TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Kazi Taher Uddin 166804IVSM

TRANSFORMATIONS BETWEEN TWO STATE MACHINE DIALECTS – SDL AND STATEFLOW

Master's thesis

Supervisor: Andres Toom
MSc
Tõnu Näks
MSc

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Kazi Taher Uddin 166804IVSM

MUDELITEISENDUSED KAHE OLEKUMASINA KEELE VAHEL - SDL JA STATEFLOW

Magistritöö

Juhendaja: Andres Toom
Tehnikateaduste magister
Tõnu Näks
Tehnikateaduste magister

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Kazi Taher Uddin

08.05.2018

Abstract

In real time software development, state machines are very widely used for presenting the concepts and logic of systems and for their simulation. Model-driven software engineering (MDE) seems to gain popularity and state machine based formalisms are a natural component of many MDE approaches. Stateflow and SDL are two widely used state machine oriented modeling languages. SDL is formally defined and standardized whereas Stateflow has an excellent tool support and integration with other MATLAB based tools. Given that these languages have a common logical basis, but their tool support offers different possibilities, it would be interesting to exchange models between them. This work address the conversion between these two languages.

First, a possible semantic mapping from Stateflow to SDL state machine and vice versa is explored. The mapping include events, guards, composite states, parallel states, temporal logic and so on. The mappings are validated using two case studies, which show that the mappings are adequate for the selected language subsets.

A feasible implementation mechanism is proposed and required algorithms are provided. The source model is first converted to an intermediate JSON data structure and, then converted to the target language. The structure of the intermediate data structure is very simple and self explanatory.

Four algorithms are provided for implementing the mappings through the intermediate data structure. Required learning materials are referred and a simple prototype is under development.

The thesis has been written in English and contains text on 81 pages, 5 sections, 36 figures and 17 tables.

Annotatsioon

Mudeliteisendused kahe olekumasina keele vahel - SDL ja Stateflow

Reaalajasüsteemide arenduses kasutatakse süsteemide kontseptsioonide ja loogika esitamiseks ning nende käitumise simuleerimiseks tihti olekumasinaid. Mudelipõhine tarkvaraarendus (MDE) näib koguvat populaarsust ja olekumasinatel põhinevad formalismid on paljude MDE meetodite osaks. Stateflow ja SDL on kaks laialt levinud olekumasinatele orienteeritud modelleerimiskeelt. SDL keele semantika on formaalselt defineeritud ning see keel on standardiseeritud, samas Stateflow keelel on suurepärane tööriista toetus ja see keel on hästi integreeritud teiste MATLABi-põhiste tööriistadega. Arvestades, et neil keeltel on ühine loogiline alus, kuid nende arendusvahendid pakuvad erinevaid võimalusi, oleks huvitav mudelite ristkasutuse võimaldamine. Käesolev töö käsitlebki mudeliteisendusi nende kahe keele vahel.

Käesolev töö uurib esmalt Stateflow ja SDLi olekumasinate elementide semantika võimalikke teisendusi mõlemas suunas. Seejuures käsitletakse sündmusi, ülemineku tingimusi, hierarhilisi ja paralleelseid olekuid, ajalist loogikat jne. Teisenduste teostatavust ja korrektsust on kontrollitud kahe suurema mudeli teisendamise ning nende käitumise võrdlemise teel.

Teisenduse efektiivseks automaatseks realiseerimiseks on töös väljapakutud lahendusskeem ja vastavad algoritmid. Lähtemudel teisendatakse kõigepealt vahpealsesse JSON andmevormingusse ja sealt edasi sihtkeelde. Kasutatav vaheandmestruktuur on väga lihtne ja arusaadav.

