small ones. This base of 650000 lines of code is the source code that is intended for translation.

The message database for MTsim, which would also be needed to translate, contains 500000 lines of code which are distributed over about 4400 files. The sizes of the message files are more evenly distributed with an average of 117 lines per file.

Thus the complete base of source code that would need to be translated is about 1.2 million lines of code. However the message files would not pose much of a problem. The syntax for messages is very strict and they are all very similar in structure.

6.2. Different approaches

6.2.1. Direct source code translation

One approach which would seem feasible and in fact possible would be to write a translator that takes a source code from one SMILE program, analyses this and creates the equivalent source code in Java. This would however require the implementation of a lexical analyzer, syntactical analyzer, semantic analyzer and a generator for Java code.

It would in fact require the creation of a completely new compiler, which compiles SMILE code to Java and since there is already a SMILE compiler, which compiles to binary code, this would mean reimplementing of something that already exists.

6.2.2. Translation via abstraction and reimplemention

Another approach of language translation is to do a reimplemention via an abstraction. What this means is that firstly an analysis of the language is conducted. This analysis is then used as basis for a total or partial reimplemention of the language. This might sound like a good idea since it might seem that many of the problems of today could be removed with a reimplemention. However, research has shown that in fact it is more likely that new problems are introduced when translating legacy code using reimplemention (which could be considered to be very similar to the task of this thesis project). The reason is that many problems which the original language, or original code, had have since long been fixed and these might be hard to spot during the analysis phase. In fact it is probable that many of the initial problems of the first implemention would
Translating SMILE to a Java framework

occur once more in the new implementation. The result is a new implementation but with old problems that have since long been forgotten about.

Also this method of translation would have very high costs. The reason is that both the analysis phase and the implementation phase would be very similar as when the program, or language, was originally designed. Then add the cost of fixing the introduction of new (and old) problems and bugs. This is therefore not a viable solution for this thesis project.

6.2.3. Translation via intermediate language

When translating one language to another it can sometimes be more effective and cheap to translate via another intermediate language. This is something that is often used in the industry today in many language translations. Often the intermediate language itself is not a real proper computer language that could be executed; instead it is language which describes other computer languages. If a translation is made to such a language there is another great gain. The abstract description, the representation in the intermediate language, of the program to be translated can be used over and over again and often into many different target languages.

6.2.4. Chosen approach for translation

The approach chosen in this project is to make use of the SMILE source code compiler to translate via the token language used by the interpreter and compiler which in turn is translated into Java. This is another example of translation via intermediate language. As previously discussed in the section on the SMILE interpreter the compiler outputs an AST which consist of short tokens which represent the program. This AST would normally then be further translated into a binary form that the interpreter can execute.

This intermediate language was originally designed to be used only internally in the TPC, however we can make use of it since it offers many advantages compared to translating directly from SMILE source code. One of the most difficult tasks, when translating program source code, is to syntactically and semantically analyze the source code. Since
this has already been done by the compiler we do not need to implement it and we can use it directly from the compiler. This is not entirely true since the token language, the intermediate language, also needs to be parsed. However this representation of the program is to a great extent easier to parse due to its simplicity and structure. One might ask why the SMILE programs are not written in this token form from the very beginning if it so much easier to parse. The answer is of course that it is also very much tougher to read and understand for a human. The token form has also many more lines of code than the original source code.

What needs to be designed and implemented in this project is what resides inside the box named Translator application in Figure 16. We need to be able to create a system that can take a TPC generated AST as input and convert this to the source code of the equivalent Java program. Furthermore we divide the system into two distinct tasks or processes. The first is to parse the AST and the second is to generate the Java source code. The reason why it is desirable to make this distinction is to follow the object oriented principles that we need to be ready for changes, but also that units should only do one task. If it, in the future, would be of interest to translate SMILE to another language it might be possible to reuse the same parser and only replace the actual translator which generates the source code of the target language.

6.3. Implementation of the parser

The input to the parser is a file with textual representation of the AST. This file is parsed, line by line, by the parser. The parser creates a new AST where each node is an object representing the corresponding node in the original AST. There is more or less a one to one relationship between the nodes in the original AST and the Java AST of the parser. The parser also needs to strip different parts of the textual AST output by the SMILE compiler. There are for example AST nodes which are not needed and meta-data for nodes (describing for example what part of the original source code a specific node of the AST was generated from).

The actual parser is a static method that is invoke with a file path as input. The method returns an AbstractSyntaxNode, which is the root node of the Java representation of the AST. Once we have an AbstractSyntaxNode, a tree structure of the original program represented as Java object, we have come a long way. This tree can then be used to generate Java code or a simple print to file or console. It is feasible that it would be possible to use this Java representation of the AST to generate source code in other languages, but this has not been examined in this project.

6.4. Implementation of the translator

For all different types of nodes that need to be translated there is a corresponding Java code generator class. When the translator is called with an AST tree as input it reads the top node (the AST root node) and identifies the code generator corresponding to that node. This is the actual translation of the program. Each translator knows how to translate a specific node, what code generator it represents, and also knows if it is supposed to call
Translating SMILE to a Java framework

Translators of child nodes. For example the code generator for a node representing an IF-statement knows that it should also translate the code blocks of the IF-statement and also Else-if and Else code blocks if they exist. These in turn handle any needed code generation of their child nodes and so on. This way every code generator only needs to know how to translate one specific type of node and they are easily replaceable. But the translation process is not complete just yet. The actual output of the translation process is not actual Java code. The output is yet another tree structure. This tree structure is a tree of code generators. It is this structure which is finally traversed, just as with the traversal of the Java representation of the AST, and which then generates the actual Java code.

