

Rheinisch Westfälische Technische Hochschule Aachen
Chair for Software Modeling and Verification



Translation of Sequential Function Charts to Transition Systems

Schaper, Hauke

Matriculation number: 265223

1st Supervisor: Prof. Dr. Katoen

2nd Supervisor: Prof. Dr. Epple

Tutor: Sabrina von Styp

Aachen, July 17, 2011

Declaration

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Aachen, July 17, 2011

Contents

1	Introduction	3
2	Preliminaries	5
2.1	Labeled Transition Systems	5
2.2	Programmable Logic Controller	6
2.3	Sequential Function Charts	6
2.3.1	Basics of Sequential Function Charts	6
2.3.2	File structure of the SFC XML-File	8
2.4	Function Block Diagram	11
2.4.1	Possible Block Types of Function Block Diagrams	12
3	Translating an SFC to an LTS	15
3.1	Translating Steps	15
3.2	Translating Action Blocks	15
3.3	Translating Transitions	16
3.4	Characteristics of the Translation	17
3.5	Translating the Example SFC	18
4	Developing a program for automated translation from SFC to LTS	21
4.1	Specifying Language, Input and Output format	21
4.2	Planing the architecture	22
4.3	Extracting and Saving the Data	24
4.3.1	Variables	24
4.3.2	Transitions	26
4.3.3	Body Elements	26
4.4	Building the LTS	28
4.4.1	Understanding body objects and the ConstructLTS class	28
4.4.2	Limitations of computer based Testing and extending the ConstructLTS class	28
4.4.3	Generated LTS	30
5	Conclusion	33
6	Bibliography	35

List of Tables

4.1	Example LTS and corresponding Aldebaran file	22
4.2	Variables extracted from the example SFC through the program	25
4.3	Transitions extracted from the example SFC through the program	26
4.4	Actions extracted from the example SFC through the program	26
4.5	Body Elements extracted from the example SFC through the program	27

List of Figures

2.1	Sample LTS	5
2.2	Sample SFC Program	7
2.3	Basic transition types	7
2.4	FBD Example 1	10
2.5	FBD Example	12
3.1	Translating Action-Block to LTS	16
3.2	Translating Transition to LTS with comparison of two variables	16
3.3	Translating Transition to LTS with comparison of a variable to a value	16
3.4	Translating - Looping	17
3.5	Translating the example SFC - Step 1	18
3.6	Translating the example SFC - Step 2	18
3.7	Translating the example SFC - Step 3	19
3.8	Translating the example SFC - Step 4	19
3.9	Translating the example SFC - Step 5	20
3.10	Translating the example SFC - Final Step - Combining	20
4.1	Class Diagram	24
4.2	JTorx Adaptation - Changing the order of guard and action	29
4.3	Generated LTS	31

Listings

2.1	XML-Structure of SFC files	8
2.2	XML-Structure of SFC files - variable declaration	9
2.3	XML-Structure of SFC files - action declaration	10
2.4	XML-Structure of SFC files - transition declaration	10
2.5	XML-Structure of SFC files - Body step declaration	11
2.6	XML-Structure of SFC files - Body actionBlock declaration	11
2.7	XML-Structure of SFC files - Body transition declaration	11
4.1	Implementation of reading the variables from XML-File	25
4.2	Generated *.aut file	30

Abstract

Programmable Logic Controller (PLC) are used in industry to control all kind of industrial units, therefore it is essential that they function correctly. PLC consist of a hardware and a software part. In this thesis the focus is on the software side of PLC.

Different programming languages are standardized for the programming of PLC. One of them are Sequential Function Charts (SFC). SFC are graphic based notations for PLC programs which offer a wide set of opportunities for programming, like parallelism and activity manipulation.

In this thesis an approach for automated translation from Sequential Function Charts to Labeled Transition Systems (LTS) is introduced. This translation enables automated testing of the generated LTS with JTorx to verify the correctness of the source SFC.

1 Introduction

This thesis presents the translation of sequential function charts to labeled transition systems.

Sequential function charts are used in industry to program logic controllers for the machinery. They are a graph based language and therefore easy to construct even without deeper knowledge of programming. The goal of this thesis is to create a labeled transition system from the sequential function chart which is then tested, to check the input output behaviour of the underlying sequential function chart, using JTorx [8].

The preliminaries will cover the basic knowledge about labeled transition systems, which are widely used in computer science, programmable logic controllers used in industry to control the machinery, sequential function charts for programming the programmable logic controllers, the structure of the underlying XML-File of the sequential function chart, specified in the IEC 61131-3 norm [3] and finally a brief introduction to function block diagrams used in sequential function charts. It also introduces an example sequential function chart which accompanies the whole thesis as an example for the single steps of the process.

Chapter 4 "Translating an SFC to an LTS" will introduce definitions on how to translate specific elements from sequential function charts to transition systems and which characteristics have to be considered during the process. After the definitions the example from chapter three will be translated to a labeled transition system.

The next step then is to develop a program which translates the sequential function chart to a labeled transition system using the definitions introduced earlier. The program is developed using the example sequential function chart introduced in the preliminaries. Input and output formats are defined and at the end the program is adapted to fit the needs of JTorx.

The last chapter covers the conclusion of this thesis.

2 Preliminaries

In the following chapter Labeled Transition Systems, Programmable Logic Controller, Sequential Function Charts and Function Block Diagrams are introduced.

2.1 Labeled Transition Systems

Definition: A labeled transition system LTS is a tuple (S, S_0, T, L) where

- S is a set of states,
- s_0 is the initial states,
- L is a labeling function and
- $T \subseteq S \times L \times S$ is a transition relation.

A Transition system work as follows, it has some initial state s_0 and can be evolved by using the transition relation T . A transition relation $(s_0, \alpha, s_1), s_1 \in S$ states, that there is a transition with the label α from state s_0 to s_1 . States with no outgoing transitions are called final states. Final states can be as well states with outgoing connections but they are marked with a doubled surrounding. Figure 2.1 shows a basic labeled transition system with three states.

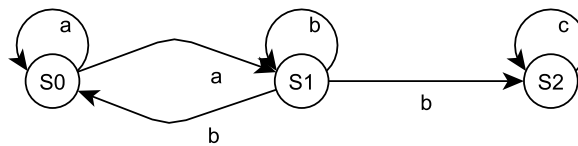


Figure 2.1: Sample LTS

The figure describes the following tuple (S, S_0, T, L) where $S = \{S_0, S_1, S_2\}$, $S_0 = S_0$, $T = \{(S_0, a, S_0), (S_0, a, S_1), (S_1, b, S_0), (S_1, b, S_1), (S_1, b, S_2), (S_2, c, S_2)\}$. Starting from S_0 the transition system can be evolved by choosing one of the two transition relations.

A labeled transition system accepts a language if every word from this language can be recreated by evolving the LTS. This example accepts the language given by the regular expression $a^+ [(b^+ a^*) | (b^+ c^*)]^+$. A word from this language is $aabbcc$ which can be evaluated in the LTS by using $(S_0, a, S_0), (S_0, a, S_1), (S_1, b, S_1), (S_1, b, S_2), (S_2, c, S_2), (S_2, c, S_2)$.

Further information about labeled transition systems and the ones presented here can be found in [1] and [4].

2.2 Programmable Logic Controller

Programmable Logic Controller (PLC) are used to control or regulate machinery in the industry. They have replaced the old hardware regulated controllers to add more flexibility in the daily work-flow making the reprogramming of a machinery more uncomplicated and therefore cheaper by just exchanging the program on the controller instead of reconstructing the controller. A PLC normally has inputs, outputs and an operating system on which the programs can be loaded. The PLC now is connected to a machinery, more precisely sensors and other measuring / controlling parts of a machine and maps the inputs to the measuring and reacts depending on the underlying program by modifying the output variables which are mapped to the controlling of the machine. In this way a PLC can control a machine depending of its current state. PLCs can be programmed in one of the five languages, Instruction List (IL), Ladder Diagram (LD), Function Block Diagram (FBD), Sequential Function Charts (SFC) and Structured Text (ST), defined in the IEC 61131-3 [3].

2.3 Sequential Function Charts

Sequential Function Charts (SFC) are one of the main programming languages for Programmable Logic Controllers. SFC are a graph based notation to define the structure and the elements of a program for PLCs. They resemble to basic transition systems which consist of transitions and steps. Furthermore they are extended with action blocks which can be specified as an SFC or as a Function Block Diagram (FBD).

2.3.1 Basics of Sequential Function Charts

In general SFCs consist of a starting step, transitions and action blocks linked to various steps. Figure 2.2 shows how a possible simple SFC may look like. Regarding this example SFC the whole translation process will be illustrated in the following Chapters.