Töö sisaldab nelja algoritmi mudeliteisenduste realiseerimiseks läbi vaheandmestruktuuri. Töös on viidatud teemakohastele taustmaterjalidele ning teisenduste automaatseks teostamiseks on loomisel lihtne prototüüp.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 81 leheküljel, 5 peatükki, 36 joonist ja 17 tabelit

List of abbreviations and terms

MDE	Model-driven software engineering
FSM	Finite State Machine
EFSM	Extended Finite State Machine
ASM	Abstract State Machine
SDL	ITU-T Specification and Definition Language
TASTE	The ASSERT Set of Tools for Engineering
ASN.1	Abstract Syntax Notation One
MSC	Message Sequence Chart
API	Application Programming Interface

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

State machines (also referred to as state diagrams, state charts etc.) are a very popular and convenient notation for expressing the logic of many kinds of reactive systems that have some state dependent nature.

The transition to model-based development is motivated both by lower costs for overall software development and by the enhanced ability to find defects early in the design cycle. [1]

Different state machine dialects range from classical finite-state automata (FSA) and extended finite-state machines (EFSM) to richer and more practically oriented formalisms,such as the UML state diagrams, StateCharts, Stateflow or the state machines in SDL.

While at the highest level all of these dialects share some common concepts, there are significant differences in the core semantics, as well as in detail, such as the supported constructs in the guard and action language.

State machine diagrams are used in the process of system (or more narrowly, software) engineering at its various stages (specification, design, implementation, verification). To decide an appropriate state machine dialect in a specific context, a good understanding of the exact semantic differences between dialects becomes handy.

Sometimes, it can be necessary to migrate from one dialect to another, either due to the different level of expressiveness needed at the different development steps or due to the different background and context of the parties involved.

1.1.1 State Machine

In general, a state machine is any device that stores the status of something at a given time and can operate on input to change the status and/or cause an action or output to take place for any given change. [32]

1.1.2 FSM

Finite-State Machine is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some external inputs; the change from one state to another is called a transition.[37] It is defined by: list of its states, Initial state and conditions for each transition. “There are several different notations we can use to capture the behavior of finite-state machines:

- As a functional program mapping one list into another.
- As a restricted imperative program, reading input a single character at a time and producing output a single character at a time.
- As a feedback system.

  Representation of functions as a table

  Representation of functions by a directed labeled graph”[44]

1.1.3 EFSM

In Extended Finite-State Machine a transition can be expressed by an “if statement” consisting of a set of trigger conditions [46]. An Extended Finite State Machine M is a 5-tuple {S, I, O, D, T} where

- S is a set of states,
- I is a set of inputs,
- O is a set of outputs,
- D is an n-dimensional linear space $D_1 \times \ldots \times D_n$,
- T is a transition relation, $T : S \times D \times I \rightarrow S \times D \times O$ [45].

In addition to FSM, EFSM can have - guards (the if condition of the transition), data operation (such as assignments, calculation), and an n-dimensional linear space. It is sometimes explained in three blocks, FSA block, arithmetic block and transition evaluation block [46].

1.1.4 Harel’s State Machines

Harel enriched the idea of state machine by adding composite and parallel states and history. “Our diagrams, which we call statecharts, extend conventional state-transition diagrams with essentially three elements, dealing, respectively, with the notions of hierarchy, concurrency and communication.” [10] As summarized in [6], Harel’s state charts, as well as UML state charts, allow one to decompose a large state space into a set of composite states, each of them containing a subset of the original state set.

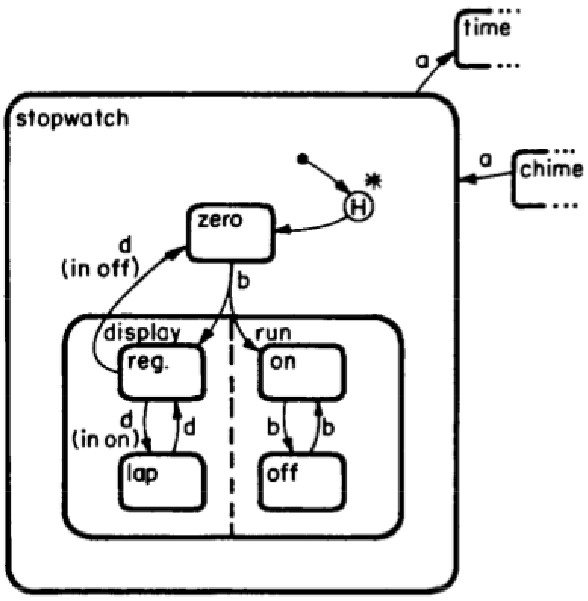


Figure 1: Harel's State Machine [10]