6.5. Example
Let us look at a full example of translation from SMILE to our Java framework. We will use a very small program as an example since the AST output of the TPC can be quite large.

6.5.1. SMILE source code

```
DECLARE;
STRING VARIABLE stringvar;
DEFINE_EVENT (SYNACK_RECEIVED, INTERNAL_IND() mymsg(payload = H'02));

INT MyFunc(INT a, INT b)
BEGIN
  VARIABLE res;
  res=a-b+1;
  RETURN res;
END;
END DECLARE;

INIT
  stringvar = "Calling MyFunc with values %p and %p. The result is %p"
END;

BEGIN
  0,START
  VARIABLE localTestVariable = (2+2)*8;
  DISPLAY(stringvar, localTestVariable, 3, MyFunc(localTestVariable, 3));
  DISPLAY("Sending SYN");
  SEND INTERNAL_REQ(S_Testprogram = "Server", S_Instancenumber = 0)
    mymsg(payload = H'01);
END;

BEGIN
  0, SYNACK_RECEIVED
  BCD VARIABLE bcd_val;
  STRING VARIABLE str_val = "2113fF";
  VARIABLE int_val = 2001;
  DISPLAY("Received SYNACK, sending ACK");
  SEND INTERNAL_REQ(S_Testprogram = "Server", S_Instancenumber = 0)
    mymsg(payload = H'03);
  NEXT_STATE(1);
END;
```
Let us look closer at the code. The first part is the declare part, here a variable of the type string is declared. This is an instance variable and specific to the test program instance. Then an event is declared, it is called SYNACK_RECEIVED and is defined as an incoming INTERNAL_IND with the message mymsg that has the field payload set to H’02. The specification of the mymsg is found in another file, in the message database.

The declare section of the source code also has the definition of a function called MyFunc. The declare section is followed by the init section, the code which is always executed at the initialization of the test program instance.

What follows if the declaration of the action code for every specific event and state.

### 6.5.2. Original TPC AST

Since the original output of the TPC AST is so extensive only part of it will be presented here. The part which is presented here is the translation of the action code for the event START for state 0, which is the code between ‘BEGIN 0, START’ and ‘END’ in the original SMILE source code. The code presented here is also stripped of all the meta-data which the TPC puts into the AST in order to know what part of the original source code is related to which part of the AST.

```
(STATESPECLIST
 (STATESPEC
   (DECCONST '0')
   (IDENTIFIER 'START')
   (FUNCVARDECLS
     (VARDECL_INT
      (VARIABLE)
      (DECL_INIT
       (IDENTIFIER 'localTestVariable')
       (EXPR
        (EXPR
         (DECCONST '2')
         (ADD)
         (DECCONST '2'))
        (MUL)
        (DECCONST '8'))))))
 (STMTS
  (PROCCALL
   (IDENTIFIER 'DISPLAY')
   (EXPR
    (FIELDLIST
     (IDENTIFIER 'stringvar'))) 
   (EXPR
    (FIELDLIST
     (IDENTIFIER 'localTestVariable'))) 
   (EXPR
    (DECCONST '3')) 
   (EXPR
    (FUNCCALL_PF
     (identifier)))))
```
Translating SMILE to a Java framework

It is easy to see the structure of the AST and it is also easy to see how what part of the SMILE code was translated into what part of the AST. It is from this point that the Java code is generated.

For the full output from the TPC of the SMILE source code presented in 6.5.1 see Error! Reference source not found.

### 6.5.3. The Java code

Let us have a look at what the Java code after translation looks like.

```java
import primitives.Internal;
import smilemessages.MessagePart;
import fsm.Action;
import fsm.BCD;
import fsm.Instance;
import fsm.MessageEvent;
import fsm.State;
import fsm.TestProgram;
```
First is the import section of the application. Which packages and classes to include are dynamically added whenever needed during translation.

What follows is the actual class which is the TestProgram. Notice how the class extends TestProgram inheriting some basic functionality from the TestProgram class.

```java
public class Client extends TestProgram{
    private String stringvar;
    private State initState;
    private State s1;

    private Integer myfunc(int a, int b){
        int res;
        res = (a - b) + 1;
        return res;
    }

    private class start_0 implements Action{
        public void execute(final Instance i){
            Integer localtestvariable = (2 + 2) * 8;
            display(i, stringvar, localtestvariable,
                    new Integer(3), myfunc(localtestvariable,
                    new Integer(3)));
            display(i, "sending syn");
            MessagePart tmp = Internal.req();
            tmp.setValue("s_testprogram","server");
            tmp.setValue("s_instancenumber",0);
            tmp.setValue("mymsg", new MyMsg().setPayload(0x01));
            send(tmp);
        }
    }
}
```