Steps are states in a sequential function chart which can be entered after the corresponding guard to the incoming transition is fulfilled. After entering a step checking the outgoing transitions determine whether the step is repeated or if a transition can be taken to the next step. If no guard of an outgoing transition is enabled the step will be looped, causing the step to be performed at least once before an outgoing transition is taken.

As shown in Figure 2.2 this SFC, and in general every SFC, has exactly one initial step which is called *Start* here. Furthermore the example has a set S of steps, with $S = \{\text{DriveOff}, \text{ToOn}, \text{DriveOn}, \text{ToOff}, \text{Error}\}$.

Action Blocks are added to steps, like *OffAction* to the *DriveOff* step. Each action block has a qualifier specifying how the action is handled. They are divided into:

- N - always active when the corresponding step is active
- S - this action will be active until it is reset

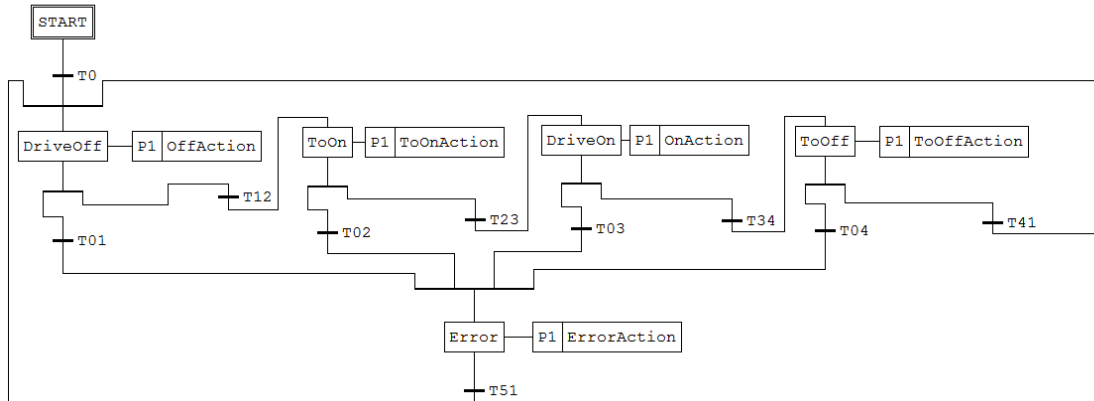


Figure 2.2: Sample SFC Program

- R - stops an action which was started with the S qualifier
- P1 - action will be performed once when the block is entered
- P0 - the action will be performed once when the block is exited

The action block itself can be an SFC or an FBD containing the action, meaning that a single action block can increase the complexity of a whole SFC if it is based on an even larger SFC.

In this example the set of action blocks **AB** is defined as

AB = {OffAction, ToOnAction, OnAction, ToOffAction, ErrorAction}.

Transitions are named with a corresponding *guard*, specifying the conditions under which the transition can be taken. If the conditions of the guard are fulfilled the transition will be seen as enabled, otherwise it is disabled. The guard is given as an FBD in which for example variables can be checked for equality.

There are five kinds of transitions in SFCs depicted in Figure 2.3.

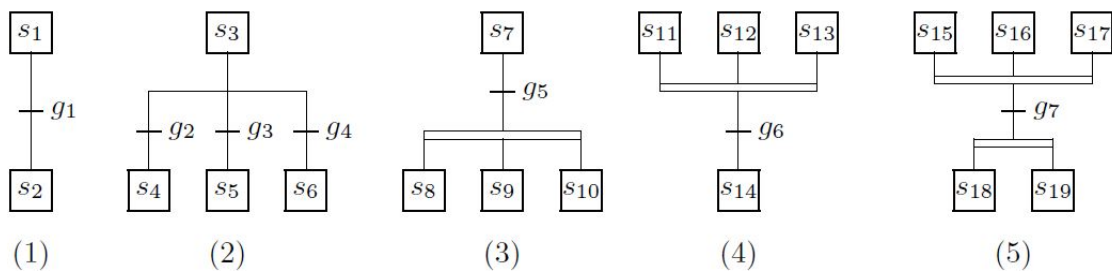


Figure 2.3: Basic transition types (Source: [2] page 3)

1. Simple transitions leading from one step to another
2. Alternative branching splitting between several transitions from which only one guard can be taken at a time, but more then one transition can be enabled
3. Divergence branching splits the transition and leads to more steps which are then executed in parallel
4. Convergence branching joins several incoming transitions
5. a combination of 4 and 5

Having more than one outgoing transition at one state can lead to conflicts determining which transition will be taken if more than one transition is enabled. For that the transitions have a priority which can be specified in the SFC. Is no priority order given the transitions will be taken in the order they appear in the XML-File having a first come first served priority.

The set T of transitions in Figure 2.3 is $\mathbf{T} = \{T0, T01, T12, T02, T23, T03, T34, T04, T41, T51\}$ which therefore defines the set of transition relations $\mathbf{T}_R = \{(Start, T0, DriveOff), (DriveOff, T12, ToOn), (DriveOff, T01, Error), (ToOn, T02, Error), (ToOn, T23, DriveOn), (DriveOn, T03, Error), (DriveOn, T34, ToOff), (ToOff, T04, Error), (ToOff, T41, DriveOff), (Error, T51, DriveOff)\}$. The guards of the transitions are not visible in this picture but each transition guard is well defined in the corresponding XML-file.

Guards are divided into two categories, one that checks expressions, marked with a question mark in front of the expression and the one setting variables to given values, identified by an exclamation mark in the front. For example, the guard $?A=1,B=0$ is enabled when A is equal to one and B equal to zero and on the other hand $!B=1_A=0$ sets the variable A to the value zero and the variable B to the value one.

2.3.2 File structure of the SFC XML-File

The underlying structure of the example SFC given in Figure 2.2 is a simple XML-file and the file format is standardized in the IEC 61131-3 [3]. In this thesis the file format provided by Beremiz ([7]) will be discussed and analysed with special focus on the example sequential function chart.

The following listing presents the overall structure of such an SFC-XML-file.

Listing 2.1: XML-Structure of SFC files

```

1 <Project>
2 <fileHeader [...]/>
3 <contentHeader [...]>
4   [...]
5 </contentHeader>
6 <types>
7   <dataTypes/>
8   <pous>
9     <pou name="" pouType="" globalID="">
10       <interface>
11         <returnType/>
12         <localVars name="">
13           <variable name="" adress="" globalID="">
14             <type>
15             </type>
16             <initialValue>

```

```

17                                     </initialValue>
18                                     </variable>
19                                 </localVars>
20                             </tempVars/>
21                             <inputVars/>
22                             <outputVars/>
23                             <inOutVars/>
24                             <externalVars/>
25                             <globalVars/>
26                             <accessVars/>
27                         </interface>
28                         <actions>
29                             <action name="">
30                                 <body>
31                                     <FBD>FBD Elements</FBD>
32                                     <SFC>SFC Elements</SFC>
33                                 </body>
34                             </action>
35                         </actions>
36                         <transitions>
37                             <transition name="">
38                                 <body>
39                                     <FBD>FBD Elements</FBD>
40                                     <SFC>SFC Elements</SFC>
41                                 </body>
42                             </transition>
43                         </transitions>
44                         <body Worksheetname="" globalID="">
45                             <SFC>Use Elements Below</SFC>
46                         </body>
47                     </pou>
48                 </pous>
49 </types>
50 <instances>
51     <configurations/>
52 </instances>
53 </Project>

```

The file starts with some meta information provided in line one to five in listing 2.1. First there are some opening tags until the `<pous>` tag in line eight is reached. Every sequential function chart is specified in one `<pou>`, so one project can contain multiple sequential function charts. The `<pou>` is structured into four main parts: **interface**, **actions**, **transitions** and **body**.

As the name interface may suggest, the first part from line 10 to 27 in listing 2.1 is covering the storage of global variables used in this project. In our example (see Figure 2.2) it contains the declaration of ACT, DriveOn, DriveOFF, ERR, C_On, C_Off, C_ACK and chkb_on as boolean input variables. See the next listing for an example how the declaration of the DriveOff variable is stated in the example XML-File.

Listing 2.2: XML-Structure of SFC files - variable declaration

```

<variable name=" DriveOff">
    <type>
        <BOOL/>
    </type>
    <initialValue>
        <simpleValue value="FALSE" />
    </initialValue>
</variable>

```

In the second part called actions in listing 2.1, from line 28 to 35, the actions are stored. Each action is contained in an `<action>` tag with an attribute called name storing the name of the. As an example the listing 2.3 shows an action with a `<body>` being specified as an FBD in which the action is described.

