CHALMERS

Formal Verification applied to Sequential Function Charts

Qaisar Ahmad Malik

Control and Automation Laboratory
Department of Signals and Systems
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden

EX038/2004

Abstract

Sequential Function Chart (SFC) is a powerful graphical technique for describing the sequential behaviors of a Programmable Logic Controller's (PLC) program. The control systems driven by PLCs are often complex, safety critical and expensive. Any failure of these systems might not only result in financial loss but can lead to causalities as well. There is a need for verification of SFC programs running on such systems. But there is hardly any available verification tool for SFCs. The goal of this thesis is to provide such formal verification tool. In this thesis few of SFC language constructs are translated to be verified formally. For formal verification we will use a technique known as Symbolic Model Checking, a technique that provides an exhaustive analysis of properties on finite state models. A model checking tool Symbolic Model Verifier (SMV) has been used for this purpose. In SFC, dynamic semantic properties such as reachability of steps, mutual exclusion between shared variables, termination, avoidance of undesired states etc. are desirable to be verified. For this purpose SFC is translated into input language of SMV. This translation is performed automatically by software developed in this thesis. After translation desired temporal properties, expressed in temporal logic, of considered SFC program can be verified by SMV.

KEYWORDS: Sequential Function Charts, Symbolic Model Checking, Verification, Temporal Logic

Acknowledgments

I thank Knut Åkesson for supervising this thesis. His guidance, help and encouragement gave me the confidence to do this work. What I know about PLC programming especially Sequential Function Charts (SFC) is because of him. His deep insight about verification-needed areas within this domain helped me a lot in finding research direction. Since this is my first research project, what I know about process of research I learned from him.

I also thank Mary Sheran at department of Computer Science for teaching me courses on Formal Methods and introducing me to this wonderful world of Formal Methods.

The work in this thesis also rests rather heavily on that of Ralf Huuck at the University of Kiel Germany. His work in this direction served as a starting point. I thank him for his support and ideas he expressed during this project.

I must thank Kenneth L. McMillan at Cadence Berkeley Labs California USA for his help and guidance in finding solution to a problem during verification part of this thesis.

Thanks to Martin Fabian at department of Signals and Systems, Peter Gammie and Koen Claessen at department of Computer Science for their valuable support during this project.

Of course, I am indebted to all the good people at Signals and Systems who make this department and its facilities work.

CONTENTS

NOTATION

Abbreviations

CTL	Computational Tree Logic
IEC	International Electrotechnical Commission
LD	Ladder Diagrams
LTL	Linear Temporal Logic
PLC	Programable Control Logic
SFC	Sequential Function Chart
SFCVerifier	Sequential Function Chart Verifier
SMV	Symbolic Model Verifier by Cadence
XML	Extensible Markup Language

1 INTRODUCTION

In this introductory chapter we give an overview of this verification project and present relevant work already done in this direction. Moreover, concept and process of model checking is also discussed in later sections.

1.1 Defining the Problem

The purpose of this thesis is to formally verify Sequential Function Charts (SFC). SFC is one of the programming languages used for Programmable Logic Controllers (PLC), a kind of industrial controller used extensively in industry.

Control systems driven by PLCs are often complex, safety critical and involve a lot of money. Any failure of these systems might not only result in a significant financial loss but lead to causalities as well. Hence, their actual programming and correctness plays a vital role (Huuck 2003).

To check correctness there is need for verification of SFC programs. SFC programs often execute in a continuous loop and may exhibit interesting concepts like parallelism and inheritance. Since SFCs are often used in safety critical systems there should be verification for mutual exclusion, deadlocks and safety. There might be some other user defined properties which should not be violated by a SFC program during its execution; these properties depend upon the nature the of system modeled, like overflowing from an acid tank or setting temperature on maximum heat level etc. It is not easy task to manually verify each and every program for given property with their execution states. Here comes the idea of automated testing, the theme of this thesis, where a SFC program is analyzed in a software tool and verification is done for general and user defined properties.

1.2 Related Work

In past researchers had developed approaches for verification of SFCs depending upon their own requirements and analysis. These include:

Timed automaton.

In this approach (D.L'Her *et al.* 1995), Sequential Function Chart programs are represented as timed automata with consideration of continuous time and it abstract from explicit scan cycles. Finally these timed automatons are verified by using KRONOS, a model checker for real-time systems.

Execution model.

In this work (Hellgren *et al.* 2001), different execution models are discussed and solution to synchronization and mutual exclusion problems are presented. The technique to translate SFC to Ladder Diagrams (LD), another PLC programming language, is also discussed. Emphasis is on the knowledge of execution model also for the verification purpose.

Software verification for PLCs.

The most related work to this thesis is done in (Huuck 2003). Formal and Informal semantics for Sequential Function Charts' constructs are defined but they remain abstract. There are no detailed algorithms/techniques for translation of many of the SFC constructs into SMV code especially when step actions are involved.

1.3 Goal of this Thesis

The goal of this thesis is to investigate the possibility of formal verification of Sequential Function Chart programs. For this purpose a subset of SFC constructs are selected to be verified. JGrafchart (Årzén 2002, Johansson 1999), a programming tool for Sequential Function Chart, is used for programming and editing SFC. JGrafchart is implemented in Java 2. It runs on every computing platform that supports this environment. JGrafchart is selected for the various reasons: it is quite stable and user friendly tool, encouraging support was available from its developer during this project and importantly the facility of SFC programs to be transformed into XML files. It was quite feasible to parse XML files to analyze program structure and building our own data structure for SFCs. Once the Sequential Function Chart programs are read from JGrafchart and data structure is built, the SFCVerifier, the tool developed in this thesis, analyses the SFC program, which may contain more than one SFC, and applies different techniques to generate code for Symbolic Model Verifier (SMV). The techniques for translating SFC constructs into SMV are discussed in more detail in Chapter 3. The Symbolic Model Verifier (SMV) uses a technique known as Symbolic Model Checking [McMillan], described in detail in following section. For verification purpose we use Temporal Logic to verify our properties, also described in following section. The Figure 1.1 shows the process, proceeding from left to right.

1.4 Model Checking

Model checking is a technique for verifying finite state concurrent systems. One benefit of this restriction is that verification can be performed automatically. The

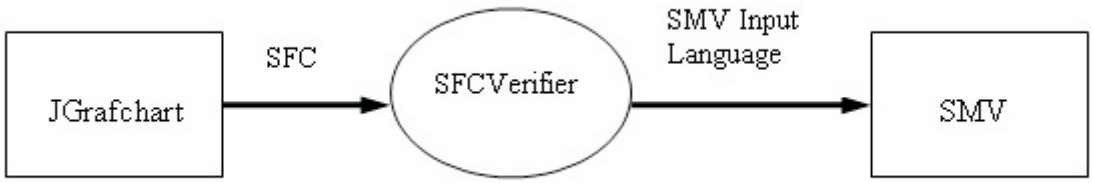


Figure 1.1. The flow of project.

model checker normally uses an exhaustive search of the finite state space of the system to determine if some specification (property of the system) is true or not (Clarke *et al.* 2000). Since SFC is a high level structuring language based on transition systems, it is easy to find a finite abstraction for a SFC program (Huuck 2003). Once a finite abstraction is found and system is translated for a Model Checker, different semantic properties can be verified.

Applying model checking to a design consists of several tasks:

1. Modeling

2. Specification

3. Verification

The process of model checking can be best illustrated by Figure 1.2 (Franceschet 2003).

The first task is to convert the design of the system into an abstract model accepted by model checker. The conversion/translation is done automatically by SFCVerifier, details of translation techniques can be found in Chapter 3. Once modeling is done, the properties of system that must be satisfied can be specified. The specification is usually given in some logical formalism. It is common to use temporal logic, which can assert how the behavior of the systems evolves over time(Franceschet 2003). After specification of properties in temporal logic, the verification is done by the model checker. Ideally verification is completely automatic. However, in practice it often involves human assistance. One such manual activity is the analysis of the verification results. When the system fails to satisfy a desired property, the model checker produces a counter example also known as error trace that demonstrates a wrong behavior (Franceschet 2003). In case of SFC program verification, error trace shows the execution states with values of variables/signals. This information might be helpful in diagnosing the error.

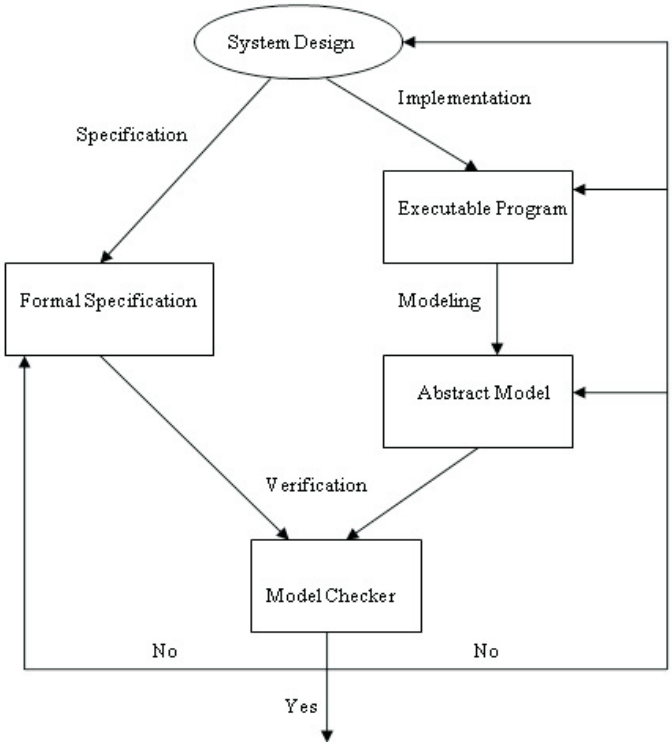


Figure 1.2. Process of Model Checking (Figure taken from (Franceschet 2003).

1.5 Challenges

There were some major challenges faced during this project. Although Sequential Function Chart (SFC) language is standardized in IEC 61131-3(IEC-61131 1998), but no formal semantics are given. The informal descriptions presented in the standard are often incomplete. Sometimes it was quite difficult to make a decision about different possible modeling and execution behaviors. The JGrafchart was only used for editing and structuring SFC programs, compiling the SFC programs in JGrafchart was not useful since it uses the other execution model not addressed in this verification project. For details about execution model addressed in this project see Chapter 2.

1.6 Contributions of this Thesis

In this work we have defined techniques and algorithms to translate Sequential Function Chart (SFC) constructs into abstract design accepted by Symbolic Model Verifier (SMV). These techniques are implemented in a software tool called SFCVerifier. The SFCVerifier is implemented in Java 2 and hence it is platform independent. The integration of JGrafchart with SMV is done through SFCVerifier to automate the verification process. The SFCVerifier has been successfully tested on number of

case studies.

2 SEQUENTIAL FUNCTION CHARTS

Sequential Function Chart (SFC) is defined in IEC 61131-3 standard (IEC-61131 1998) as elements of a program-structuring language for Programmable Logic Controllers (PLCs). SFC evolved through Grafcet (David 1995) from safe Petri nets (Fabian 2004).