As can be seen the blocks of action code matches very well with the action code blocks of the original SMILE program, however it is no longer as easy to set fields and data of the messages.

```java
private class synack_received_0 implements Action{
    public void execute(final Instance i){
        BCD bcd_val;
        String str_val = "2113ff";
        int int_val = 2001;
        display(i, "converting string to bcd variable");
        display(i, "received synack, sending ack");
        MessagePart tmp = Internal.req();
        tmp.setValue("s_testprogram","server");
        tmp.setValue("s_instancenumber",0);
        tmp.setValue("mymsg",new MyMsg().setPayload(0x03));
        send(tmp);
    }
}
```
Translating SMILE to a Java framework

```java
public Client()
{
    super();
    s1 = new State();
    initState = new State();
    final MessageEvent synack_received =
        new MessageEvent("SYNACK_RECEIVED",
            this, Internal.ind().setPayload(0x01));
    initState.setAction(synack_received, new synack_received_0());
}
```

The constructor of the test program, which is presented in the following piece of code, is the heart of the application. From here states, events and actions are bound together. In the code below we create a MessageEvent called synack_received which states that whenever an Internal Ind message with the payload set to 0x01 is received we shall invoke the action code which is the class synack_received_0.

This is exactly what the original SMILE code would do, but it is worthwhile to notice that it becomes a bit more complicated in the Java code.

```java
@Override
protected Action initAction() {
    return new start_0();
}
@Override
protected State initState() {
    return initState;
}
```

There are also two abstract methods of the TestProgram class. Two methods which all test programs must implement, these are initAction() and initState(). These methods are used for the framework to determine which state and which action are the initial ones which should be used at initialization. This gives a bit more flexibility than the original SMILE code since it is here possible to choose which state should be the first state.
7. Discussion

The main goal of the master thesis project has been to determine the feasibility and cost of an automatic conversion MTSim from SMILE to Java. The first conclusion is that the object oriented paradigm is very well suited as a base for state machine implementation. There are design patterns that are suitable to use in order to make the design more flexible and extendible. These patterns should definitely be considered for any state machine implementation in an object oriented environment.

It was very early clear that an automated conversion of SMILE as a language to Java would be possible. SMILE in itself is quite small and does not have any special constructions that cannot be implemented in Java. The big question, and the one with real value, was to determine, or at least approximate, the cost of such a conversion. Another interesting point was to determine if the result would be useable.

The software that is needed for a conversion can be divided into two parts, a run time environment and a translator. The main effort and cost will be needed for the framework since it is here where the main functionality lies. If the framework contains all the functionality and offers a simple syntax then the translator itself does not need to be very complex. This is thanks to the fact that the compiler that is used today to generate binary code from SMILE programs has the functionality of outputting an AST which can be used as an input to the translator.

The implementation phase of the master thesis resulted in a prototype for a Java framework for running translated test programs and also a translator which could take applications written in SMILE and translate them into the equivalent Java test program. This was tested on a handful of smaller real test programs with successful result but the main tests has been conducted with SMILE applications written specifically to test the translator and the framework.

7.1. Limitations

There are several limitations of the implementation in this thesis project. There are different reasons for them but mostly it is due to limited resources (time). Also since the implementation part of the thesis project was to design a prototype some features were chosen not to be implemented since they by themselves would not help us greatly in gaining an opinion about the feasibility of a total translation.

7.1.1. General framework

- No support for A3 and A8 (GSM & UMTS) ciphering (from LRM)
- No file handling
- No output to TCCT
- No support for logging
- No multi-threading which means no semaphores
Discussion

- No random numbers
- No tagging of messages in the send statement
- Nextstate command must have state argument
- No support for delaying execution
- No operator input

7.1.2. Messages
- Only internal messages supported
- No check for mandatory fields
- Only support for high-to-low encoding
- No support for structs in message fields
- No support for choice fields
- No support for set fields
- Integer fields only supported in big-endian
- No support for integer array fields
- Length fields only supported for string fields, and string field must follow the length field.
- No support for user data length field.
- No support for pointer fields
- No support for BCD fields
- No octet filler fields
- No SCCP encoding field
- Very limited support for data field attributes

7.1.3. Translator
- No support for GOTO-statements
- No support for BCD-strings
- No support for integer structures
- No translation of comments
7.2. Problems that need to be addressed

7.2.1. Unsigned integer issue

Section 4.4.1 Translating unsigned integers to Java presents what can be considered the toughest problem that needs to be solved before a complete translation can be made. Java’s lack of support for unsigned integers means that either an unsigned integer class needs to be implemented in the Java framework or the unsigned integers of SMILE need to be translated into signed integers. Both have drawbacks. The simplest solution, to translate all SMILE integers into class objects, have many advantages but at the same time has the very important drawback of making the code very complex. Furthermore it will make for very ineffective memory handling. The JVM was created with optimization for 32bit integers (the Java primitive integer type) in mind which should be utilized.

If one chooses to translate unsigned SMILE integers to signed Java integers there are other problems that need to be addressed. The loss of precision or introduction of bugs could be a problem but this is the solution to the problem that is probably going to create the most useful code and is thus the approach recommended.

7.2.2. Better modeling of variable scope

The current implementation of variables and variable scope gives the same behavior as the original SMILE programs. It is however quite ineffective since they are all stored in objects at different levels. Usage of a global variable for example would require multiple method calls in the Java framework as opposed to only dereferencing it directly in SMILE. This could prove to be a bottleneck for the overall performance. There are other possible designs that would be possible, one would be to let every test program instance have a reference to every global variable. This would however go against principles of object oriented design. It could however improve performance. Further study of possibilities for modeling variables and variable scope would be desirable but is not covered in this thesis project since its main aim is to determine the feasibility alone, not to create an optimal implementation.