Listing 2.3: XML-Structure of SFC files - action declaration

```

<action name=" ToOffAction">
  <body>
    <FBD>
      <outVariable localId="1">
        <expression>ACT</expression>
      </outVariable>
      <inVariable localId="2">
        <expression>1</expression>
      </inVariable>
      <outVariable localId="3">
        <expression>DriveOff</expression>
      </outVariable>
      <outVariable localId="4">
        <expression>DriveOn</expression>
      </outVariable>
      <outVariable localId="5">
        <expression>ERR</expression>
      </outVariable>
    </FBD>
  </body>
</action>

```

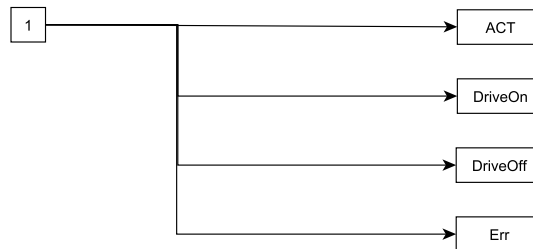


Figure 2.4: FBD Example 1

Visualizing the listing 2.3 could result in the graphic 2.4.

There is one input variable specified as 1 and four output variables. The only possible interpretation can be, that the four output variables are set to the value of the input variable, setting them to 1 in this case.

Transitions, referred to as part three of listing 2.1 (line 37 to 42) is about transitions in the sequential function chart and each transition is expressed in a `<transition>` with defining sub nodes. It has an attribute that stores the name and a body in which the guard is specified either as an FBD or SFC. The example uses only FBDs for the guards. In the FBD input and output variables are declared, as well as a block with a type name expressing which action is used. Here it is an “EQ” block which means that the input variables are compared to equality and the output will be true if they are equal. The transition can only be used if the two input variables `chkb_on` and `1` are equal which is the case if `chkb_on` is true. The internal block variables are referring to the global block variables, so `IN1` and `IN2` are the both input variables and `OUT` is `T01`.

Listing 2.4: XML-Structure of SFC files - transition declaration

```

<transition name="T01">
  <body>
    <FBD>
      <outVariable localId="1">
        <expression>T01</expression>
      </outVariable>
      <inVariable localId="2">
        <expression>chkb.On</expression>
      </inVariable>
      <block localId="3" typeName="EQ">
        <inputVariables>

```

```

                <variable formalParameter="IN1">
                </variable>
                <variable formalParameter="IN2">
                </variable>
            </inputVariables>
        </inOutVariables/>
        <outputVariables>
            <variable formalParameter="OUT">
            </variable>
        </outputVariables>
    </block>
    <inVariable localId="4">
    <expression>l</expression>
    </inVariable>
</FBD>
</body>
</transition>

```

The last part covers the <body> of the SFC where all steps, transitions and action blocks are defined and linked to the previously specified elements. In other words, the body states how the SFC is structured and what is connected with what.

Steps - Every step has a name and a local ID and the initial state is flagged with an attribute *initialStep* which is set to true.

Listing 2.5: XML-Structure of SFC files - Body step declaration

```

<step localId="1" name="START" initialStep="true">
</step>

```

Action Blocks - Each action block contains an action with a qualifier attribute (link to qualifier declaration) and a node specifying the reference to the former introduced action.

Listing 2.6: XML-Structure of SFC files - Body actionBlock declaration

```

<actionBlock localId="7">
    <action localId="0" qualifier="P1">
        <reference name="ToOffAction" />
    </action>
</actionBlock>

```

Transitions have a condition note in which the reference to the transition from earlier is given, allowing to find and interpret the corresponding guard.

Listing 2.7: XML-Structure of SFC files - Body transition declaration

```

<transition localId="10">
    <condition>
        <reference name="T01" />
    </condition>
</transition>

```

2.4 Function Block Diagram

Function Block Diagrams (FBD) are one of the three programming languages specified in the IEC 61131-3 [3]. It is a graphical programming language for PLCs reminding of logical function plans in logic programming. A function block diagram consists of three components: function, in- and output variables and lines connecting them. Using the graphical notation, it is uncomplicated to understand the programs and build them with different functions in an ordered structure. An example shown in figure 2.5, raises value A by one and then compares the value to B.

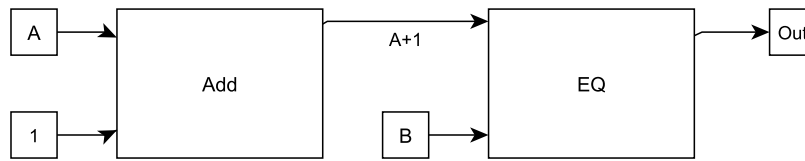


Figure 2.5: FBD Example

2.4.1 Possible Block Types of Function Block Diagrams

In all examples in this thesis the block type equality is used to explain certain operations. FBDs offer a lot more block types which can be used when programming. A short extract with useful types is given in the next listings.

Numerical Functions

- SQRT - Square root
- LOG - Logarithm
- EXP - Exponentiation
- COS - Cosine

Arithmetical Functions

- ADD - Addition
- MUL - Multiplication
- SUB - Subtraction
- DIV - Division
- MOD - Modulo
- EXP - Exponent

Bitwise Functions

- AND - output is the logical add of the two input bits.
- OR - output is the logical or of the two input bits.
- XOR - output is the logical xor of the two input bits.
- NOT - output is the logical not of the input bit.

Selection Functions

- SEL - the binary selection has three inputs, one bool and two variables of any kind. Depending of the state of the boolean one of the variables is set as output.
- MAX - returns the maximum of the two input variables.
- MIN - returns the minimum of the two input variables.
- LIMIT - has three inputs, a maximum value, a minimum value and one input value. If the value is in between the two limits the value is returned, if it is greater then the maximum, the maximum is returned and if it is smaller then the minimum, the minimum is returned.

Comparison Functions

- GT - greater than
- GE - greater or equal
- EQ - equal
- LT - less than
- LE - less or equal
- NE - not equal

The next chapter will now cover the essential parts of translating SFCs to LTS using the basics learned in this chapter.

3 Translating an SFC to an LTS

Building up on the previous chapter, elements can be translated from sequential function charts to labeled transition systems. The three main elements, steps, action blocks and transitions from sequential function charts are very similar to the existing elements in a labeled transition system.

3.1 Translating Steps

In general steps in an SFC resemble states in an LTS, defining a position in the graph. Furthermore states in an LTS represent a state of the given system as steps in SFCs can be linked to an action block adding more functionality to it. Assuming the focus only on the step and not on the linked action block translating a step to a state is a simple one to one relation. For example the start step can directly be translated to a state in an LTS. That can be applied for every step of an SFC because steps are states with no internal functionality in an SFC except for linking to an action block, in- and outgoing transitions. This means in terms of translating an SFC to an LTS, reaching a step just means creating a state in the corresponding LTS.

3.2 Translating Action Blocks

Action Blocks in comparison to steps are much more complicated to translate. First of all the kind of the action block has to be determined, it can be a sequential function chart (SFC) or an Function Block Diagram (FBD). Due to the complexity of sequential function charts used as action blocks, this thesis presents only FBD action blocks.

Comparing that to LTS the action block resembles a transition with a label setting variables to a specific value, considered as an output. For example it can set specific variables to a fixed value or raising a variable by one each time the step is entered, letting the variable act as a counter. This means that the action block is translated to a transition and another state in the LTS. Taken the example 2.2 it means that the action block will be represented by a transition with the labeling from the action block action, for example *!DriveOn-1_DriveOff-1_ACT-1_ERR-1*, leading to a new state. Figure 3.1 illustrates this translation.

This translation is only applicable if the action qualifier (see chapter 2.3.1) is P1, P0 or N. If the qualifier is S then the action has to be added to every following transition until the action block is reset.

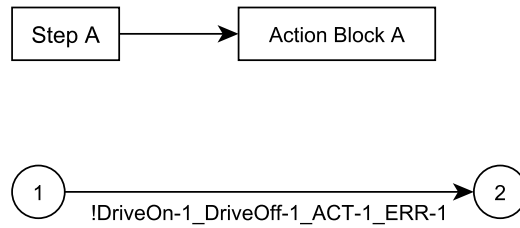


Figure 3.1: Translating Action-Block to LTS

3.3 Translating Transitions

Transitions in an SFC are linked to guards which can be used to check whether a transition is enabled or disabled. Referring to that check the transition including the guard can be seen as a transition with an input check to specific values given in the guard. So contrary to the action block performing an output the transition checks the validity of an input as explained in the guard classification in 2.3.1. Knowing this fact an SFC transition can be translated to one or more LTS transition using the FBD from the guard as an input check. If a transition guard is comparing to values of type bool for example, we create two transitions, one for each possible equality of the variables (3.2). Referring to the example given in listing 2.4 a translation will be given in figure 3.3. More types and there possible translations were discussed in 2.4.1.