Programmable Logic Controllers have been used in industrial applications since the early 70s. Originally, PLCs were designed to replace hard-wired relay-logic in applications where use of ordinary computers could not be economically motivated. The PLCs proved to be very versatile, and their application areas have increased constantly, as have their ability to handle more complex control issues. However, their original heritage still influences both their behavior and programming. To simulate the parallelism inherent in the wired relay-logic, a PLC executes cyclically, reading and storing the inputs, executing the entire user-program and finally writing the outputs. This read-execute-write cycle, called a scan cycle, effectively simulates parallel behavior from an input-output point of view. For the outside viewer, and specifically the plant, the output-signals change their state simultaneously in response to the input-signals, given that the scan cycle time is short with respect to the time constants of the plant (Hellgren *et al.* 2001).

A Sequential Function Chart depicts sequential behavior of PLC program. SFC program can be used at the top level to show the main phases of a process, such as, 'Startup', 'Pumping', 'Emptying' or the main states of a machine, like 'Running', 'Stopped' etc. It can also be used at any other level. For example, SFC can be used at low level to describe the behavior of a function block handling a serial communication device with various states such as 'Off', 'Carrier-Detected', 'Transforming' and so on (Lewis 1995).

SFCs are transition systems consisting of steps and transitions interconnected by directed links. For every SFC there is exactly one initial step, though JGrafchart (Årzén 2002, Johansson 1999, Årzén and Johansson 2002) allows more than one initial step in one SFC program but we have assumed it to be one in this project for clarity and simplicity. A SFC can be closed or open as shown in Figure 2.1. With each step there may be attached set of actions, and with each transition a transition condition. In this project we have assumed the transition condition to be a Boolean expression.

2.1 SFC Constructs

The Sequential Function Charts' language constructs, considered in this project for verification, are discussed one by one in following subsections.

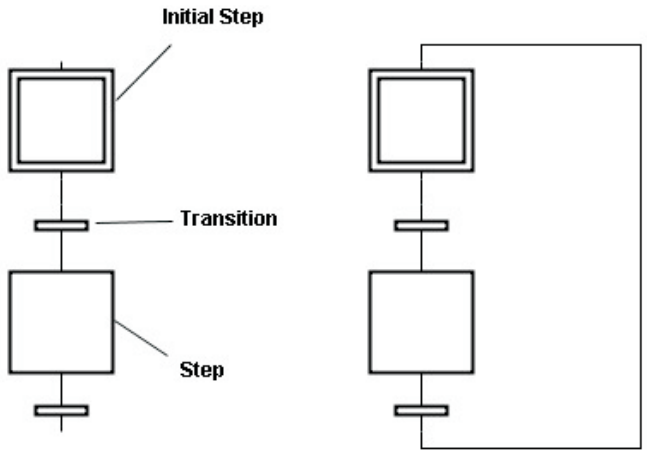


Figure 2.1. One closed and one open SFC.

2.1.1 Step

A step is graphically represented by a rectangular box usually with a name identifying the step as shown in Figure 2.2 the step is named S1.

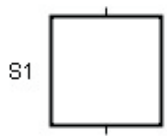


Figure 2.2. SFC Step.

2.1.2 Initial Step

The starting point in SFC point is known as initial step and as described earlier there is exactly one initial step in one SFC. The initial step is graphically drawn by a bordered rectangle (or double square) usually with a name as shown in Figure 2.3 the initial step is named S0.

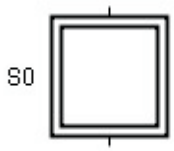


Figure 2.3. An initial Step.

2.1.3 Transition

The transitions are graphically represented by horizontal bar. Attached with every transition is a Boolean expression, as shown in Figure 2.4 the Boolean condition is 1 (logical True). This Boolean expression may involve variables, steps, input and output signals written in propositional logic.

Figure 2.4. A SFC Transition.

2.1.4 Actions

As described earlier that with each step there might be set of actions. The action block is drawn on the right of a step. The actions within action block are separated by semicolon (;). An example of a step with its associated set of actions is shown in Figure 2.5

In Figure 2.5 there are two internal Boolean variables named 'A1' and 'B1'. There are also used input and output signals named as 'Input' and 'Output' respectively. These variables are used within action blocks and referred in transition conditions in above example. Each action statement within action block has an action qualifier. The standard IEC 61131-3 defines a range of qualifiers which define precisely when a particular action executes in relation to its associated step(Lewis 1995). But in this project we consider only three kinds of action qualifiers as described below, based on JGrafchart (Årzén and Johansson 2002) language reference.

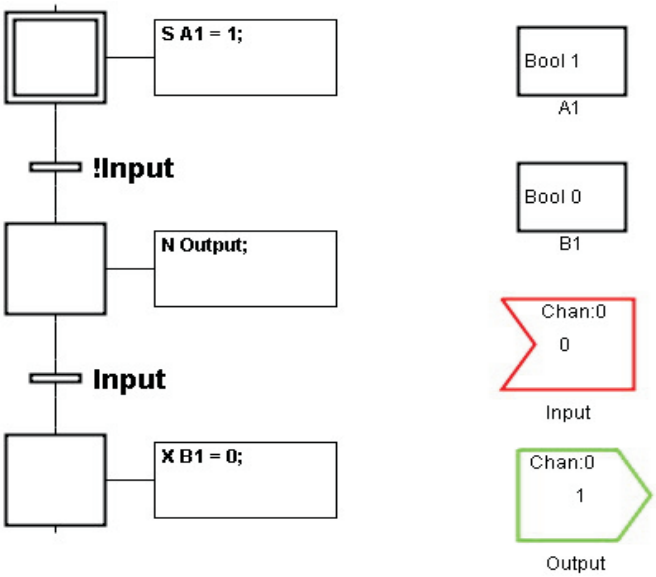


Figure 2.5. The SFC Actions.

Normal Action (Non-Stored Action)

In IEC-1131-3 standard (IEC-61131 1998) normal actions are defined for programs which execute when their respective steps are active. In this project we allow only to write a Boolean variable with normal actions and normal actions associate the truth-value with the activation status of the corresponding step, the variable becomes true when the program is running and false otherwise. Thus we model a program as a Boolean variable. The syntax for a normal action is

```
N ‘‘variable’’;
```

Enter Action (Stored Action)

An enter action is executed once when a step becomes active. The syntax for enter actions is

```
S ‘‘action’’;
```

Exit Action

An exit action is executed once immediately before the step is deactivated, the syntax for exit actions is

```
X ‘‘action’’;
```

The ‘‘action’’ in the syntax definitions is an assignment and its syntax is

‘‘variable’’ = ‘‘expression’’

Where ‘‘variable’’ is a reference to either an internal variable or an output variable. The ‘‘expression’’ is the value assigned to variable used on left side.

In Figure 2.5, the initial step has a stored action where it sets the value of variable ‘A1’ to 1 (logical True). This value will be set through out the program execution until it is changed again in the same step or in later steps. The step after initial step sends value to output signal named ‘Output’. Since this is a non-stored action, the output signal will have value until this step remains active. The last step has an exit action which sets the value 0 (logically False) to variable ‘B1’. This action will be executed as this current step deactivates or we can say when next step will be activated. As described earlier transitions have transition conditions which can refer to variables and input signals. In example discussed in Figure 2.5 the transition condition after initial step waits for input signal ‘Input’ to get low (logically not high/ not True) after it happens the initial step is deactivated and next step is activated. Similarly transition between last two steps waits for input signal ‘Input’ to get high (logical True). More about steps, transitions, actions and other SFC constructs can be found in (Lewis 1995).

2.1.5 Parallel Branching

The parallel constructs also known as parallel bars are used to indicate beginning and end of a parallel branch (Johansson 1999). To indicate divergent path (alternative path) a parallel split is used, and for convergent path a parallel join is used as shown in Figure 2.6

The SFC evolves, starting from the initial step, as the transition are fired, proceeding steps are deactivated, and successor steps are activated. SFC evolution can go along diverging paths (alternatives). When there are two or more transitions from a single step the sequence diverges to one, and only one, of the possible steps. In an SFC sequence only one step can be active at a time, unless the parallel (sub) sequence is used as shown in Figure 2.6. Here, when step S1 is active and transition T1 evaluates to True, the transition activates both S2 and S4. This sub-sequence once initiated continues to evolve in parallel until they at some point converge. In Figure 2.6 convergence occurs at steps S3 and S5. Only when both of these steps are simultaneously active the transition to S6 will take place. Thus, we have sort of synchronization of sub-sequences. The step S6 will only be active when both steps S2 and S4 are active simultaneously and transition T4 evaluates to True. There is no restriction on subsequences, any valid SFC constructs are allowed. However, some allowable constructs may lead to very unsafe behavior, such as one shown in Figure 2.7.

In Figure 2.7, when step S4 is active and T6 is true (while T3 is false, in case user

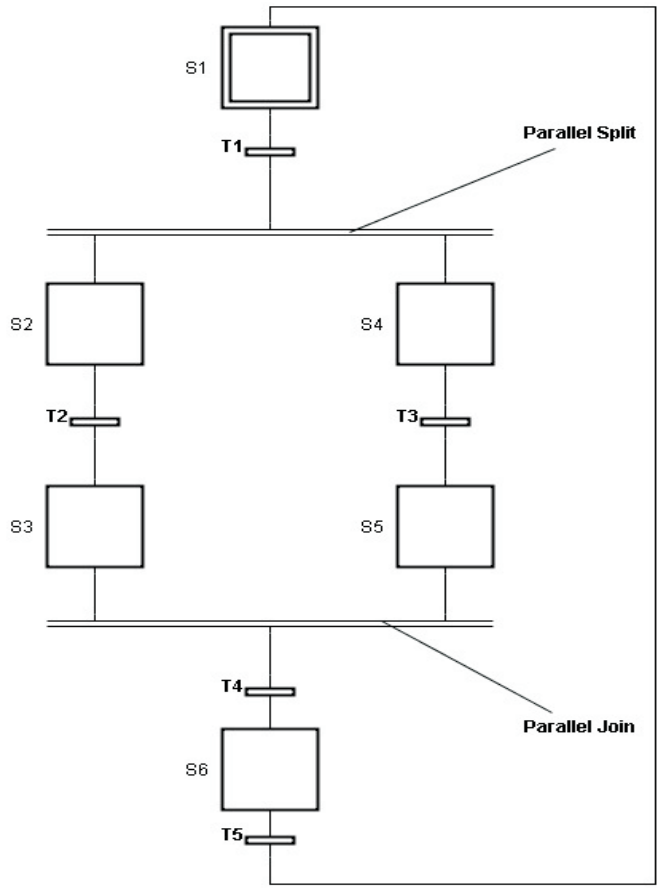


Figure 2.6. Parallel Branching within SFC, example based on (Fabian 2004).

has defined Left priority) the SFC branches out of the parallel sub-sequence S4-S5 and activates initial step S1. The active step in S2-S3 sequence remains active, so now we suddenly have S2 or S3 active together with S1. If T1 again becomes true it will happen that either both S2 and S3 are active simultaneously, or S2 (or S3) is re-activated without being deactivated. In either case, the behavior is unpredictable and may even be catastrophic. Other examples of unsafe designs can be found in (Lewis 1995) as well as in the IEC 61131-3 standard (IEC-61131 1998). There can be more than one SFC running concurrently, may share variables or signals, may depend upon each other's actions. This project also deals with verification of multiple SFCs.