7.2.3. Delay of execution

Currently it is not possible to delay execution of a test program. Primarily this comes from the fact that we cannot pause the execution of a block of action code. We cannot stop execution between two lines of Java code, keep running other test programs and then return to that same exact Java statement.

It is suggested that if this functionality is to be implemented it will preferably be integrated along with implementation of multithreading of the framework. This way we could use the internal functionality of waiting and delaying execution of threads in order to achieve delay of execution of test programs.
7.2.4. Translating comments

One important aspect of program translation is translation of comments in the original source code. This is a very difficult subject since the comments in the original code often are associated with specific methods, blocks of code, lines of code or other sections of the program. It is not obvious where these comments should be put in the translated code since it might not have the same structure. This could prove to be very difficult when translating program source code. At the same time this is a very important subject to this thesis since it is clearly stated in the proposed thesis goals that source code, once translated, should be maintainable and that it should be easy to keep developing in this new language. Source code comments are often extremely important for maintainability and further extensions and development since it offers the human who is programming detailed information about what the source code is meant to do.

In this case, the translation from SMILE to Java we are quite lucky that the main structure of the SMILE programs, the FSM structure of the programs, are quite easily translated into Java and the structure itself is in many cases kept the same. We have the same basic elements such as variables, action code, initialization sequences, message definitions, messages etc. Almost every element of the original SMILE language has a similar equivalent in the target framework. This should make translation of comments fairly simple.

There is one major problem however. Since the compiler does not have any use what so ever of the comments the first thing the compiler does when parsing the SMILE code is to throw them away. Since our translation takes the AST as input from the compiler we have lost all comments before the SMILE program even reach our translator.

Luckily there is a possible work around for this dilemma. The AST which is used as input to the translator has meta-tags at every AST token. This means that for every token in the AST there is a tag which clearly states where this exact command was translated from the source code. We have information about row and column of every token of the AST. If we take the original source code as input, along with the AST, to the translator we could then trace exactly which statement in the original SMILE code was translated to each part of the AST. This could be used to translate comments. We could parse the original source code for comments and for each comment associate it with a piece of code. This database of comments and which piece of code it was associated with could then be matched against the AST at translation in order to insert the comments in the target code. There is a standard which is followed in the MTsim code which states that comments should refer to the statement, or the block of code, which is located below the comment in the source code. This is the reason why this proposal for associating comments with parts of the code is feasible.

This is something that is currently not implemented in the scope of this thesis project but it is considered to very possible.
7.3. Suggestion for future improvements

7.3.1. Extend model for message primitives
The model of message primitives should be extended. The only two message primitives that are currently implemented are the Internal message primitives and the Net message primitives. The reason is that this was all that was needed for demonstration of the concept.

7.3.2. Extend model for user messages
The user messages in the prototype only support three field types, integer fields, length fields and string fields. This should be extended in any future model to hold any type of data that might be needed to be sent in messages.

7.3.3. Externalize message and primitive definitions
The current design uses Java classes for message and primitive definitions. This might not be desirable since it requires a recompilation of the software in order to add new messages or new message primitives. In order to avoid this, a solution using external definitions for messages and primitives could be used. One example is to externalize messages and primitives into XML files that could be read during run-time by the framework. This is similar to how the current system works using specific files for messages.

7.3.4. Scalability
Scalability was one of the more important subjects that were discussed at the very beginning of the project. The scalability issue was investigated briefly and then left for the future. The reason why this is not something that was addressed in this project is the time it would take. The project is on a tight time budget and it was decided that further investigation of the issue would cost too much time.

The result of the investigation was a suggestion of how multithreading could be implemented for the framework. The suggestion can be found in section 5.7.

7.3.5. Improve precision of recurring timers
Recurring timers are essential in the SMILE language and they are currently implemented in Java in a quite rudimentary way and do not provide very good precision. In order to improve precision they should be implemented to compensate for lack of precision from one timing event to the next. If the framework discovers that there is an offset in the time from the expected timer event to it is actually raised the framework should try to compensate for it.

The current design adds the time interval of the recurring timer to the current time as soon as the timing event has been triggered. This is not a very good solution since the loss in precision will increase for every recursion. Instead the recurring timer could keep track of how much difference there is between the actual time the timer was triggered and the
Discussion

expected time. This could be used, as mentioned before, by the framework to compensate for precision loss for the next time the timer triggers.

There are of course other more advanced and complex possibilities of improving the precision of recurring timers but the simple solution discussed above should suffice and give precision equivalent to that of the current system.

7.3.6. Making message definitions static

It is preferable to make message definitions static. This is because the definition in itself should not hold any data; it should only work as a template of how messages should be constructed. The suggested way of doing this is to have static methods that take all necessary data and return the corresponding buffer of bits or bytes.

7.4. Cost prediction

There has been an explicit request for an estimation of the amount of money or time needed in order to do a real translation of MTsim in its entirety and that is what will be presented here.

Remember that this is a very gross estimate based on how much time it would take for me to implement the specific feature. The time presented below includes unit testing.