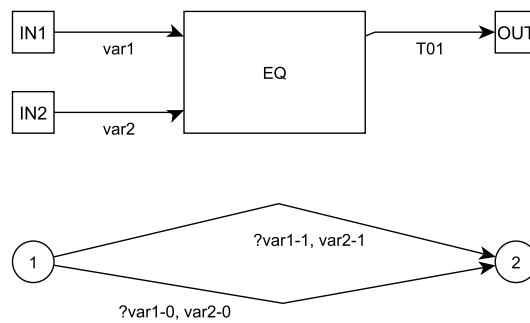


Figure 3.2: Translating Transition to LTS with comparison of two variables

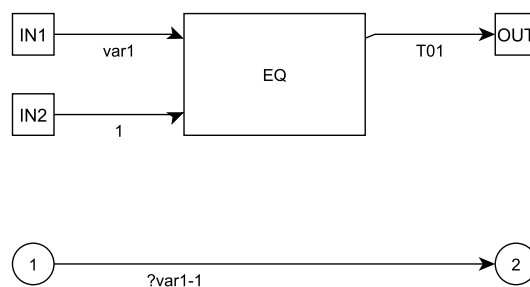


Figure 3.3: Translating Transition to LTS with comparison of a variable to a value

A special case is the branching of transitions, meaning the splitting of one outgoing transition to n outgoing transitions, depicted in figure 2.3 picture 2. In this case the ordering of transitions is from left to right and each branch holds its own guard.

In addition to the equal function block there are several more blocks mentioned in 2.4.1 and the following paragraphs give an overview of how to translate them.

Numerical functions can be translated one to one by adding the function to the variable, for example the square root can be replaced by $\text{input} * \text{input}$ in the transition system.

Aithmetical functions have two input variables and the output is the result of the operation, so you can just replace the block by the function itself. The **comparison functions** however are a bit more complex, there the type of the input variables has to be considered. Taken the greater than function, if used with two boolean as input it can be translated to a guard $G1$ with the labeling Input1-1_Input2-0 if the input mapping is input one greater than input two. If the input is not boolean all combinations from the range of the two input types have to be covered, making it impossible to translate it for other types than boolean or finite intervals. If a finite interval is given, the result is the combination of all greater than relations in this interval leading to a state explosion if the interval gets bigger.

Another example is the greater or equal relation, using it for boolean results in three guards, $G1 = I1-0_I2-0$, $G2 = I1-1_I2-1$ and $G3 = I1-1_I2-0$. Restrictions given for other input than boolean is exactly the same as for the greater than relation.

3.4 Characteristics of the Translation

When no outgoing transition of a step is enabled it means that SFC will stay in this action-step until one of the outgoing transitions is enabled. Making this possible the LTS has to be extended by a loop from the state in front of the transitions to the step-state negating the conditions of the transition like shown in figure 3.4.

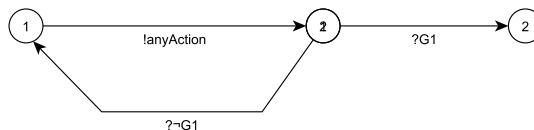


Figure 3.4: Translating - Looping

If the action qualifier is $P0$ then the action is only performed when the step is exited, so the self loop starts and ends in state 2, assuring that the action is only performed once.

3.5 Translating the Example SFC

Figure 2.2 can now be translated using the definitions 3.1 up to 3.4. Starting from the start step, following the transitions and applying the corresponding rules. The translation are separated into parts - every part handles exactly one step one \rightarrow transition \rightarrow step two sequence, where step two is the beginning step of the next sequence.

Definition: Sequence $S_x = (S_x, T_x, S_{x+1})$, $S_{x+1} = (S_{x+1}, T_{x+1}, S_{x+2})$.

- **Part 1:**

The first part consists of the sequence (Start, T0, DriveOff). Applying definition 3.1 the "Start"-step gets converted into a state called S11, defining the first (x1) generated state (S) from step one (1x). After the step is translated the action block needs to be converted following description 3.2. Since there is no action block in this special step, an empty transition is generated leading into step S12. From this S12 the transition with the guard "T0" leads to the next step. Considering description 3.3 the transition from S12 to S13 is being produced. Finally the loop described in 3.4 needs to be applied, creating a new transition from step S12 to step S11.

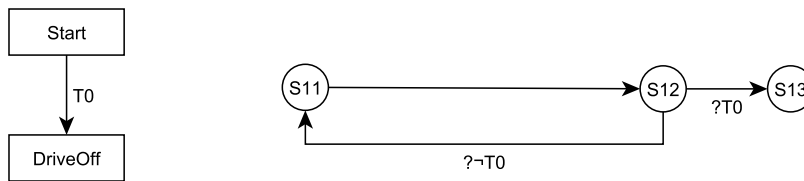


Figure 3.5: Translating the example SFC - Step 1

- **Part 2:**

Step "DriveOff" has two outgoing connections, relying on the definition the leftmost is taken first. This step generates the following two sequences: $S_1 = (\text{DriveOff}, T01, \text{Error})$ and $S_2 = (\text{DriveOff}, T12, \text{ToOn})$. Since the action is the same because the start step is exactly the same, the two action transitions and two new steps generated from these sequences got merged into one.

The translation generates the following LTS-fragment depicted in figure 3.6.

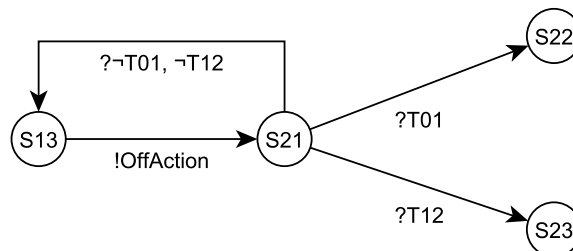


Figure 3.6: Translating the example SFC - Step 2

- **Part 3:**

Sequence $S_4 = (\text{Error}, T51, \text{DriveOff})$ is translated in the same way as part 1. Fig-

Figure 3.7 shows the result.

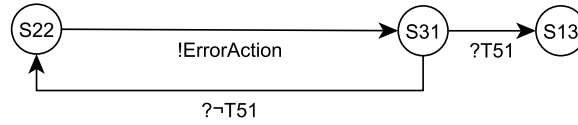


Figure 3.7: Translating the example SFC - Step 3

A first complete cycle is finished and the created fragments from part 1 to 3 can be combined creating the LTS in figure 3.8.

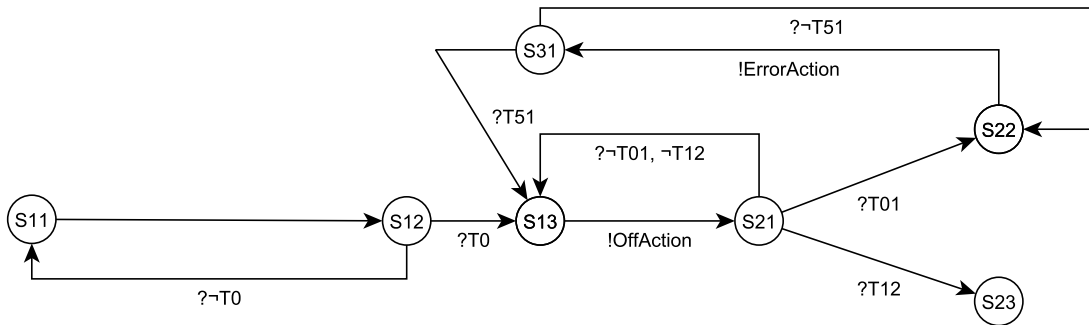


Figure 3.8: Translating the example SFC - Step 4

• **Part 4 to Part 6:**

The following sequences can be derived from the SFC

$S = \{(ToOn, T02, Error), (ToOn, T23, DriveOn), (DriveOn, T03, Error), (DriveOn, T34, ToOff), (ToOff, T04, Error), (ToOff, T41, DriveOff)\}$. Figure 3.9 shows the translation for the part 4 to 6 and figure 3.10 depicts the whole generated LTS.

In conclusion this chapter showed the elemental rules for translating a sequential function chart to a labeled transition system and demonstrated it on the example sequential function chart step by step. The next chapter will evaluate a program, using this rules, to automatically translate sequential function charts.