2.2 SFC Execution Models - Scan Cycles

The standard defines rules for building an SFC from aforementioned basic elements and describes how to execute SFCs by giving evolution rules similar to the firing

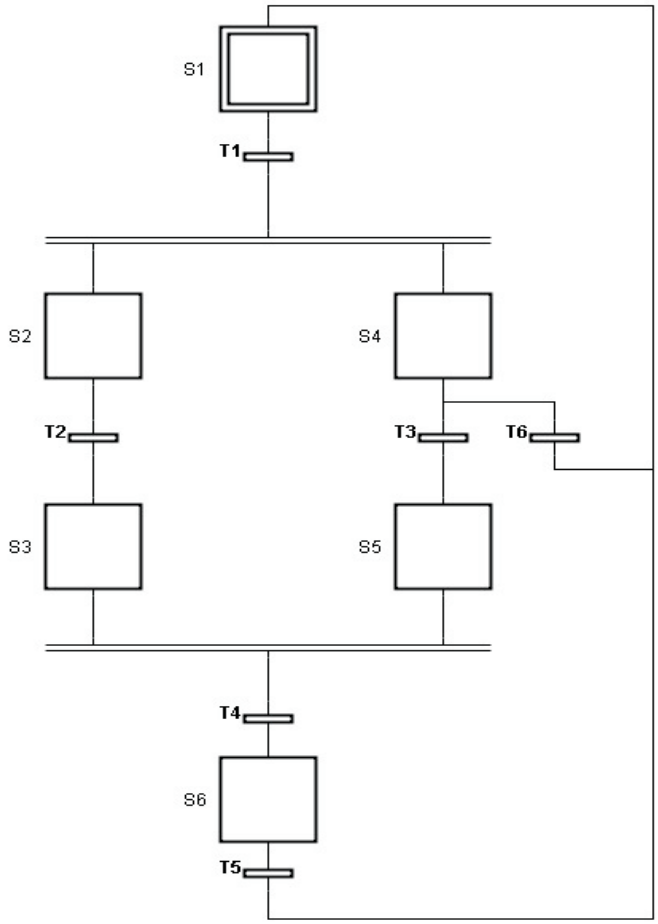


Figure 2.7. Unsafe SFC, example based on (Fabian 2004).

rules of Petri nets. However, execution is only defined on an abstract level not taking into account concrete aspects of program execution. Since PLC programs are cyclic in nature and therefore are SFCs. In every scan cycle fist input from environment (e.g. from sensors of a plant such as pressure or temperature sensors) is read and stored. Then the PLC program is executed based on the stored input, i.e. actions of the active steps are executed, which may change the output and afterwards the transitions are taken. At the end of each cycle the output is sent to environment, i.e. to the actuators of a plant such as valves and motors (Huuck 2003).

From this 3-stage execution cycle there are two possible execution models. In first one after reading inputs transition is evaluated and if it is enabled then transition is fired and exit actions of the old step(s) are executed with enter actions of the new step(s). This continues with the next transition. This execution model is also referred to as “immediate transit” and is used by some school of thoughts and it is easy to implement mutual exclusion by this model. In second execution model after reading inputs all enabled transitions are found and then all exit and enter actions are executed. In this way it is easy to implement synchronization. This execution

model is called "deferred transit". In this project we have chosen first approach, the "immediate transit".

The two execution models are illustrated in Figure 2.8 and Figure 2.9, these examples are taken from (Hellgren *et al.* 1999). In IEC-1131 standard the `.X` notation with step-name represents activeness of step. For example if we write `StepName.X` then it will return true when step is active and false otherwise. Here in following examples, within transition conditions, we assume to have `.X` as postfix with step-names.

Let us first examine Figure 2.8. The two SFCs are supposed to mutually exclude each other from having steps S2 and S4 active simultaneously. Assume steps S1 and S3 are active in beginning. In the case that deferred execution model is used both step S2 and S4 will be activated. Indeed, when transition conditions are evaluated neither S2 nor S4 is active and transition conditions `''!S4''` and `''!S2''` are both true. On the other hand, when immediate transit execution model is used, the mutual exclusion works. Let for instance the left SFC be checked first. Then, because S3 is active, `''!S4''` is true and S1 is deactivated and S2 is activated. The transition condition `''!S2''` is now false so that S4 is not activated, as intended.

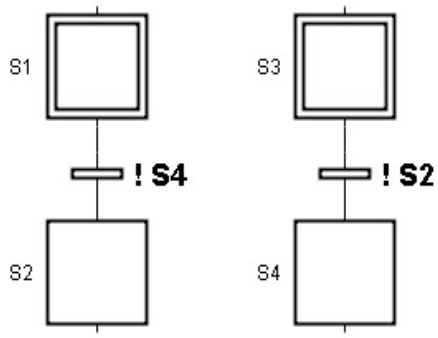


Figure 2.8. Mutual exclusion based on execution models, example based on (Hellgren *et al.* 1999).

Let us now examine Figure 2.9. The two SFCs are to execute the transitions leading to steps S2 and S4 synchronously. Again, assume that step S1 and S3 are active in beginning. In the case that deferred transit execution model is used, the synchronization works. Indeed, when transition conditions are evaluated both transition conditions, `''S3''` and `''S1''` are true. Consequently, both S2 and S4 are activated. On the other hand, in case that the immediate transit execution model is used, the synchronization fails. Let for instance the left SFC be checked first also in this case. Then, because `''S3''` is true, S1 is deactivated and S2 is activated. The transition condition `''S1''` is now false so that S4 cannot be activated, thus preventing the intended synchronization to occur.

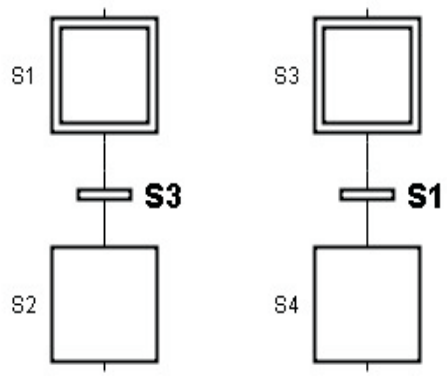


Figure 2.9. Synchronization based on execution models, example based on (Hellgren *et al.* 1999).

Abstract Formal Semantics

The efforts have been made to formalize the syntax and semantics for SFC. The (Bornot *et al.* 2000) and (Huuck 2003) provide the detailed abstract formal semantics. This formal framework serves as a basis for formal reasoning. The formal semantics are important because it provides alternative approaches to semantic ambiguities present in the standard. In this project we have taken into account the formal semantics already defined in above citations.

3 VERIFICATION OF SEQUENTIAL FUNCTION CHARTS

In this project verification is done by a Symbolic Model Verifier (SMV). The SMV is a formal verification tool, which verifies every possible behavior of the system for user defined specifications. The specification is collection of properties about the modeled system. For example in this project we can have properties about mutual exclusion, deadlocks, synchronization etc. These properties are defined in a notation called temporal logic. This allows concise specification about temporal relationships between signals. Temporal logic specifications about finite state systems can be automatically formally verified by a technique called model checking (McMillan 1999*a*). The SMV is quite useful in automatically verifying properties as it also generates counter examples i.e. the behavioral trace where violation is done.

As described in Chapter 1, there are three stages in model-checking process:

1. Modeling
2. Specification
3. Verification

We will discuss each of these stages with respect to Sequential Function Charts in following section.

3.1 Modeling

The modeling is in fact the translation of system into finite state space model. The design of the system is converted into an abstract model accepted by a model checker. This modeling phase is most time consuming and complex in whole project. One has to consider the whole system behavior and different interactions with environment. Error in modeling phase can lead to invalid verification results or uncertain behavior. For modeling, SMV has its own language.

The SMV language can be roughly divided into three parts - the definitional language, the structural language, and language of expression. The definitional part of the language declares signals and their relationship to each other. It includes type declarations and assignments. The structural part of the language combines definitional components. It provides language constructs for defining modules and structured data types, and for instantiating them. It also provides constructor loops, for describing regularly structured systems, and a collection of conditional structures that make describing complicated state transition tables easier. Finally expressions

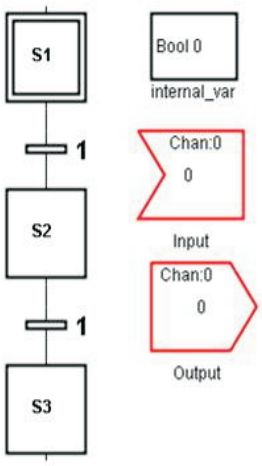
in SMV are very similar to expressions in other languages both hardware description languages and programming languages. It includes the constants, conditional operators, comparison operators, arithmetic operators and other common expressions found in programming languages. (McMillan 1999*b*)

Here modeling means representing the SFC as abstract finite system in SMV, which means translating SFC constructs into SMV input language. This translation should be automatic to avoid any mistakes possible in manual work. For this automatic modeling a software system is built in this project called SFCVerifier. A big advantage of this automatic translation by software is the reusability. Usually there are several constructs of SFC language in one program, like variables, signals, steps, transitions, parallel branching etc. There might be more than one SFCs in one program, so modeling needs careful representation of each and every construct to portray the accurate behavior of SFC program. For translation of these SFC constructs several rules were researched and are described in following section.

There is one main module in each SMV program and all the language constructs are defined and operated in this main module. Following are discussed the SFC constructs and their translation in SMV.

Variables

For each SFC Boolean variable, input/output signal there is defined a Boolean variable in SMV program module. In Figure 3.1, the SFC program variables are shown with their generated SMV variables. These variables are initialized as they were initialized in SFC program. If a variable was not initialized in SFC program, it's value will be remained undetermined in SMV program.




```
--This file is automatically generated
MODULE main()
{
    internal_var : boolean;
    Input : boolean;
    Output : boolean;
    S1 : boolean;
    init(S1) := 0;
    S2 : boolean;
    init(S2) := 0;
    S3 : boolean;
    init(S3) := 0;
```

Figure 3.1. Example of Variables in SFC with their SMV generated code.

Steps

For each step in SFC, a Boolean variable is defined in SMV program. The variable has the same name as that of step. If user has named the step then that name will be used otherwise name assigned by JGrafchart will be used. The variables representing steps are initialized to 0 (logical False) since steps are not in active mode in beginning. In Figure 3.1, on left side SFC with variables is shown while SMV generated code for variable definitions is shown on right side.

After SFC program starts the initial step becomes active. And when next transition becomes true the next step becomes active and previous step is deactivated. This sequential behavior is shown in SMV by using `'next'` operator. We can say "A step is active when it is entered from previous step, with incoming transition enabled, or it remains active for more than one scan cycle and is not left, no outgoing transition enabled". We will see this in next code example.

Transitions

The Transitions are also known as guards. When previous step is active with its stored and non-stored actions executed, the transition conditions are evaluated. It means transition conditions are evaluated conditionally. That's why we have used transition conditions in conditional operators, i.e. in `''If Else''` statements. The Figure 3.2 shows a simple SFC excerpt with its translation in SMV code.