<table>
<thead>
<tr>
<th>Task</th>
<th>Estimated time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message translator</td>
<td>7 weeks</td>
</tr>
<tr>
<td>Message decoder</td>
<td>8 weeks (for a decoder equal to that of MTsim), less for a more rudimentary decoder such as the one in this master thesis project.</td>
</tr>
<tr>
<td>Communication libraries</td>
<td>8 weeks</td>
</tr>
<tr>
<td>Either by adapting existing ones or by creating new ones.</td>
<td></td>
</tr>
<tr>
<td>Multithreading by implementation of the suggested executor service, see Section 5.7</td>
<td>6-8 weeks</td>
</tr>
<tr>
<td>Program handling and scheduling of test programs</td>
<td>5 weeks</td>
</tr>
<tr>
<td>Timer handling</td>
<td>3 weeks</td>
</tr>
<tr>
<td>Event handler</td>
<td>5 weeks</td>
</tr>
<tr>
<td>Support for all SMILE data types</td>
<td>4 weeks</td>
</tr>
<tr>
<td>Error handling and logging</td>
<td>3 weeks</td>
</tr>
<tr>
<td>Random numbers</td>
<td>2 weeks</td>
</tr>
<tr>
<td>Task</td>
<td>Duration</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Tagging of send messages</td>
<td>4 weeks</td>
</tr>
<tr>
<td>SMILE to Java framework translator</td>
<td>10 weeks</td>
</tr>
<tr>
<td>Integration of framework</td>
<td>4 weeks</td>
</tr>
</tbody>
</table>

My rudimentary estimation would suggest that this is a project of at least 70 man weeks. This estimation is just to get an understanding of how large a project this would be.

### 7.5. Feasibility

The concept of automatically translate SMILE programs to Java is very much feasible. It will however incur some problems as we have discussed. Also the sheer amount of code that is to be translated means that even if a great framework was created and a very efficient translator was created the actual translation would be very difficult.

One of the main problems of the idea of an automatic translation of SMILE to Java is that the outcome, the translated code, will not be very readable. Unfortunately machine translated code tend to be quite bad even though it might be correct and functioning. The reason is that the computer is systematic, correct but lacks the intelligence and flexibility that a human developer has.
8. Review

8.1. Problem statement and requirements
Let us go back and review the problems and requirements that were stated in the introduction. Let us also discuss if the thesis has provided solutions to these problems and requirements.

8.1.1. Problem statements
Is it possible to automatically convert the application MTsim from SMILE into Java?
The simple answer is yes, it is possible. However as has been discussed earlier it can be a greater task than initially thought. See section 7.5 for further discussion on this.

How much user interaction would be needed for such a conversion?
The only real need for user interaction, or for manual translation, would be translation of structures which are not supported by the framework or by the translator. One such structure which was decided not to be supported by the framework is the SMILE GOTO control structure. Each time the original SMILE code has a GOTO structure the translation will fail and manual interaction will be needed. This is however very rare.

As of right now quite a lot of manual interaction would be required since there are very many SMILE structures that are not supported by either the framework or the translator. Most of these however should be perfectly translatable in a future design thus the amount of user interaction required can be regarded as very little. However it is possible to choose a design where more elements of the original code are translated with user-interaction to make the translation more accurate. One of these are translation of source code comments where we could let the user decide where to copy the comments to the new code.

How much time would such a translation require?
The actual translation of the source code is quite fast it seems to correspond linearly to the size of the SMILE source code to be translated. There have not been any formal measurements done during the project. One reason is that the actual translation would, in an ideal case, only be done once, after the framework and translator has been implemented. The actual time for the translation process is not very important.

What are the risks involved?
The main risk is introducing new logical bugs with regards to the unsigned/signed integer dilemma (see section 4.4.1 and section 7.2.1 for more information). Another problem is that the translation of source code comments (section 7.2.4) could fail, due to comments in the SMILE code not being placed where they are expected to be. This could mean that
comments of one particular piece of code in the SMILE program is actually set as comment to an entirely different piece of code in the Java program. If this is the case it would create great confusion for any developer trying to maintain and extend the program.

**How can any arbitrary program written in SMILE be converted to a Java framework?**

It would be extremely difficult to automatically translate any arbitrary SMILE program to Java. There are constructs and data types in SMILE that do not at all match to Java. It would require very much intelligence of the translator to translate things like this. One example of a construct that does not translate well into Java is the Goto construct found in SMILE.

**Will the code of the new Java framework be maintainable?**

This is proven to be one of the most problematic aspects of the translation. The translation process itself can be made to be quite secure and fast. However, even with a well designed framework the actual quality of the code decreases.

There are two main reasons for this. The first reason is that the translation is made by a computer. A computer will not generate code which will be as easily understood to humans as code written by humans. There is little flexibility in an automatic conversion of language and there is little room for intelligence during the actual conversion. If a human would do the translation manually the result would be worth a lot more.

The other reason why the code is not as good after the translation is that the SMILE language was actually written to accommodate FSM applications of this kind. Although a good design of the FSM framework used in this project ensures that the framework itself is extendible, maintainable and flexible, the actual test programs will not be as clean and understandable after the translations compared to the original SMILE code.

### 8.1.2. Requirements

The performance of the suggested framework shall not be worse than that of the current solution.

Measurements of performance has not been conducted and has been suggested as work for the future, this requirement cannot be verified.

Translating from SMILE to the Java framework should not require extensive human interaction.

This requirement can be considered to be partly satisfied since we have shown that the prototype framework and translator can translate several test programs without any
Review

interaction needed. However the framework prototyped in this thesis project does not implement all functionality which will be required for a full translation of SMILE. This could lead to much more human interaction needed during translation.