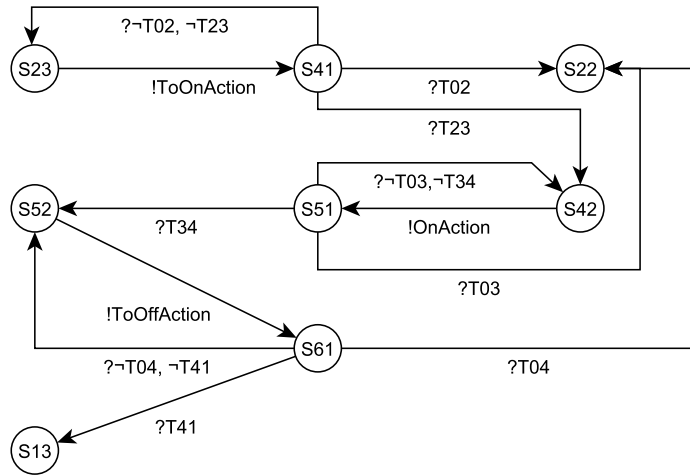


Figure 3.9: Translating the example SFC - Step 5

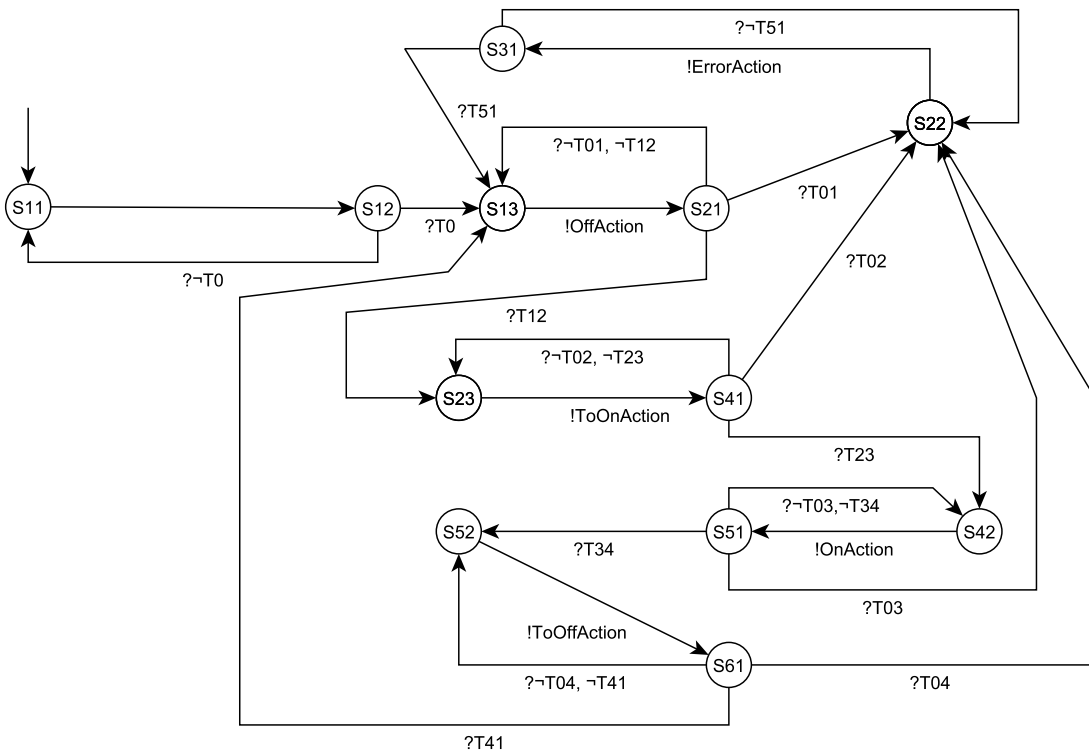


Figure 3.10: Translating the example SFC - Final Step - Combining

4 Developing a program for automated translation from SFC to LTS

After introducing the definitions of what has to be translated and how it theoretically works, this chapter now focuses on the implementation of the translation. Furthermore the limitations of JTorx [8] and the resulting adoptions for the program are discussed.

4.1 Specifying Language, Input and Output format

As a **language** for the program **Java** [9] is the first choice due to the portability to other systems and the easy integration into other programs.

For generating the **input** sequential function chart for the translation, Beremiz [7] is used, which saves the created sequential function chart as an **XML-File**, see chapter 2.3.2. Beremiz offers an uncomplicated user interface and a drag and drop editor for creating diagrams and charts. Therefore the input of the program will be the XML-File created by Beremiz.

The output needs to be usable by JTorx, which supports a variety of different file formats as model input. JTorx is a tool for model-based-testing implemented in Java, for more information see [8].

The two best fitting input formats are Aldeberan (*.aut) and Graphml (*.graphml), wherefore Graphml is missing the simplicity of Aldeberan, therefore Aldeberan is the output file format. Aut files are very intuitive and easy structured.

An .aut file starts with a defining line for the start point, number of transitions and the number of states in this LTS. The states are labeled with numbers and the initial state which should always be numbered with 0.

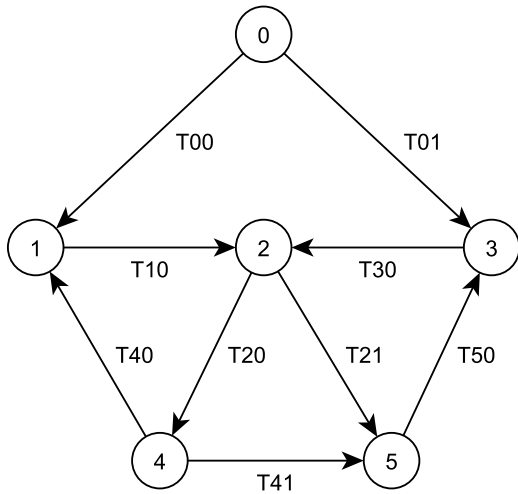
First line: des (<first-state>, <number-of-transitions>, <number-of-states>)

Each following line represents an edge leading from an arbitrary state a to an arbitrary state b using transition t.

Edge Line: (<state-a>, <transition-t>, <state-b>)

States must be numbers in the Aldebaran file format, whereas transition can be anything up to 5000 characters enclosed between quotation marks. If no special characters and no white-spaces are used, the quotation marks can be excluded. The labeling of the states can be a disadvantage, because it is hard to connect two states over a name, because it is not possible to give states descriptive names. How this problem is handled in case of the translation is explained later on.

The simplicity of Aldebaran files can be directly seen in table 4.1.



Example Aldebaran file:

```

des (0,9,6)
(0,"T00",1)
(0,"T01",2)
(1,"T10",2)
(2,"T20",4)
(2,"T21",5)
(3,"T30",2)
(4,"T40",1)
(4,"T41",5)
(5,"T50",3)
    
```

Table 4.1: Example LTS and corresponding Aldebaran file

4.2 Planing the architecture

Every programming process is started by creating a structure and hierarchy for classes in a program. First of all a class diagram will be created by discussing the needed functionality and model objects for the translation.

Analysing the task the following separation of actions can be extracted:

- reading and extracting the XML-File,
- saving the data extracted into model classes,
- construct the LTS in Aldebaran-format,
- save it to a file.

From that listing three controller classes can be extracted, the class which handles the extraction and storage of the XML-File, a class that constructs the LTS and one class to write the output file. To make the program more structured an administration class can be added, which handles the data objects and is therefore responsible for storing and reading the model objects.

As model classes variables, actions and transitions can be identified. Having a deeper look at the XML-File structure, implementing another model class for the objects specified in the body part is useful.

Model-Classes with needed attributes:

- **Class:** Actions
Attributes: Name, Value
 The value is the value which will be the label of the transition in the LTS.
- **Class:** Variables
Attributes: Name, Type, Value, Categories
 Type = Datatype, Category = Input, Output, ...

- **Class:** Transitions
Attributes: Name, BlockType, InVarHashMap, BlockInputList
BlockType is the type of the block in the FBD. InVarHashMap is responsible for mapping the internal captions for the variable and the variable name, wherefore BlockInputList saves the captions of the incoming block variables.
- **Class:** BodyObject
Attributes: ID, Name, Type, Qualifier, RefName, references, visited

Controller-Classes with needed functions:

- **Class:** XMLFunctions
Methods: fillLists, searchChildNodes, exploreChildNodes, getNodePosition, fillVar[Action, Transition, Body]List, getAttribute
fillLists will be called by the main function, this triggers the fill[x]List functions which will then explore the XML file and search for their elements and add them via the administration to the lists managed by the administration. getAttribute is a sub-function which returns the value of an attribute from a node.
- **Class:** Administration
Methods: add elements to the managed lists, return specific elements from lists
The administration is for managing the representatives of the model classes, therefore it needs to be able to add new objects to the lists stored by itself.
- **Class:** ConstructLTS
Methods: buildLTS
BuildLTS is a method which will convert the stored data into an LTS.

Figure 4.1 shows a simplified class diagram of the needed classes, their dependencies and the most important methods and attributes.