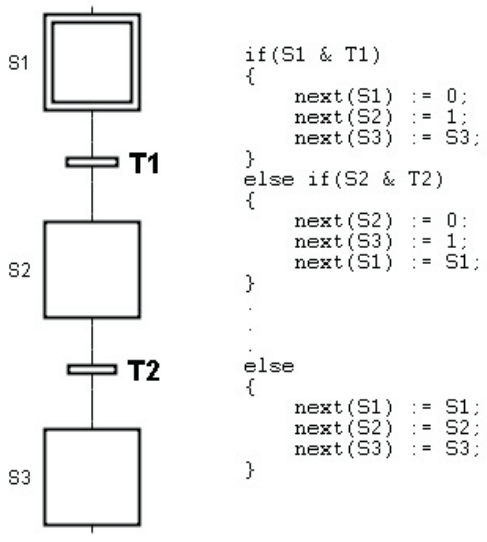


Figure 3.2. Example of Transitions in SFC with their representation in SMV generated code.

The first `'if'` block's condition can be read as "if step S1 is active and transition T1

is enabled" then in next sequence step S2 will be activated while deactivating step S1. Notably remaining steps, here step S3, will have their previous state. The last default block represents the case when there is no change in steps and transitions, that is when step remains to their previous state.

Actions

The actions are also translated in SMV since they have an important role in SFC programs. As we discussed before in Chapter 2 that in this project we have considered only 3 kinds of actions, i.e. Stored actions, Non-stored actions and Exit actions.

Stored Actions

For stored actions, the action is performed when its respective step is active. The action begins execution immediately when step becomes active. Since this action is stored, the action continues to execute until a reset action is reached. If a stored action is never reset, it will continue to execute indefinitely.

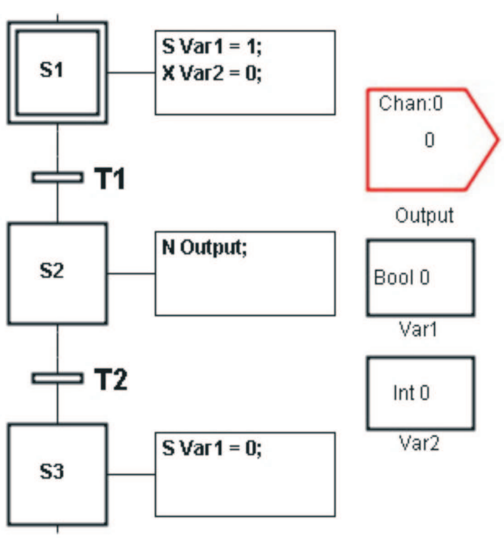
Non-Stored Actions

For non-stored actions the action is performed only when its respective step is active and its effect should remain until this step is deactivated and next step is activated. This behavior of non-stored action is simplified in SMV by having two actions for each non-stored action; the first action is "enter action" and second is "exit action". The "enter action" is performed like normal stored action, i.e. when step is active. The "exit action" is performed when step deactivates, i.e. when next step is activated, this "exit action" is inverse of it's enter action. This behavior is shown in Figure 3.3

Exit Action

Exit action is performed after its respective step is deactivated, i.e. when next step is activated. In Figure 3.3, an SFC example is shown with all the three types of actions and their SMV translations.

In example discussed in Figure 3.3, the first step S1 has two actions; one stored action and one exit action. The first "if statement" in code listing on right side shows stored action, the code line is also marked with arrow. While the first step's exit action is executed in second step's code block, i.e. in second conditional block, where variable 'Var2' is assigned a value 0 (logical False), this code line is also




```
if(start)
{
    next(S1) := 1;
->  next(Var1) := 1;
    next(S2) := S2;
    next(S3) := S3;
    next(Output) := Output;
}
else if(S1 & T1)
{
    next(S2) := 1;
->  next(Output) := 1;
->  next(Var2) := 0;
    next(S1) := 0;
    next(S3) := S3;
    next(Var1) := Var1;
}
else if(S2 & T2)
{
    next(S3) := 1;
->  next(Var1) := 0;
->  next(Output) := ~(Output);
    next(S2) := 0;
    next(S1) := S1;
}
```

Figure 3.3. Example of Actions in SFC with their SMV generated code.

marked with arrow for readability. This second conditional block has also another action which pertains to its step S2. Since this is a non-stored action therefore it is put in this block. But its exit action is performed when next step is activated, that's why its exit action is put in next conditional block, i.e. block pertaining to next step S3. This exit action is also marked with arrow in code listing.

Multiple SFCs

There can be more than one SFCs working concurrently in one program. As it has been discussed in Chapter 2 that there are three phases in a scan cycle namely.

- Input phase
- Execution phase
- Output phase

In case of Multiple SFCs, the Execution phase is divided between SFCs, thus introducing micro-cycles. The concept is that, after reading input, the steps in multiple SFCs are executed turn by turn and while switching from one SFC to another the input does not change. For this micro-cycle a variable named 'turn' is introduced. This scenario is depicted in Figure 3.4. In the Figure 3.5 is shown an excerpt of code generated for two SFCs, where micro scan cycles are introduced. This figure

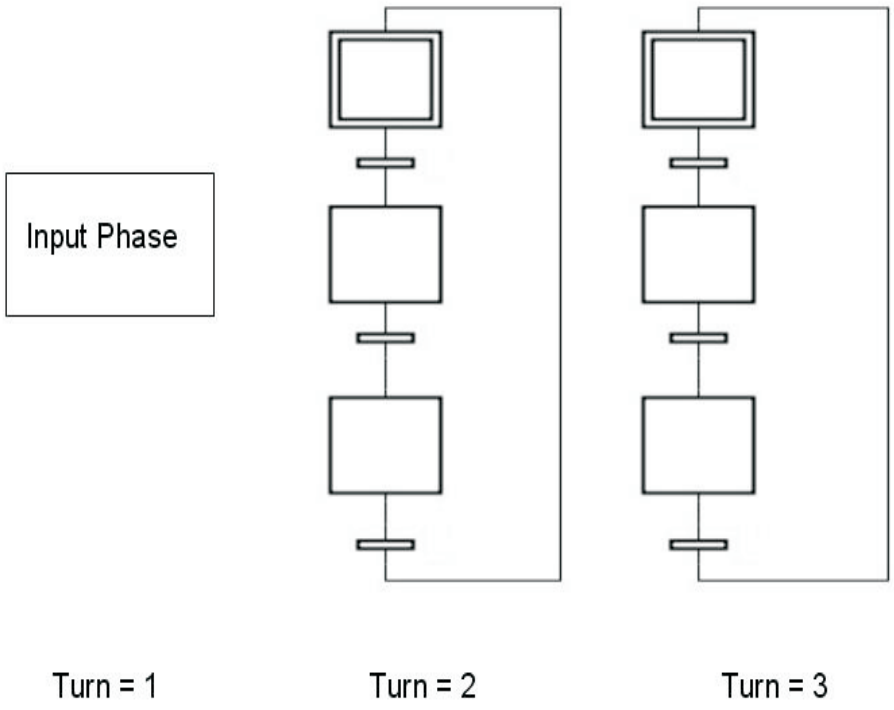


Figure 3.4. Describes how Multiple SFCs are modeled in SMV.

shows a rough structure of SMV code generated for two SFCs. The 'turn' variable can have value in range from 1 to (number of SFCs +1). This 'turn' variable is initialized to value 1. Then in each execution turn its value is changed as can be seen by code line:

```
next(turn) := (turn mod 3)+1;
```

The turn variable is assigned a new value (next value) depending upon previous value.

Parallel Branching

The parallel branching constructs in SFC program are also translated in SMV code. When control enters in a parallel split, more than one step gets activated and execution goes in concurrent fashion. To reflect this concurrency in SMV each parallel branch is represented as complete SFC program. This conversion can be seen in the example in Figure 3.6.

```
-- This file is automatically generated
MODULE main()
{
    turn : 1..3;
    init(turn) := 1;
    .
    .
    --**************SFC # 1***************
    if(turn = 1)
    {
        if(...)
        {   .
            .
            next(turn) := (turn mod 3)+1;
            .
            .
        }
        else if(...)
        {   .
            .
            next(turn) := (turn mod 3)+1;
            .
            .
        }
        else
        {   .
            .
            next(turn) := (turn mod 3)+1;
            .
            .
        }
    }
    --**************SFC # 2***************
    else if(turn = 2)
    {
            if(...)
            {   .
                .
                next(turn) := (turn mod 3)+1;
                .
                .
            }
            else if(...)
            {   .
                .
                next(turn) := (turn mod 3)+1;
                .
                .
            }
            else
            {   .
                .
                next(turn) := (turn mod 3)+1;
                .
                .
            }
    }
    --**************Input Phase***************
    else if(turn = 3)
    {   .
        .
        next(turn) := (turn mod 3)+1;
        --non-deterministic assignments for all Input signals like
        -- next(inputSignal1) := {0,1};
        -- next(inputSignal2) := {0,1};
        --for two input signals 'inputSignal1' and 'inputSignal2'
    }
    .
    .
```

Figure 3.5. Example structure of SMV generated code for Multiple SFCs.