Also problems such as the integer data type problem (sections 4.4.1 and 7.2.1) could be avoided with human interaction.

The translation from SMILE to Java should not introduce new logical bugs.

As of right now this requirement cannot be considered fulfilled. The reason is that with the current design there is no guarantee that no new logical bugs are introduced. This is mainly due to the choice of translation of the integer data types.

See sections 4.4.1 and 7.2.1 for discussions on the integer data type problem.

8.2. Conclusion

The overall conclusion of this master thesis work is that it is feasible to do a complete and automatic translation of SMILE code to a Java framework. However there are some obstacles which need to be addressed. The integer problem, translation of comments, and delaying of execution are some of them. I am convinced that there are good ways of handling those problems; however, further studies will be needed to find suitable solutions.

The implementation costs are another factor to consider. It would probably be quite an extensive implementation required, especially if the output Java code should be maintainable and readable which should always be the aim when creating code.
Appendix A. Design patterns

The concept of Design patterns is widely used in software engineering. It originates from the architect and author Christopher Alexander who introduced most of the terminology used for design patterns today [4]. In his book *A Pattern Language* he presents a number of patterns which are structured methods for solving common and recurring problems in structural architecture. The book is today regarded as the birth of modern design patterns.

The concept has been adapted into several fields of science and engineering and has been made very popular in Object-oriented programming by the so called Gang of Four (GoF) who compiled the most common and accepted software design patterns at the time into the bestselling book *Design Patterns: Elements of Reusable Object-Oriented Software*.

State pattern

The state pattern, sometimes referred to as the object of states pattern, is categorized as a behavioral pattern [4]. Behavioral patterns are used for solving algorithmic problems and to divide responsibilities between objects. They not only describe the objects of a design but also the interaction between them.

The state pattern lets objects change their behavior during run time, making it appear that the object changed its class. It is this property of the pattern for which it is mostly used, letting an object have different behavior depending on which state it is currently in.

![Figure 17 Structure of the State pattern as described by GoF](image-url)
Design patterns

**Visitor pattern**
The Visitor pattern is a way of separating an algorithm from the data on which it operates. In essence it separates the problem into two class hierarchies. The first is called the node hierarchy which is the nodes on which operations need to be made. The second hierarchy is the visitors which perform the actual operations. The nodes do not know what operations will be made on them, only that they need to accept visitors. This is useful when there is a structure of data on which different operations need to be done. By keeping the algorithms for these operations separate from the data we have a higher degree of reusability.

![Figure 18 Basic structure of the visitor design pattern](image)

**Command pattern**
The command pattern encapsulates the concept of an action or a command, along with its parameters, into an object. The command pattern, just like the state pattern, is classified as a behavioral pattern.

The idea behind the pattern is for different objects to be able to invoke actions or commands without having to know which exact object is the recipient. The interaction between two objects usually means one of the objects (issuer) having a reference to the
other object (recipient) and then invoking functions or methods on the recipient. However it is not always desirable to hold references to the recipient.

With the command pattern the issuer holds a reference only to the command object which in turn handles the communication with the recipient.

![Diagram of Command pattern](image)

**Figure 19 Structure of the Command pattern as described by GoF**

**Singleton pattern**

The singleton design pattern is used to restrict instantiation of a class to one single instance, the singleton. This is useful when only one instance of a class is needed and multiple instances could be harmful or inefficient. The singleton has a private instance, a singleton instance of itself. The class has a private constructor, thus allowing no external element to create new instances of the class. Instead, to get an instance of the class, external modules need to call a static method which returns the internal instance of the singleton.

![Singleton class diagram](image)

**Figure 20 Basic structure of the Singleton design pattern**

Singletons should be preferred over using application wide static methods. The main reason is that the solution with static methods is not polymorphic. It does in fact go against object-oriented principles altogether. Usage of the singleton pattern is by some considered bad practice and some do not consider it to be acceptable object-oriented
Design patterns

programming the reason being that it has a private constructor, which cannot be extended. If the constructor was made protected instead of private the singleton class could then be extended, this would however also mean that it would be possible to create multiple instances and thus breaking the singleton contract. Another consideration is that we cannot create an interface that forces classes to be singletons since the constructor is private and not part of the interface. We cannot, by interfacing techniques, force classes to implement private methods.

The singleton pattern has some drawbacks but the advantages are often so great that singletons should be used.

Flyweight pattern

One of the problems of object-oriented programming is that the number of objects sometimes becomes extremely high. Using an excessive number of instantiated objects will affect performance and memory usage.

The Flyweight pattern is intended to partly help decrease the number of object instances by allowing sharing of objects.

Each flyweight is divided into two parts, the extrinsic and the intrinsic part. The extrinsic part is state-dependant and cannot be shared among clients. The intrinsic part is not dependent on the state of the flyweight and can thus be shared.

Every client that needs a specific object implemented with the Flyweight pattern makes a request for an object. At this time the system only need to allocate new memory to the extrinsic part of the flyweight, since this is the only part not shared among clients, instead of having to allocate memory for the entire object.

The intrinsic part of the flyweight is managed and handled inside the flyweight whereas the extrinsic data must be handled by the client and passed to the flyweight when invoked.