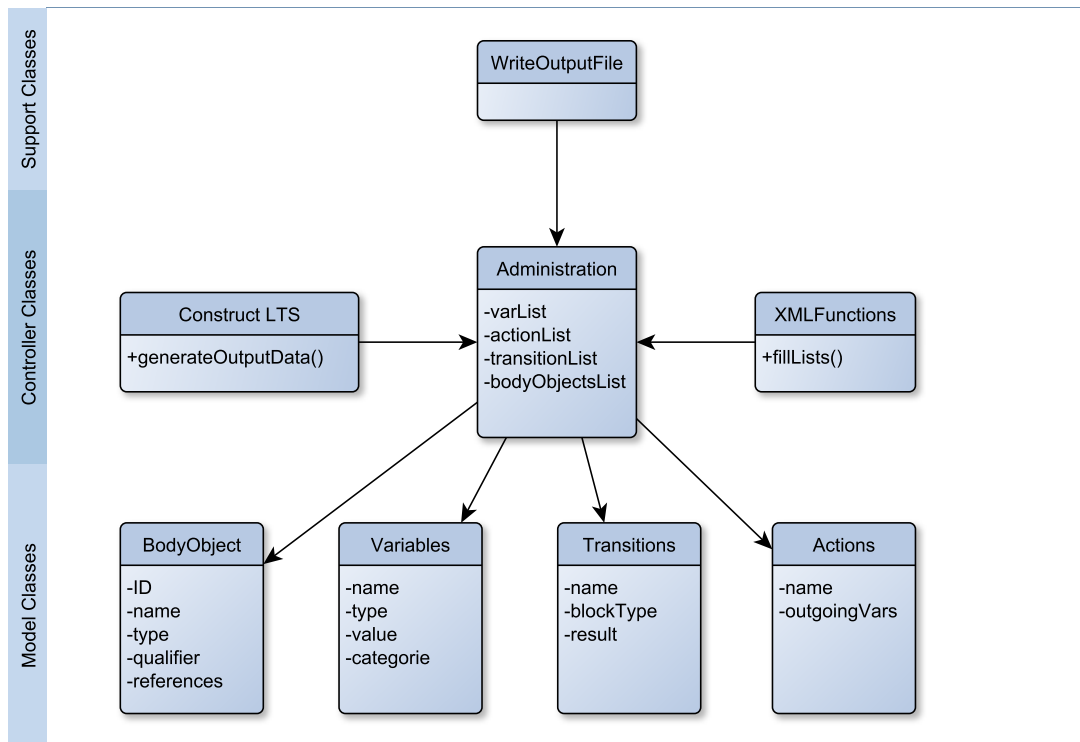


Figure 4.1: Simplified Class Diagram

4.3 Extracting and Saving the Data

4.3.1 Variables

The XML-File starts with the definition of the variables used in the sequential function chart, so at first the variables will be extracted. For that purpose the XPath class ¹ offers some needed functionalities for navigating through the XML-File and will therefore be used for the main navigation.

If the path to the variables is been constructed, the following path will be received: `"/project/types/pous/pou/interface"`. So the actual path is being set via the XPath class object. The result of evaluating the XPath object with the given part is a list of nodes which is searched for "variable" sub-nodes. Considering the tree structure of an XML-File each element called node can be divided into sub elements, the children of the node when looking at the tree structure and this nodes are named sub-nodes. A list of nodes from the sub-nodes will be extracted by searching for nodes with the name "variable" (see listing 2.2). In this given list of nodes are only nodes which define variables, so the name can be extracted from the attribute name. The categories (input, output, inOut etc.) of the

¹import javax.xml.xpath.*

variable are retrieved by analysing the parent node. Its name specifies the categories of the variables included in it.

The data type of the node is found when looking at the first child element of the sub node type. So a search for the sub node type is being performed and then the first child is been taken and the name of that child resembles the data type of the variable.

Every variable has an initial value which is set as a value attribute to the node simplevalue, which is also a sub node of the node variable. So extracting the initial value of a variable is finding the sub node and reading the value of the attribute simplevalue.

Finally the variables have to be stored in some kind of data format in the Java program. For that a simple class "Variable" class is being implemented containing only getters for the four string attributes name, type, value and category, The addVar method from the administration class is being called to store the variable in the variable list contained in the administration object.

Name	Type	Initial Value	Category
ACT	BOOL	FALSE	outputVars
DriveOn	BOOL	FALSE	outputVars
DriveOff	BOOL	FALSE	outputVars
ERR	BOOL	FALSE	outputVars
C_On	BOOL	FALSE	inputVars
C_Off	BOOL	TRUE	inputVars
C_ACK	BOOL	FALSE	inputVars
chkb_On	BOOL	FALSE	inputVars

Table 4.2: Variables extracted from the example SFC through the program

```

//Fills the varlist from the xml source
private void fillVarList(Document document) throws XPathExpressionException{
    //Naviagte to the interface part of the cml file where the variables are stored
    XPath xpath = XPathFactory.newInstance().newXPath();
    String exp = "/project/types/pous/pou/interface";

    //extract a NodeList from the given path
    NodeList varListParent = (NodeList) xpath.evaluate(exp, document, XPathConstants.NODESET);

    //filter for nodes of the categorie variables (in case that there are other nodes which we
    dont want)
    List<Node> matchNodes = searchChildNodes("variable", varListParent);

    for (int i = 0; i < matchNodes.size(); i++){ //iterate through all Variablenodes
        Node workingNode = matchNodes.get(i); //set working node for less writing
        String nodeName = getAttr(workingNode, "name"); //extract the name of the var, which is
        save in the name attribute
        String nodeCategorie = workingNode.getParentNode().getNodeName(); //set the variable
        categorie, depending on the parent node

        //search for the type of the node, which is stored in a node --> getNodeName
        List<Node> catNode = searchChildNodes("type", workingNode.getChildNodes());
        String nodeType = catNode.get(0).getChildNodes().item(1).getNodeName();

        //search for the initial value of the node, which is specified in an attr in the
        simplevalue node
        List<Node> valueNode = searchChildNodes("simpleValue", workingNode.getChildNodes());
        String nodeValue = getAttr(valueNode.get(0), "value");

        //finally add the variable to the varlist in the admin class
        administration.addVar(nodeName, nodeType, nodeValue, nodeCategorie);
    }
}

```

Listing 4.1: Implementation of reading the variables from XML-File

4.3.2 Transitions

Transitions are extracted using the same class as used for the actions. The path to the nodes is in this case `"/project/types/pous/pou/transitions"`. This node list is then searched for child nodes with the name transition and this child nodes are saved in another node list. This node list now contains only the needed informations and can be received directly. The name of the transition is stored in an attribute named "name".

A transition now consists of an FBD which has multiple input variables and a block type. The input variables are searched by searching in all child nodes for elements which are called "inVariable" and from that list the information about the variables is read. The block contains informations about which variables are mapped to which internal inputs. At the end of the method the newly extracted object is stored via the administration class.

Name	Text
T01	?chkb_On-1
T02	?chkb_On-0
T03	?chkb_On-0
T04	?chkb_On-1
T12	?C_On-1
T23	?chkb_On-1
T34	?C_Off-1
T41	?chkb_On-0
T51	?C_ACK-1
T0	?true

Table 4.3: Transitions extracted from the example SFC through the program

Actions are read and stored in the same manner as transitions and variables are, table 4.4 lists the actions from the example in an extracted format.

Name	Text
ToOffAction	ACT-1 DriveOff-1 DriveOn-1 ERR-1
ToOnAction	ACT-1 DriveOff-0 DriveOn-0 ERR-0
OnAction	ACT-1 DriveOff-1 DriveOn-0 ERR-0
OffAction	ACT-0 DriveOff-0 DriveOn-1 ERR-0
ErrorAction	ACT-0 DriveOff-0 DriveOn-0 ERR-1

Table 4.4: Actions extracted from the example SFC through the program

4.3.3 Body Elements

Body elements are the elements which are contained in the body part of the XML-File and therefore define the actual SFC giving references to the former specified objects. A

body object is either a step, action block or one of the transitions defined in 2.3. The extracting algorithm works as follows, first the path to the nodes is set to `"/project/type-s/pous/pou/body"` and the list of all nodes is stored. Searching for each type of object on this list results in new lists which can then be easily disassembled and stored as individual body objects which different types. Only steps have a name, but they do not have qualifier nor reference names. Qualifier are unique for action blocks handling when and how often an action from an action block is performed, see 2.3.1 for more informations. The next section describes how Ids and references are connected, enabling the built of an LTS.