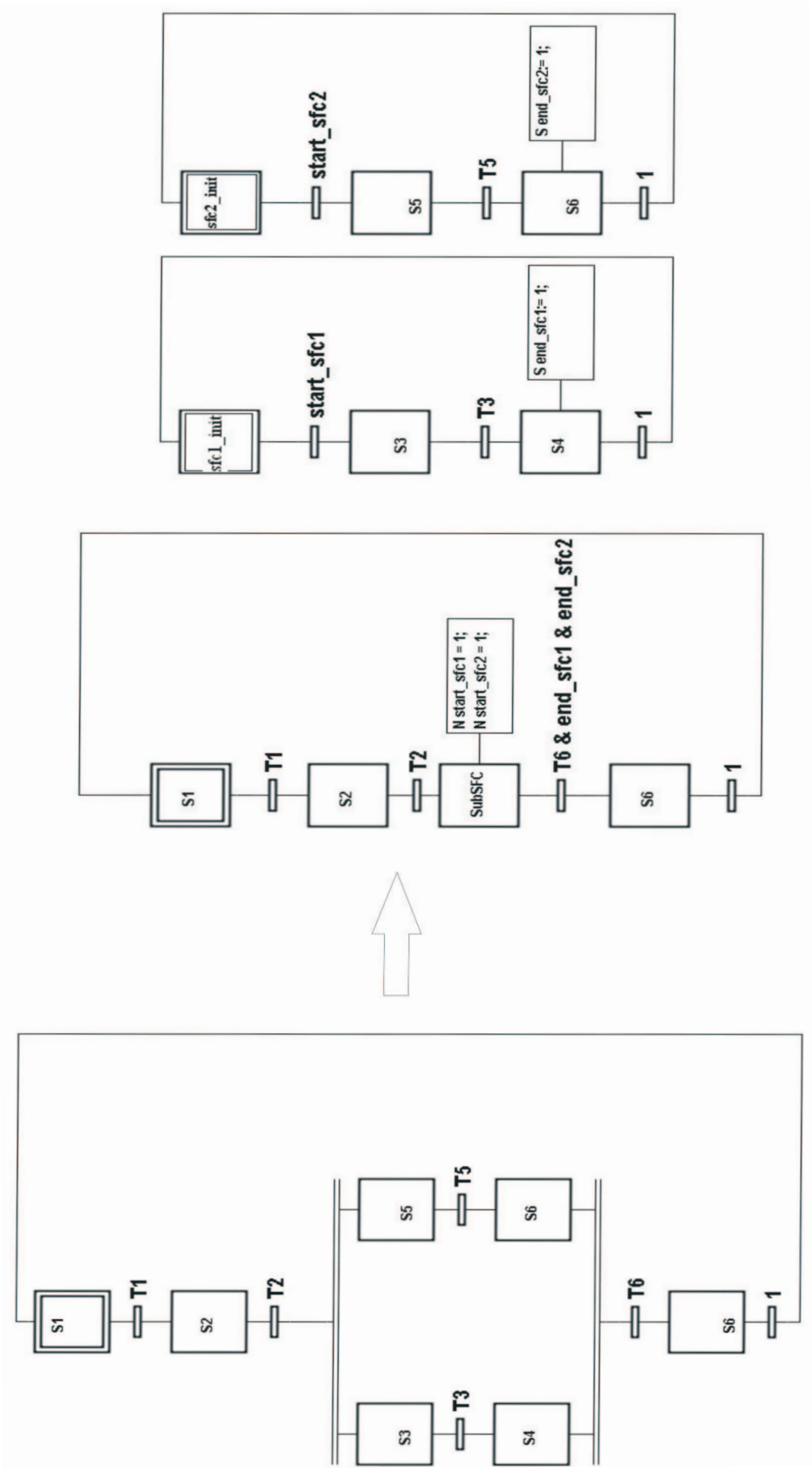


Figure 3.6. Describes how Parallel Branching is converted to be modeled in SMV.

In Figure 3.6, a SFC having parallel construct is split forming two more SFCs. The structure of original SFC is also changed. On right side of arrow is the original SFC with parallel constructs used, and on the left side are the transformed SFCs, where left most SFC after arrow is original SFC, here parallel split is replaced by a new step named 'SubSFC'.

The parallel branches are replaced by new SFCs, as shown in right most of the arrow. During this transformation few new Boolean variables are declared internally in data-structure. To each new SFC is added an initial step and a transition after initial step. These new-added transitions after initial steps have conditions, which will be enabled when the step SubSFC is active. Notice that step SubSFC has two non-stored actions, when this step is activated these actions are executed, making internally declared variables 'start_sfc1' and 'start_sfc2' True. These 'start_sfc1', 'start_sfc2' are in fact the start indication in their respective SFC. Similarly two internally declared variables 'end_sfc1' and 'end_sfc2' are present as stored actions in last steps of branch-converted-SFCs.

After transformation the execution process for SFC in Figure 3.6 will be: when step S2 is active and transition T2 is enabled, the step SubSFC is activated while deactivating step S2. After step SubSFC is activated its non-stored actions are performed. Since there are now three SFCs, due to transformation, the execution phase is divided among these three SFCs, as discussed in previous topic on Multiple SFCs. After performing of actions, when next turn for 2nd or 3rd SFC comes, the transition after initial step is fired, since initial steps were enabled in previous turns. Now at this stage step S3 and/or S5 are activated, depending upon turns, thus starting normal execution of branch-converted-SFCs. Turn for main SFC may come in between but it will not progress since its next transition is not enabled. The branch-converted-SFCs, when come to their last step, will make two internally declared variables 'end_sfc1' and 'end_sfc2' true, thus enabling transition after step 'SubSFC' in main SFC, which in turn will activate step S6 while executing "exit actions" of 'SubSFC'. This is how parallel branching or in other words concurrent behavior is modeled. The SMV generated code will have similar structure like shown in Figure 3.5 except that there will be total 3 SFCs for example discussed in Figure 3.6.

3.2 Specification

After modeling comes the specification phase. In this phase we specify the properties of system to be verified. These properties can be general for all SFCs or specific to particular SFC. The general properties include checking Reachability of each and every step, repeated Reachability (liveness / deadlock). The specific properties can be checking for synchronization, mutual exclusion or other safety related properties. These properties are specified in Temporal Logic (Pnueli 1981). Temporal logic formulas are interpreted over Kripke's structures (Kripke 1963). The temporal logic extends propositional logic, i.e., Boolean proposition with connectives such as

logical conjunction, disjunction and negation, with modal operators. The modal operators are operators like always or eventually, which allow reasoning over execution sequences and can be combined with the usual connectives (Huuck 2003). There are also other operators such as next operator, until operator etc. More details about these operators can found in (McMillan 1999*b*).

Reachability

To specify that whether a Step S1 is Reachable, we can write like this

```
assert F( S1);
```

Here assert shows that this property is certainly true. The 'F' operator stands for finally. There is an enclosed propositional formula with this operator. Thus we can read above as "step S1 must eventually (finally) be true (activated) after some finite time."

Repeated Reachability

To specify that whether Step S1 is Reachable infinitely often, we can write

```
assert G( S1 -> F(~S1) );
```

Here operator 'G' stands for globally (always) and the arrow operator is 'implies' operator also known as 'conditional operator'. We can read this formula as "its globally (always) true that if step S1 is true (activated) in some finite time then eventually step S1 will be false (deactivated) after some finite time." It means there will be no deadlock and an active step will not be active for infinite time, once it is activated it will be deactivated after some finite time.

User Defined Properties

The above stated reachability properties are generated automatically for each step in SFC. But user can also specify some other property and append it to already generated properties in SMV file. One example of user defined property is testing for mutual exclusion, for example mutual exclusion between two steps S1 and S6 can be stated in temporal logic as

```
assert G ~(S1 & S6);
```

Here '~' operator is unary operator 'not' and & operator is logical 'and' operator. We can read above formula as "it is globally true that step S1 and step S6 cannot be true simultaneously". If this property is violated it means there is some case where

both steps are active at the same time.

Fairness Properties

In SFC program digital inputs can be used. These digital inputs represent environment (e.g. sensors) and SFC program has no control over their value. If we wish to verify a SFC program with input signals then it may be required to verify it with changing value of input signals. It means that that input variables should change their value infinitely often, rather than sticking to some constant value. To model this behavior we add fairness constraints in our specification. For each input variable one fairness constraint is added. The fairness constraint is written like this

```
assert (G F(inputVar)) & (G F(~inputVar));
```

Here right-hand-side of '&' operator says that 'inputVar' must finally be true after some finite time and left-hand-side says that 'inputVar' must finally be false after some finite time. These fairness constraints are added for each input signal and reachability and repeated-reachability properties are verified assuming these fairness constraints.

3.3 Verification

After specifying properties in temporal logic, the SMV system can verify them. The verification results can be viewed through SMV viewer called 'vw'. If certain property fails during verification SMV provides a counter example also called as error trace. In Figure 3.7, is shown snapshot of SMV viewer where a counter example for modeled system is shown.

In Figure 3.7, on the left column are the names of variables, signals and steps. The rest of columns show values of these variables, signals and steps with respect to execution counter, that is the top most row. Here on 8th execution counter the property 'deadlockfree_S6 'is false. This property was specified as

```
deadlockfree_S6:  assert G (S6 -> F(~S6) );
```

The step S6 when activated never gets deactivated, as shown in the error trace in Figure 3.7.

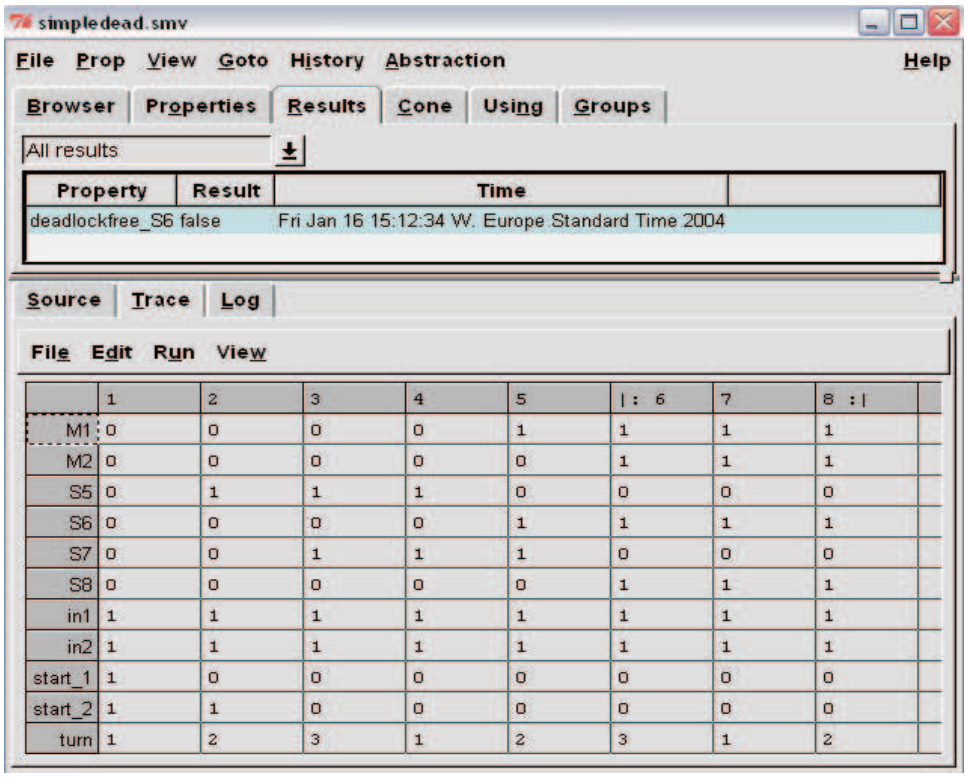


Figure 3.7. SMV during Verification process when it generated an error trace.

4 CONCLUSION

4.1 Summary

In this thesis work we have presented formal verification technique for PLC language Sequential Function Charts (SFC). A subset of SFC constructs is covered for this verification. Model checking is used as the main approach for verification purpose and model checking tool Symbolic Model Verifier (SMV) is used. Formal verification work has been processed through modelling, specifying and verifying SFCs. For modelling SFCs are translated into input language of SMV. The translation techniques are implemented by development of a software tool 'SFCVerifier'. JGrafchart, a programming tool for Sequential Function Chart, is used as source of SFC programs. The SFCVerifier converts these SFC programs into internal data structure to model a finite state system. The input language for SMV is generated by SFCVerifier with specification of properties in temporal logic. These properties can be general like reachability tests or it can be specific to system like mutual exclusion. Once processed by SMV these properties can be tested and for unverified properties error trace is generated. These error trace helps to check, when and where during the execution, problem occurred.

4.2 Future Work

The future work in this direction can be verification of hierarchical SFCs, timed SFCs, procedures, exceptions, other types of variables/signals and rest of other constructs not covered in this thesis. Since some research groups use execution model, in which enabled transitions are found and then exit and enter actions are executed. Verification of SFCs by modelling according to this execution model can be a good future work.