The Flyweight pattern increases performance and memory usage for systems which have problems of object overhead, but has the drawback of bringing complexity to the software designs. A class diagram for the Flyweight pattern as described in [4] is shown below.
Figure 21 Structure of the Flyweight pattern as described by GoF
Appendix B. Java

The Java programming language is developed at Sun Microsystems. It originates from the Oak Language from 1990 and has grown to be one of the most widely used programming languages in the world. A search to compare the number of registered software projects on SourceForge\(^3\) gives the list presented in Table 17. Over 170000 projects registered at SourceForge clearly indicate the popularity of Java, at least in the open source community. This statistic is supported by Table 18 which presents the top 10 rated software languages by TIOBE software as of 2008-02-01. TIOBE bases the ratings on the number of relevant hits for a particular programming language from the largest search engines.

The birth of Java

In 1990 Sun Microsystems was competing with Microsoft in the workstation market, which at the time was relatively small. Sun Microsystems was left behind as Microsoft was beginning to dominate the market with products for the Intel-based workstations. At the time Sun Microsystems chief researcher Bill Joy retreated to Aspen Colorado for advanced research and founded Aspen Sun Smallworks. His team of programmers was planning on producing software for personal digital assistants (PDAs) and cellular phone platforms and started using C++. They realized, however, that it was too complex and with the aim of using a simpler programming language created a new language from scratch which they initially called “C++ minus minus”. Aspen Sun Smallworks, deciding that the PDA market was not yet big enough, changed focus to interactive TV. At the time, the programming language of choice for interactive TV set-top-boxes was a language called Oak, a small, safe and platform independent language which inspired them for what eventually became Java. [9]

Java became the talk of the industry and even before the official release of Java it had been licensed to industry giants like Microsoft, Intel and IBM.

\(^3\) The largest website (http://www.sourceforge.net) for open source software developing projects, it hosts over 170000 projects in various programming languages and various areas of software.
### Table 17  Number of registered SourceForge projects for some popular programming languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Number of registered projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>29118</td>
</tr>
<tr>
<td>C++</td>
<td>24401</td>
</tr>
<tr>
<td>C</td>
<td>20703</td>
</tr>
<tr>
<td>Python</td>
<td>7938</td>
</tr>
<tr>
<td>C#</td>
<td>7178</td>
</tr>
<tr>
<td>Delphi/Kylix</td>
<td>2493</td>
</tr>
<tr>
<td>Visual Basic</td>
<td>2422</td>
</tr>
<tr>
<td>TCL</td>
<td>1045</td>
</tr>
<tr>
<td>Pascal</td>
<td>508</td>
</tr>
<tr>
<td>Fortran</td>
<td>283</td>
</tr>
</tbody>
</table>

### Table 18  Top 10 rated programming languages from TIOBE

<table>
<thead>
<tr>
<th>Programming Language</th>
<th>Ratings 2008-02-01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>21.48%</td>
</tr>
<tr>
<td>C</td>
<td>14.86%</td>
</tr>
<tr>
<td>(Visual) Basic</td>
<td>11.60%</td>
</tr>
<tr>
<td>PHP</td>
<td>9.89%</td>
</tr>
<tr>
<td>C++</td>
<td>9.27%</td>
</tr>
<tr>
<td>Perl</td>
<td>6.21%</td>
</tr>
<tr>
<td>Python</td>
<td>4.76%</td>
</tr>
<tr>
<td>C#</td>
<td>4.51%</td>
</tr>
<tr>
<td>Delphi</td>
<td>2.80%</td>
</tr>
<tr>
<td>JavaScript</td>
<td>2.33%</td>
</tr>
</tbody>
</table>
Java

**Java virtual machine**

Java source code is not compiled into machine native instructions as is the case with C/C++. Instead Java is compiled into a Java bytecode which shares many similarities with ordinary microprocessor machine code. The Java Virtual Machine is an application that works as a runtime environment, an interpreter, for the Java bytecode applications. In essence it simulates a processor. It performs all the normal operations that a normal hardware processor would do; it handles stacks and instruction execution, memory management and so on. It does this in accordance with a strict specification. The specification of the virtual machine is open which means that anyone could create a Java compliant virtual machine.

The Java Virtual Machine offers some very important benefits. First of all it lets user programs execute in a safe environment, a virtual environment, making Java a very safe language to use. Another major advantage is that the Java Virtual machine is relatively small and light weight which means that it is easily ported to different platforms and can even be implemented as part of other applications such as for example web browsers.

Running programs on a virtual machine is obviously slower than running programs on the native machine, the real hardware. The speed of the JVMs is however getting faster and faster with every new released version.
Appendix C.  Glossary

The following acronyms and abbreviations are used throughout the thesis. Acronyms and abbreviations are also defined the first time they are used in the report.

**2G**  Second generation wireless telephone technology networks

**2Gsim**  Ericsson AB product for simulation of second generation wireless telephone networks.

**ASCII**  American Standard Code for Information Interchange — ASCII has a set of 128 characters each represented by 7 bits, 1 bit is used for parity check. ASCII is the U.S. version of the International Alphabet No 5 (IA5).

**AST**  Abstract Syntax Tree

**BCD**  Binary Coded Decimal — Numbering system using base 2 where every digit is represented with four binary digits.

**BSC**  Base Station Controller

**FIFO**  First In, First Out principle. — Also known as FCFS, First Come, First served. A way of prioritizing data or processes in a queue. Data objects or processes which have been in queue longer have priority.