ID	Name	Type	References	Qualifier	Refname
1	START	step		null	null
2	ToOn	step	16	null	null
3	DriveOn	step	17	null	null
4	DriveOff	step	22	null	null
5	ToOff	step	18	null	null
11	Error	step	23	null	null
7	null	actionBlock	5	P1	ToOffAction
6	null	actionBlock	2	P1	ToOnAction
8	null	actionBlock	3	P1	OnAction
9	null	actionBlock	4	P1	OffAction
12	null	actionBlock	11	P1	ErrorAction
10	null	transition	24	null	T01
13	null	transition	25	null	T02
14	null	transition	26	null	T03
15	null	transition	27	null	T04
16	null	transition	24	null	T12
17	null	transition	25	null	T23
18	null	transition	26	null	T34
19	null	transition	27	null	T41
20	null	transition	11	null	T51
21	null	transition	1	null	T0
22	null	selectionConvergence	20 21 19	null	null
23	null	selectionConvergence	10 13 14 15	null	null
24	null	selectionDivergence	4	null	null
25	null	selectionDivergence	2	null	null
26	null	selectionDivergence	3	null	null
27	null	selectionDivergence	5	null	null

Table 4.5: Body Elements extracted from the example SFC through the program

4.4 Building the LTS

Using the extracted data from the last section and the rules described in the last chapter, an algorithm can be constructed which generates the transition system and saves it into a file. The most essential part of the reconstruction is the list of body objects and how they are linked to each other, making it possible to construct the labeled transition system.

4.4.1 Understanding body objects and the ConstructLTS class

The body objects are the main core of the disassembled sequential function chart, therefore it is essential to understand how the different objects are connected. Considering table 4.5 there is only one single object with no reference, which is the start step. No object has a reference to the start step because it is the first object and no step goes back to the initial step again, making it the start object for the algorithm (compare 2.2).

The next step is now to find the successor of the start object. The successor has a reference to its predecessor, so searching in the reference column for the ID of the start object will result in the successors. In this example there is only one successor, the transition with the ID 21 and the referenced name "T0".

Iterating over the the Ids and the references the first part of the LTS can be constructed following the definitions 3.1 to 3.3.

In our Example the trace with the Ids 1, 21, 22, 4, 9, 24, 10... can be rebuilt. When reaching step Error over and over again, a program can easily run into an endless loop when it always tries to follow all outgoing connections.

The model therefore needs an attribute visited for every step, so if the program runs through a loop and comes back to a former visited state it has to stop, otherwise it will not terminate. If a step is left the visited attribute is set to true, so the step can not be the starting point for another trace.

After having built the LTS to this degree, loops have to be added to stay in a state if no outgoing transition is enabled and performing the concatenated action, as described in 3.4. The output labeled transition system is now exactly the same as the manually generated one depicted in figure 3.10.

4.4.2 Limitations of computer based Testing and extending the ConstructLTS class

In theory the generated LTS could now be given to JTorx to verify the correctness of the input and output behaviour of the source SFC, but when checking an LTS there are some points that differ between computer based testing and human testing.

Adaptation 1:

A human can tell which is the first transition to go if more than one transition is enabled, because it is the leftmost / topmost of all outgoing transitions, but how should JTorx handle this decision?

When no additional data is specified it is some sort of random in which order JTorx takes the transitions because it has no knowledge about leftmost / topmost from the Aldebaran file and it can not look into the XML-File from the source SFC to see which transition is defined first.

Therefore another way has to be thought of preventing JTorx from taking a wrong transition at any time. For example if there exist two outgoing transitions T1 and T2 and both are enabled and from the SFC can be seen that T1 should be taken first. This means if T1 and T2 are enabled, T1 is always taken. Only if T1 is disabled transition T2 will be taken. The transition T2 must be extended by negating T1, so that it can only be taken if not both are enabled and T1 is always taken if both are enabled. T2 becomes T2, $\neg T1$. The ConstructLTS class is extended by adding all previously defined outgoing transitions from a state to every following outgoing transition. For that we add every transition to a hash map with the negated transition guard as the value and the step id as key. Every time a transition guard is read the program looks into the hash map for other values with the same id as a key and adds them to the active guard. Now JTorx can determine which transition can be taken next.

Adaptation 2:

One peculiarity of the testing environment is, that the outgoing guards have to be in front of the actions after a state in the transition system, so the LTS has to be adapted and it needs to be proofed, that both LTS are equal. Figure 4.2 depicts the two versions of a state with two outgoing transitions.

The testing environment consists of a server running an emulation of the sequential function chart, an adapter which translates the messages between the server and JTorx. JTorx is the third part of the test environment. When the server processes one JTorx input it stops and requests the next set of actions. This means before a step is entered the guards are already set and send to JTorx to get the right action back, resulting in the adaptation for the transition system.

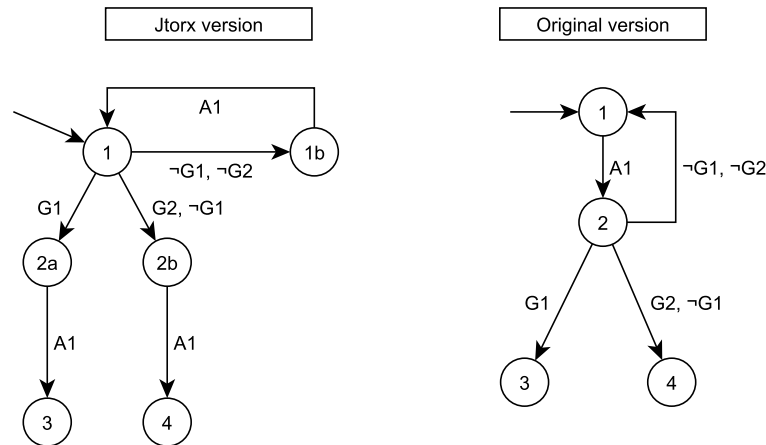


Figure 4.2: JTorx Adaptation - Changing the order of guard and action

In figure 4.2 state 1b is the state added for the loop. G1, G2 are the two guards for the transitions and A1 is the action performed after state 1, so the action connected to the step 1 in the underlying SFC. State 2 and 3 are the successors of state 1. States 2, 2a and 2b are program generated states.

Arriving in state 1 results in three scenarios which can occur:

1. G1 and G2 are not enabled \rightarrow that sets the transition with the guard $\neg(G1, G2)$

to enabled and the other two transitions as disabled. In the JTorx version the only enabled transition is taken to state 1b and from there the action transition is taken back to step 1, so the only thing that happens in this loop is the execution of the action A1. In the original version the action A1 is first performed and then the only enabled transition of the three outgoing transitions in step 2 is taken back to step 1, so the result from the JTorx and the original version are the same, in both versions the action A1 is executed once in the loop.

2. G1 is enabled and G2 either enabled or disabled \rightarrow in this case the action A1 has to be performed once and the guard G1 has to be checked and must be enabled before we reach state 3. In the JTorx version the checking of the guards happens first, because all outgoing are starting in state 1. Based on the assumption that G1 is enabled the transition to step 2a is taken and from there the transition concatenated with the action A1 is taken leading to state 3. In the original version the action is first performed leading to state 2 checking the two guards and G1 being enabled, state 3 is reached. Comparing the action sequence of both versions it is in both cases the one time execution of the action A1 and the final state is in both variants state 3. \Rightarrow They can be considered equal.
3. G2 is enabled and G1 is disabled \rightarrow same as 2.

After applying these two adaptations to the ConstructLTS class the final transition system can be generated.

4.4.3 Generated LTS

The program now generates an Aldebaran file shown in listing 4.2 which can be checked by JTorx. That ends the translation of an SFC to an LTS usable by JTorx to check the correctness of the source SFC. The final LTS is depicted in figure 4.3.

Listing 4.2: Generated *.aut file

```

1 des (1, 32, 22)
2 (1, "?true", 1000)
3 (1000, "!ACT-0_DriveOn-0_DriveOff-0_ERR-0", 4)
4 (4, "?chkb_On-1", 4000)
5 (4000, "!ACT-0_DriveOff-0_DriveOn-1_ERR-0", 11)
6 (11, "?C_ACK-1", 11000)
7 (11000, "!ACT-0_DriveOff-0_DriveOn-0_ERR-1", 4)
8 (4, "?C_On-1,chkb_On-0", 4001)
9 (4001, "!ACT-0_DriveOff-0_DriveOn-1_ERR-0", 2)
10 (2, "?chkb_On-0", 2000)
11 (2000, "!ACT-1_DriveOff-0_DriveOn-0_ERR-0", 11)
12 (2, "?chkb_On-1,chkb_On-1", 2001)
13 (2001, "!ACT-1_DriveOff-0_DriveOn-0_ERR-0", 3)
14 (3, "?chkb_On-0", 3000)
15 (3000, "!ACT-1_DriveOff-1_DriveOn-0_ERR-0", 11)
16 (3, "?C_Off-1,chkb_On-1", 3001)
17 (3001, "!ACT-1_DriveOff-1_DriveOn-0_ERR-0", 5)
18 (5, "?chkb_On-1", 5000)
19 (5000, "!ACT-1_DriveOff-1_DriveOn-1_ERR-1", 11)
20 (5, "?chkb_On-0,chkb_On-0", 5001)
21 (5001, "!ACT-1_DriveOff-1_DriveOn-1_ERR-1", 4)
22 (1, "?false", 1001)
23 (1001, "!ACT-0_DriveOn-0_DriveOff-0_ERR-0", 1)
24 (2, "?chkb_On-1,chkb_On-0", 2002)