There is hardly any work done for verification of other PLC languages and since PLCs allow mixture of different programming languages, verification with respect to integration of different programming languages will certainly be big future challenge.

Bibliography

Årzén, Karl-Erik (2002). Jgrafchart. http://www.control.lth.se/∼karlerik/Grafchart/JGrafchart.html, last visited April 2004.

Årzén, Karl-Erik and Johansson, Charlotta (2002). *JGrafchart Language Reference.* available from *karlerik@control.lth.se.*

Bornot, S., Huuck, R., Lakhnech, T. and B.Lukoschus (2000). An abstract model for sequential functiom charts. In: WOODES 2002, 5th Workshop on Discrete Event Systems, Ghent, Belgium.

Clarke, Edmund M., Grumberg, Orna and Peled, Doron A. (2000). Model Checking. *MIT Press.*

David, R (1995). Grafcet: A powerful tool for specification of logic controllers. IEEE Transaction on Control Systems Technology.

D.L'Her, Parc, P.Le and Marc, L (1995). Proving sequential function chart programs using automata. In: Proceedings of 2nd AMAST workshop on Real-Time Systems.

Fabian, Martin (2004). Control and communication systems. Lecture notes. Signals and Systems, Chalmers University of Technology.

Franceschet, Massimo (2003). Lecture notes. http://staff.science.uva.nl/∼schlobac/Teaching/AR2003/massimo_1.pdf, last visited May 2004.

Hellgren, Anders, Fabian, Martin and Lennartson, Bengt (1999). Synchronized execution of discrete event models using sequential function charts. In: Conference on Decision and Control. *IEEE.*

Hellgren, Anders, Fabian, Martin and Lennartson, Bengt (2001). On the execution of discrete event systems as sequential function charts. In: Conference on Control Applications. *IEEE.*

Huuck, Ralf (2003). Software Verification For Programmable Logic Controllers. PhD thesis. Christian-Albrechts-University of Kiel.

IEC-61131 (1998). Programmable controllers - programming languages, second edition. Committee draft IEC 61131-3. International Electrotechnical Commission, Technical committee No. 65.

Johansson, Charlotta (1999). A Graphical Language for Batch Control. PhD thesis. Department of Automatic Control, Lund Institute of Technology.

Kripke, Saul A (1963). Semantical consideration on modal logic. Technical Report 16:83-94. Acta Philosophica Fennica.

Lewis, R.W (1995). Programming Industrial Control Systems Using IEC1131-3. *The Institution of Electrical Engineers, London, UK.*

*McMillan, Kenneth L. (1999*a*).* Getting starting with SMV. *Cadence Berkeley Labs.*

*McMillan, Kenneth L. (1999*b*).* The SMV Language. *Cadence Berkeley Labs.*

Pnueli, Amir (1981). The temporal semantics of concurrent programs. Technical Report 13:1-20. Theoretical Computer Science.

Appendix A

Case Studies

In this chapter we have presented verification case studies for some Sequential Function Chart (SFC) examples. The SFC programs are shown with their verification issues. The tools used are JGrafchart for SFCs, SFCVerifier for analysis + code generation and Cadence SMV for verification of generated code. For all the discussed example the SFC programs executable in JGrafchart and SMV code generated by SFCVerifier can be found on following web address

http://www.s2.chalmers.se/~knut/masterthesis/2004/malik

A.1 First Case study

In this example two concurrent SFC structures are verified. These concurrent structures interact with each other and there is possibility of deadlock as shown in Figure A.1

The two SFCs start from their initial step, and execution control will advance depending upon the values of input variables 'in2' and 'in1'. The initial steps are named S1 and S3 in their SFCs. These structures were analyzed and SFCVerifier and SMV code was generated. When verification was applied to the generated code in SMV, the SMV system detected a deadlock with error trace. The trace showed a deadlock possibility where both step S2 and step S4 were active at the same time setting variables M1 and M2 to true respectively. The step S2 cannot leave because it's outgoing transition is not enabled because M1 is true. The step S4 cannot leave because it's outgoing transition is not enabled because M2 is true. Both of these steps wait for each other since the exit action for step S2 will be executed in step S1 and for S4 it's exit action will be executed in step S3. Thus these two steps will wait for each other infinitely without proceeding.

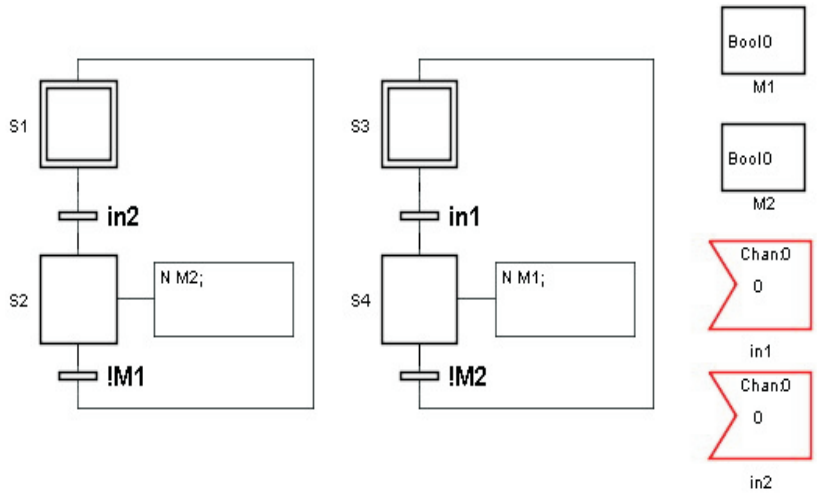


Figure A.1. SFC program with possible deadlock.

Second attempt

In second attempt we modified our example to avoid deadlock and then tried verification. As shown in Figure A.2, now the condition has been added before step S2 that it cannot enter step S2 until variable M2 is false, similarly another condition is added before step S4 that it cannot enter step S4 until variable M1 is true

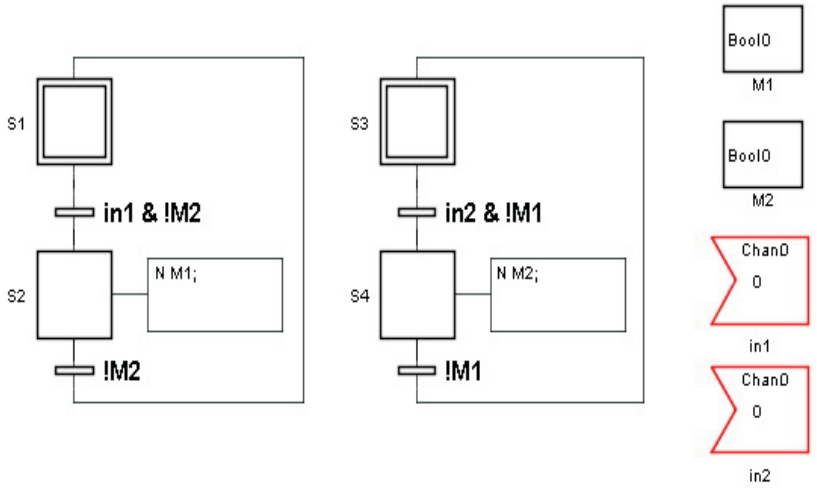


Figure A.2. Improved SFC program with non-trivial deadlock possibility.

Now this should avoid any deadlock. When this program is verified through SFCVerifier it complained about deadlock. Precisely speaking this time it was not a deadlock but a starvation instead. There is a possibility that input variable 'in1' is false at the same time when M2 becomes false. Since input variables are something not

controlled by SFC program, their value come from environment, one cannot predict about their behavior. We have added fairness constraints that every input variable should change its value infinitely often. The model checker follows this fairness and changes the value of input infinitely often but makes the input 'in1' false when 'M2' becomes false, thus conjunction will always evaluate to false. But on the other hand it will allow one SFC to execute infinitely often while starving the other. This is a nontrivial situation and can happen in practice.

Third attempt

To avoid this starvation we modify our example to add one extra step as shown in Figure A.3. Now this time step S1A and step S3A have been added in their respective SFCs. Now once input signal is true the execution control will proceed and there is no deadlock or starvation as verified by SMV.

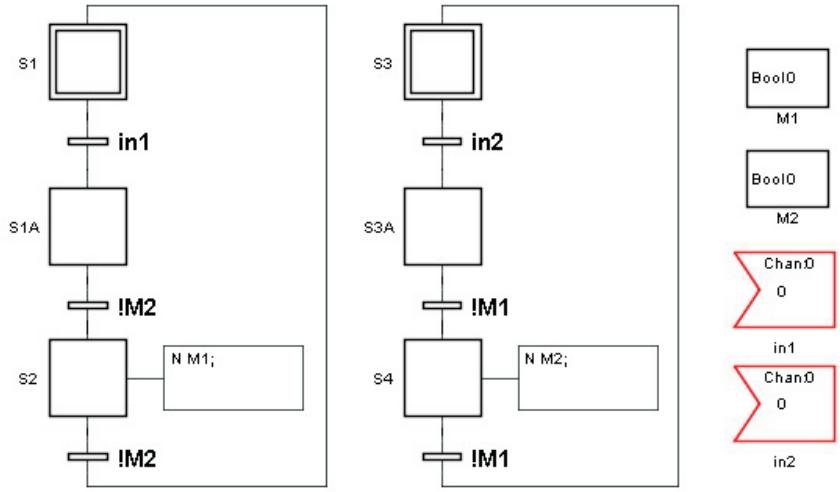


Figure A.3. SFC program free of deadlock and starvation.

A.2 Second Case study

This example is taken from one of the exam solutions by Martin Fabian, at department of Systems and Signals at Chalmers University of Technology, for course in Control and Communication Systems.

The Problem

There is a system consisting of three resources (machines) M1, M2 and M3. The system has to manufacture two products A and B. These two products are to be manufactured in a concurrent fashion. Both of the products use all the three machines. The product A uses machines in a sequence that first it acquires machine M1 and then while holding machine M1 it acquires machine M2, then it leaves machine M1 and while holding machine M2 it acquires machine M3. Finally it leaves machine M2 and completes the process with machine M3. The product B uses machines in a bit different sequence. The machine usage sequence for both products A and B can be seen in Figure A.4.

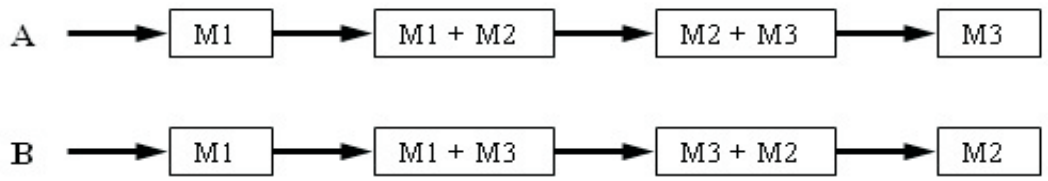


Figure A.4. The sequence of machine usage by products A and B.

The output signals are:

USE_M1: It indicates that machine M1 is currently in use.
USE_M2: It indicates that machine M2 is currently in use.
USE_M3: It indicates that machine M3 is currently in use.