**FPA**  Function Point Analysis — Measurement of functional size of software application.

**FSM**  Finite State Machine

**GC**  Garbage Collection

**GNU**  GNU’s not Unix

**GoF**  Gang of Four — Refers to the authors of the popular book Design Patterns: Elements of Reusable Object-Oriented Software; Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

**GPL**  GNU General Public License

**GSM**  Global System for Mobile communications

**IA5**  International Alphabet No 5 — Alphabet specified by ITU-T recommendation T.50

**IETF**  Internet Engineering Task Force

**ITU-T**  International Telecommunication Union- Standardization— United Nations agency for information and communications technologies.

**J2SE**  Java 2 Standard Edition

**Java NIO**  Java New Input/Output — New IO library for Java added in J2SE 1.4
<table>
<thead>
<tr>
<th><strong>Glossary</strong></th>
<th><strong>Definition</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>JVM</strong></td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td><strong>LGPL</strong></td>
<td>GNU Lesser General Public License</td>
</tr>
<tr>
<td><strong>MTsim</strong></td>
<td>Mobile Traffic Simulator</td>
</tr>
<tr>
<td><strong>RFC</strong></td>
<td>Request For Comments — Documents describing internet standards by IETF.</td>
</tr>
<tr>
<td><strong>SLOC</strong></td>
<td>Source Lines of Code — The number of source code lines in a file.</td>
</tr>
<tr>
<td><strong>SMILE</strong></td>
<td>Proprietary Ericsson programming language used in TSS.</td>
</tr>
<tr>
<td><strong>SUT</strong></td>
<td>System Under Test — System that is tested for correct operation.</td>
</tr>
<tr>
<td><strong>TCP</strong></td>
<td>Transmission Control Protocol — Connection based transport layer network protocol defined in RFC 793.</td>
</tr>
<tr>
<td><strong>TPC</strong></td>
<td>Test Program Compiler</td>
</tr>
<tr>
<td><strong>TSS</strong></td>
<td>Test and Simulation Solutions system or Traffic Simulator System</td>
</tr>
<tr>
<td><strong>UML</strong></td>
<td>Unified Modeling Language — An Open Management Group (OMG) standard used to model software.</td>
</tr>
<tr>
<td><strong>UniMod</strong></td>
<td>Unified Modeling — SourceForge project for executable state chart diagrams</td>
</tr>
</tbody>
</table>
References


[7] jfsm. URL: https://jfsm.dev.java.net/ (2008-03-06)


[15] OMG, OMG Unified Modeling Language (OMG UML),Superstructure, V2.1.2


Glossary

Index

2
2Gsim..............................................................1

A
A Pattern Language......................................82
Abstract Syntax Tree......................................23
action code..................................................14
Alexander, Christopher.................................82
Array literals ..................................................17

B
Base Station Controller..................................1
BCD literal
   SMILE......................................................16
Behavioral patterns......................................82
Black-box framework..................................43
BSC...........................................see Base Station Controller

C
Christoph Kessler.........................................3
Command pattern.........................................83
Common variables .......................................19

D
data types
   SMILE......................................................17
   translation..............................................32
Declare section.........................................15
Department of Computer and Information Science....3

E
Ericsson AB..................................................1
extrinsic data...............................................12

F
filler..........................................................16
Finite State Machine
   definition...............................................6
   Implementation
      Object oriented ...................................9
      Procedural.........................................8
      transition..........................................6
Flyweight pattern......................................85
FSM.................................................. See Finite State Machine

G
Gang of Four...............................................82

garbage collection.......................................51
Global include file......................................19
Global variables........................................19

H
Hexadecimal string literals..........................16

I
IDAsee Department of Computer and Information Science
identifiers
   SMILE......................................................17
   translation..............................................31
Instance variables......................................19
Integer literal
   Java......................................................28
   SMILE......................................................16
intrinsic data............................................12
inversion of control....................................45

J
Java bytecode..............................................89
Java Virtual Machine..................................51, 89

L
Linköping University....................................3
Local variables..........................................20
Log settings..............................................13

M
match pattern.............................................48
Mobile Traffic Simulator................................2, 1
modular arithmetic.....................................34
MTsimsee Mobile Traffic Simulator, see Mobile Traffic Simulator

N
Niklas Lanzén..............................................3

O
Object of States........................................82
   See State pattern
operators
   SMILE......................................................18
   translation.............................................36

P
Parameter..................................................13, 19
S
SMILE.............................................................. 2, 1, 13
State pattern ....................................................... 82
String literal
   SMILE................................................................ 16, 29
symbol table ....................................................... 23
T
Test case........................................................... 13
Test input file .................................................... 13, 19
Test Program Compiler ...................................... 23
   Analyze ........................................................... 24
   Declare ........................................................... 23
   Generate ........................................................ 24
   Lister .............................................................. 24

Main........................................................................ 23
Parser ............................................................... 23
Scanner ............................................................. 23
Support.............................................................. 24
TP_REFERENCE.................................................. 21
TPC........................................... See Test Program Compiler
Transition table.................................................. 7
TSS Language .................................................... 13, see SMILE

U
Unified Modeling ............................................... 53
UniMod........................... See Unified Modeling
unsigned dilemma.................................................. 32

W
White-box framework........................................ 43
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Said Aspen