```

```

25 (2002, "!ACT-1_DriveOff-0_DriveOn-0_ERR-0", 2)
26 (3, "?chkb_On-1,C_Off-0", 3002)
27 (3002, "!ACT-1_DriveOff-1_DriveOn-0_ERR-0", 3)
28 (4, "?chkb_On-0,C_On-0", 4002)
29 (4002, "!ACT-0_DriveOff-0_DriveOn-1_ERR-0", 4)
30 (5, "?chkb_On-0,chkb_On-1", 5002)
31 (5002, "!ACT-1_DriveOff-1_DriveOn-1_ERR-1", 5)
32 (11, "?C_ACK-0", 11001)
33 (11001, "!ACT-0_DriveOff-0_DriveOn-0_ERR-1", 11)

```

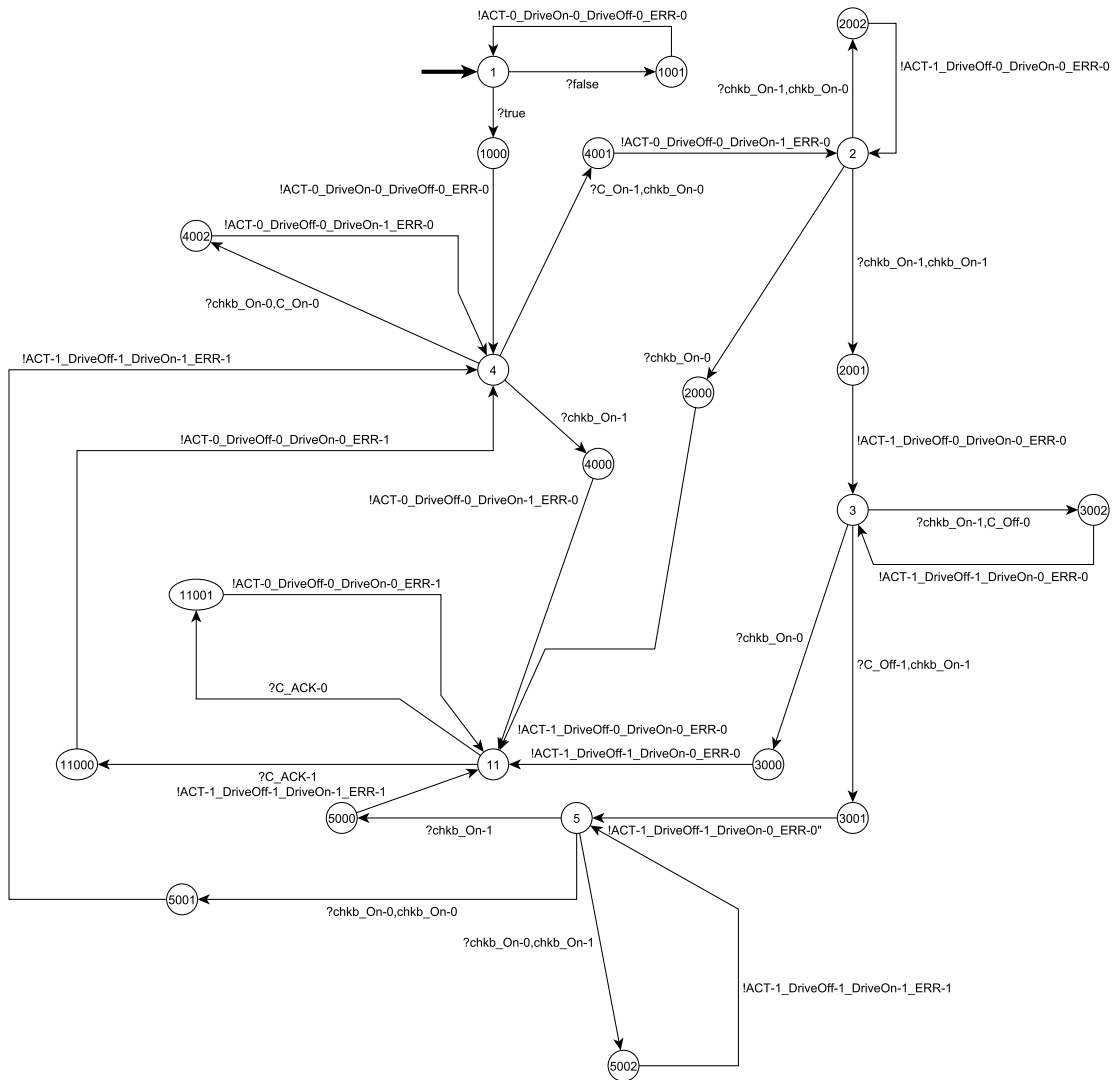


Figure 4.3: Generated LTS

This chapter covered one way of developing a program for automating the translation of sequential function charts to labeled transition systems. The program evaluated here is only a very limited prototype and includes a small amount of the rules shown in this thesis.

5 Conclusion

In this thesis the development of rules for translating sequential function charts to labeled transition system was introduced. That makes automatic testing based on JTorx for the input and output behaviour of the underlying sequential function chart possible. Furthermore a program was developed, using the knowledge gained in the former chapters, which can translate a given sequential function chart and output it into a labeled transition system in the Aldebaran file format.

Rules for translating sequential function charts were defined, covering most elements listed in the IEC 61131-3, divided into 4 major rules: Translating steps, action blocks, transitions and adding loops if no transition is enabled. This gave a base for applying the rules to a sequential function chart, demonstrating how they work and how the result looks like, creating an introduction for the development of an algorithm.

Based on the rules, a program was planned and developed to translate sequential function charts to labeled transition systems. The planning included the sketching of a class diagram and the creation of a model structure. With the given class diagram a NetBeans project was created, generating the model classes and implementing the functionalities of the controller classes. First the data was extracted from the underlying XML-File using predefined JAVA packages for XML handling and stored the data into the model objects. The next step was the implementation of the algorithm to build the transition system and adapt it to the constraints discussed in section 4.4.1. A recursive approach was taken, where "state - (transition / divergence / convergence) - state" chains were built and then translated, using the former defined rules, giving the end state as input for the function again. The labeled transition system is stored in Aldebaran format into a file, placed in the application folder. For time reasons only the most relevant functions for that thesis were included in the program, for example considering the block types, only the equality was included, but the program structure is extendible and most of the extension points are marked with comments, explaining how to add other functionalities.

In conclusion this thesis has shown a way to translate sequential function charts into labeled transition system stored in the Aldebaran file format and a basic program which can be extended to be able to translate all kinds of input sequential function charts into labeled transition systems.

Using labeled transition systems as output in the translation has to be enhanced, because labeled transition systems are missing a lot of functionalities, starting with basic ones, such as timed processes or guards with more complex assertions expressed in first order logic.

Another evolution of this thesis can be the extension of the program to support the whole set of available sequential function chart and function block diagram objects and integrate it into the testing environment. To extend the functionality and significance, the output can be changed into symbolic transition systems (STS) or timed automata in future research. Also the input of other input formats for programmable logic controllers, for example instruction lists or ladder diagrams, can be considered for future work.

6 Bibliography

- [1] Christel Baier, Joost-Pieter Katoen: *Principles of Model Checking*, The MIT Press
- [2] Nanette Bauer, Ralf Huuck, Ben Lukoschus, Sebastian Engell: *A Unifying Semantics for Sequential Function Charts*
- [3] PLCopen Technical Committee 6 *XML Formats for IEC 61131-3, Version 2.01*
- [4] Mark Timmer, Ed Brinksma, Marielle Stoelinga: *Model-based testing*
- [5] Lars Frantzen, Jan Tretmans, Tim A. C. Willemse: *A Symbolic Framework for Model-based Testing*
- [6] Nanette Bauer, Sebastian Engell, Ralf Huuck, Sven Lohmann, Ben Lukoschus, Manuel Remelh, Olaf Stursberg: *Verification of PLC Programs Given as Sequential Function Charts*

Software used in this Thesis:

- [7] Beremiz: <http://www.beremiz.org/>
- [8] JTorx: <http://fmt.cs.utwente.nl/redmine/projects/jtorx>
- [9] Java: <http://www.java.com>
- [10] Aldebaran: <http://www.inrialpes.fr/vasy/cadp/man/aldebaran.html#sect6>
- [11] Omondo - EclipseUML http://www.ejb3.org/download_studio_eclipse_3.5.html
- [12] NetBeans IDE 7.0 <http://netbeans.org/>
- [13] yEd Graph Editor http://www.yworks.com/en/products_yed_about.html