The input signals are

M1_OK: It indicates that machine M1 is free for usage.
M2_OK: It indicates that machine M2 is free for usage.
M3_OK: It indicates that machine M3 is free for usage.
M1_M2_OK: It indicates that machines M1 and M2 are free for usage.
M1_M3_OK: It indicates that machines M1 and M3 are free for usage.
M2_M3_OK: It indicates that machines M2 and M3 are free for usage.

The products are manufactured concurrently and there is only one slot in each machine, means one machine can handle only one product at a time and these machines should be handled in a mutually exclusive way.

Solution

The SFC program for above stated problem is shown in Figure A.5

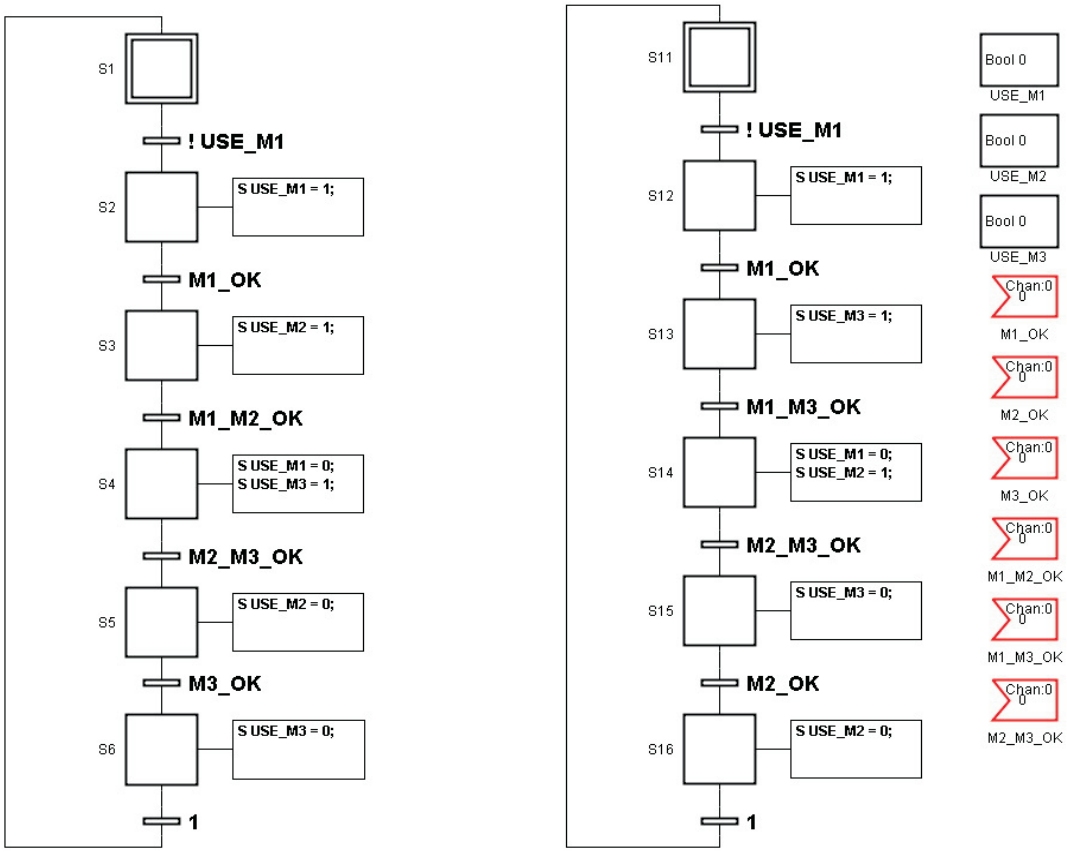


Figure A.5. SFC program with one slot per machine.

When this structure is analyzed in SFCVerifier and it's generated code is tested in SMV it proved to be a correct program without any deadlock or starvation problem.

Extension to the Problem

In SFC program shown in Figure A.5, the two SFC structures are concurrent but this is not an efficient system because one of the products has to wait for acquiring the machine since it is used by other product. The Figure A.6 shows an implementation where capacity of machine M1 is increased. Now machine M1 can handle two products at the same time. One in slot M1a and other in slot M1b. Each slot can

handle exactly one product.
For this system, the output signals are:
USE_M1a: It indicates that machine slot M1a is currently in use.
USE_M1b: It indicates that machine slot M1b is currently in use.
USE_M2: It indicates that machine M2 is currently in use.
USE_M3: It indicates that machine M3 is currently in use.

The input signals are:
M1a_OK: It indicates that machine slot M1a is free for usage.
M1b_OK: It indicates that machine slot M1b is free for usage.
M2_OK: It indicates that machine M2 is free for usage.
M3_OK: It indicates that machine M3 is free for usage.
M1a_M2_OK: It indicates that machine slot M1a and machine M2 are free for usage.
M1b_M2_OK: It indicates that machine slot M1b and machine M2 are free for usage.
M1a_M3_OK: It indicates that machine slot M1a and machine M3 are free for usage.
M1b_M3_OK: It indicates that machine slot M1b and machine M3 are free for usage.
M2_M3_OK: It indicates that machines M2 and M3 are free for usage.

Now in this system we can handle both products A and B in Machine M1 at the same time. But after using machine M1 acquiring further machines without any synchronization can result in a deadlock as verified by SMV for SFC given in Figure A.6

If execution control enters in steps S3 and S15 at the same time or in steps S5 and S13 at the same time then it will result in a deadlock in future since these (steps) will try to use machine which other step is using and each step will wait without releasing machine needed by other step, thus resulting in a deadlock.

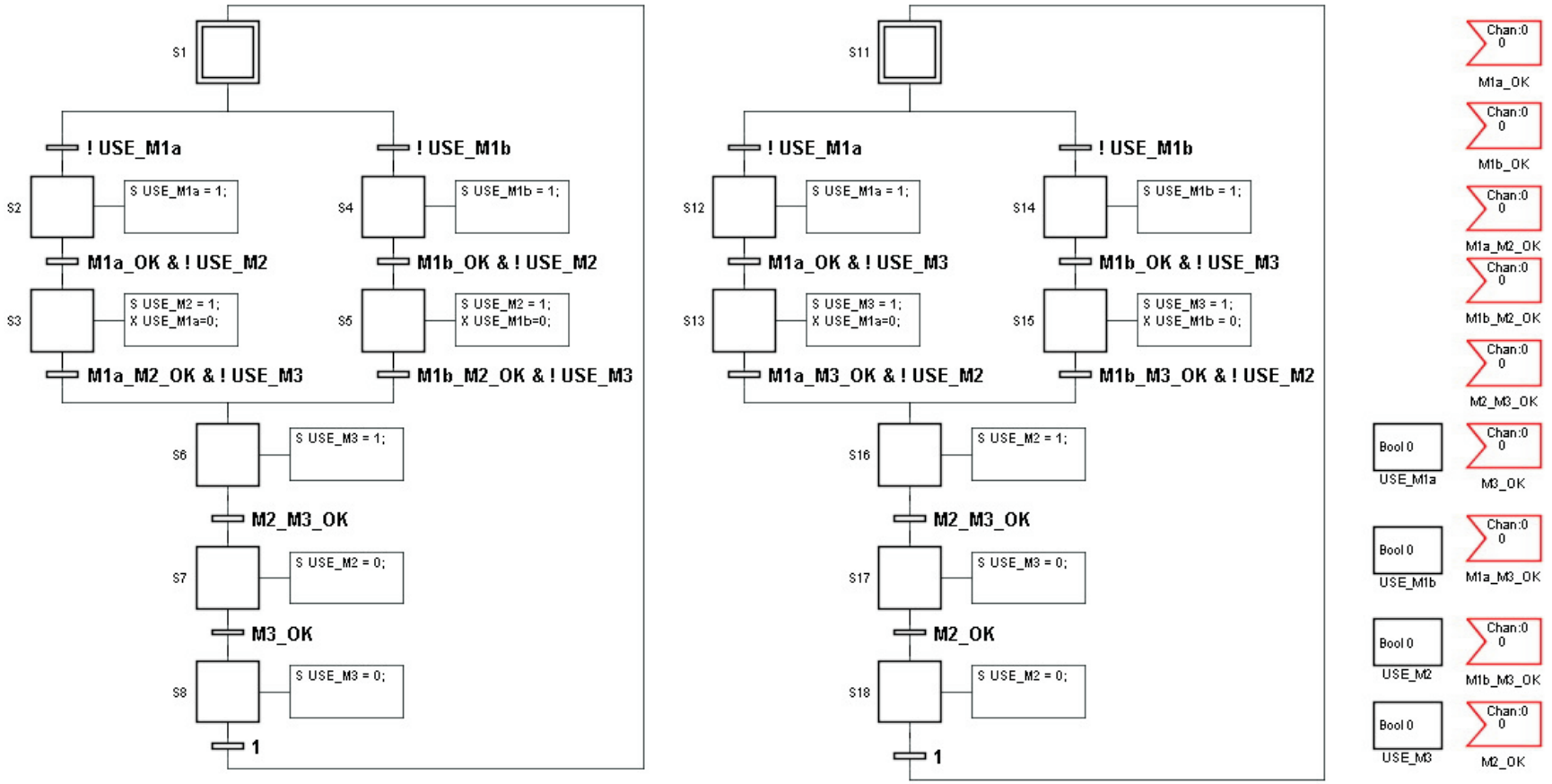


Figure A.6. SFC program with two slots for machine M1.

To avoid this problem we will use semaphore for synchronization. The semaphore-implemented SFC is shown in Figure A.7

When this implementation was tested, it avoided deadlock but it complained about possible starvation. Since conditions before steps S3, S5, S13 and S15 are dependent upon conjunction of output signal and input signal, and the local variables are controlled in other SFC. There might be the case that one SFC continues to execute and SMV chooses the value of input signal to be false when local variables' conditions evaluate to true, thus making the whole proposition false. To avoid this starvation problem we changed our SFC structures to add one more step in each branch. The new modified SFC program is shown in Figure A.8

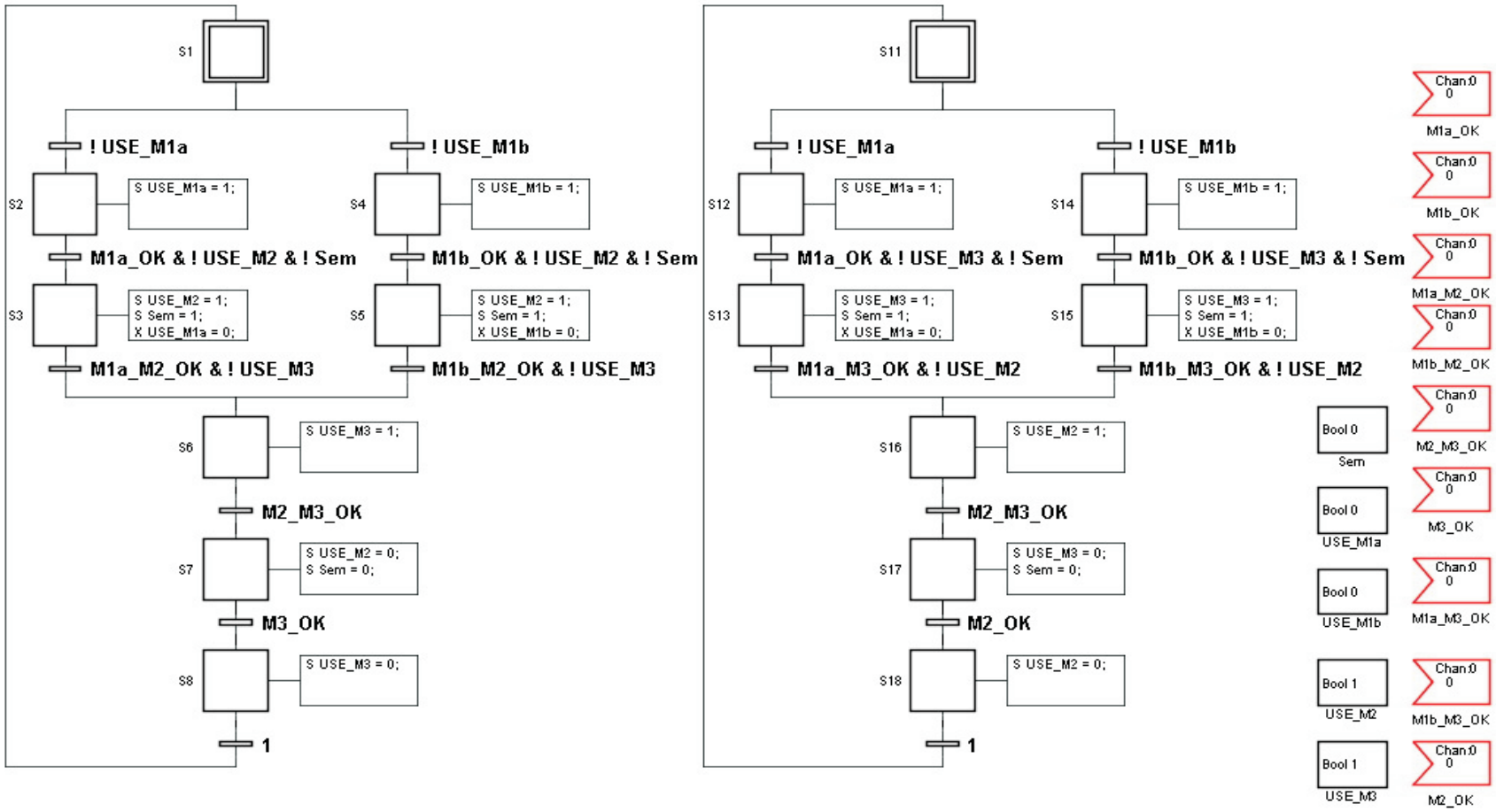


Figure A.7. SFC program with two slots for machine M1 and with Semaphore implementation.

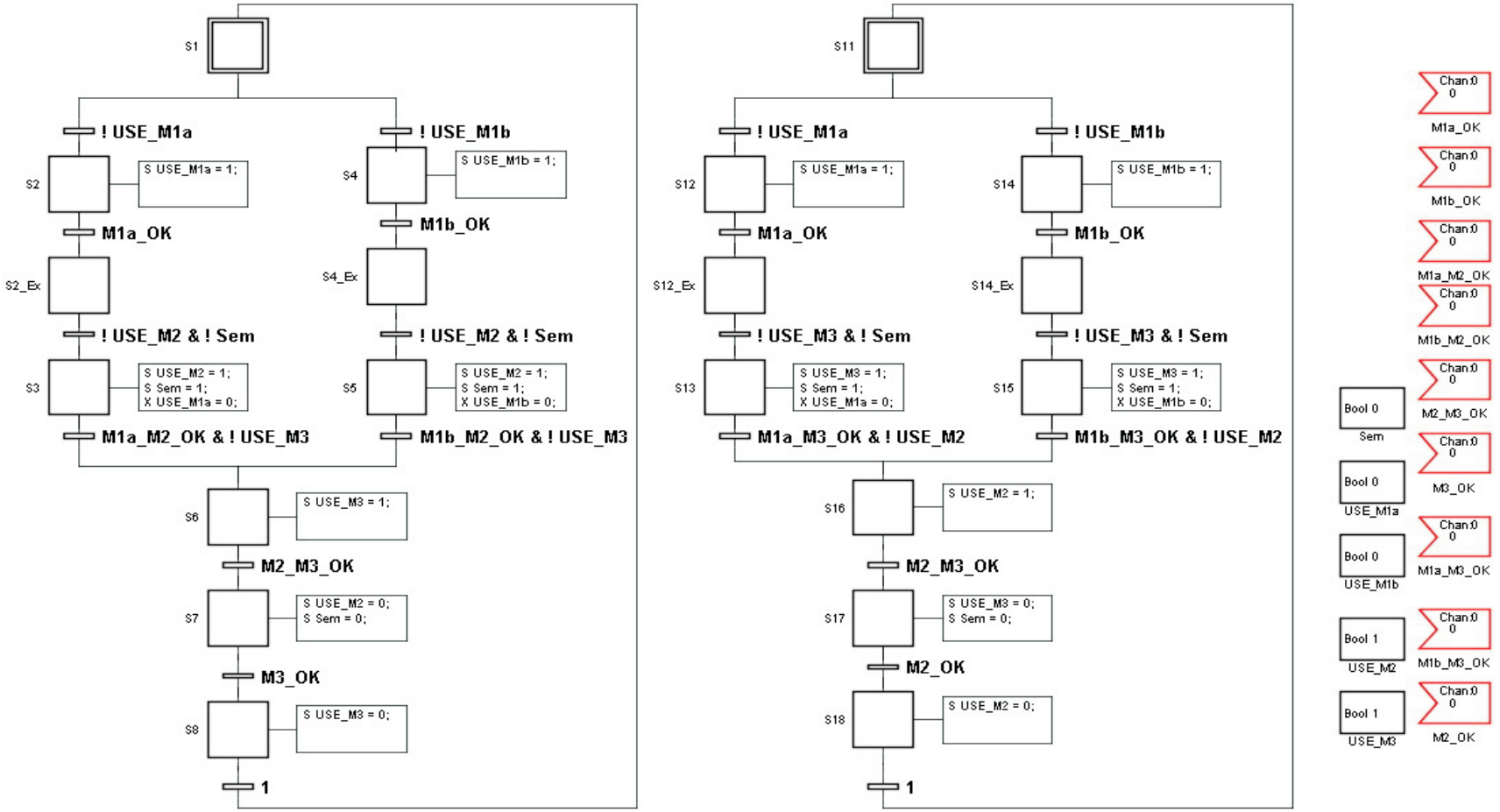


Figure A.8. SFC program with two slots for machine M1,a Semaphore and extra step to avoid starvation.

When this system was verified it complained about possible starvation of branch (note that before it complained about SFC starvation). There is a possibility that due to timing of input signals it always takes right branch in first SFC and left in second SFC or vice versa. It means we should implement a fair scheduling in our SFC to avoid such starvation. This is done in SFC shown in Figure A.9. This implementation verified all properties to be true, thus is a complete accurate SFC program. We don't always need to change our SFC program in case of starvation; the other possible solutions can be inclusion of constraints in the system which does not include such traces where input signals are changing their values in order to negate conjunction. Another solution is to ignore starvation errors (where possible) raised by SMV and checking of other important properties.

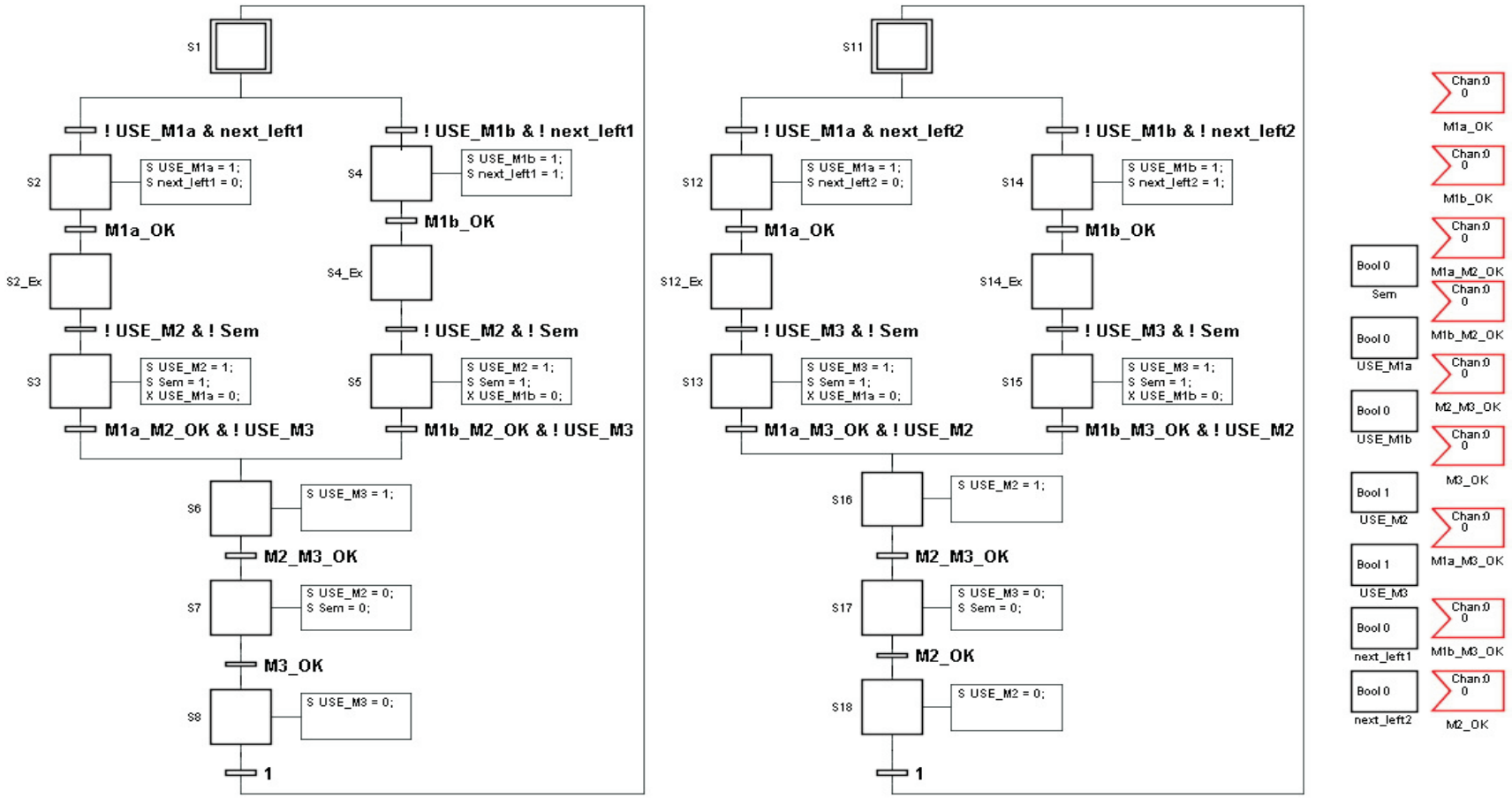


Figure A.9. Improved version of previous SFC program to avoid branch starvation.