1.1.5 UML State Machines

“In UML, each class has an optional state machine that describes the behaviour of its instances (the objects). This state machine receives events from the environment and reacts to them. The reactions include sending new events to other objects and executing internal methods on the object.” [11] UML state machine editors (such as YAKINDU state chart Tools), allows to create state machines that are conceptually very similar to Harel’s state machine.

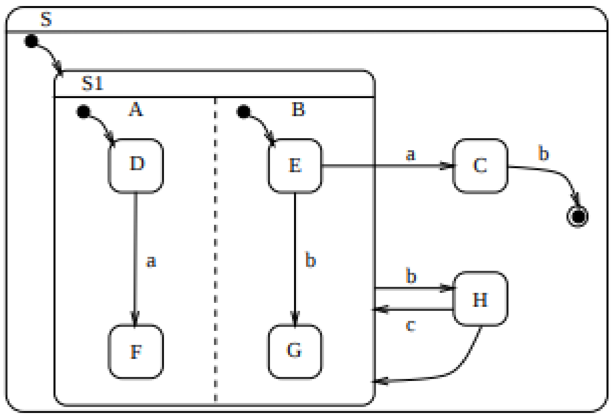


Figure 2: UML State Machine

1.1.6 ASM

Abstract State Machines (ASM) combine declarative concepts of first-order logic with the abstract operational view of distributed transition systems. ASMs are based on many-sorted first-order structures, called states [5]. In simpler word ASM provides the visualization with logics that makes a conceptual state machine easy to understand.

In the original conception of ASMs, a single agent executes a program in a sequence of steps, possibly interacting with its environment. This notion was extended to capture distributed computations, in which multiple agents execute their programs concurrently [39]. “Starting from an initial state, the agents perform concurrent computations and interact through shared locations of the state. The behaviour of ASM agents is determined by ASM programs, consisting of ASM rules.” [5] ASM updates when transitions triggers. A transition triggers based of rules that are defined by a guard or combination of guards. “These update sets define state transitions that result from applying all updates simultaneously.”[5] On transition ASM performs computations as well.

1.1.7 SDL

The Specification and Description Language (SDL) is a specification language targeted at the unambiguous specification and description of the behavior of reactive and distributed systems. For SDL-2000, Abstract State Machines (ASMs) have been chosen as the underlying formalism[6]. In SDL systems are described by blocks that are connected through channels - carrying messages. The messages/signals are passed from other agents or environment. The reception of a signal may cause a state transition. Transitions may contain actions that output signals to other agents or environment. Messages are treated in FIFO order. SDL supports:

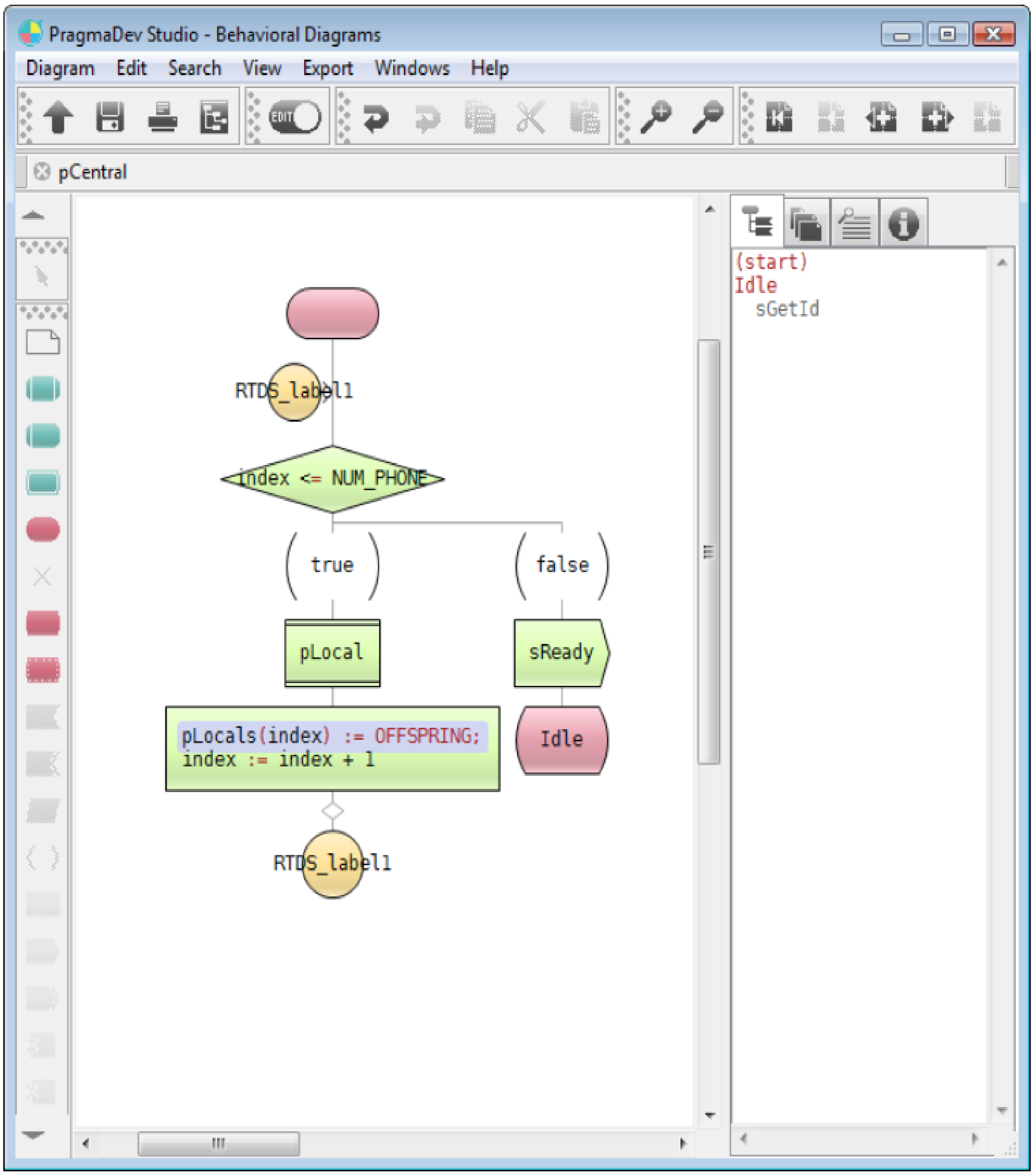
- Predefined data types (boolean, integer, natural, real, character, charstring, duration, time)
- Struct, choice, array, string, bag

- New data types based on existing data types and having additional operators and constraints
- Constants
- Variables
- Read only remote variables (variables of other processes)
- Procedures (inside block or procedure, accepts parameter and return value to caller, can send and receive messages, can modify data of declaring agent)
- Macros (can't receive messages, no transition, can’t call itself)
- Composite states (sub-states)
- Timers and manipulation of timers
- Dynamic process creation
- Classes (of block or processes, connected with gates), subclasses

SDL does not support global variables. But, constants can be declared anywhere.

1.1.8 Available Tools for Handling SDL State Machines

One of the available SDL state machine modeling tool is PragmaDev Studio. It is a complete, large scale tool that comes with commercial support [42]. It integrates tools for system architects, developers and testers. “PragmaDev has established partnership with key players in the real time domain. Customers include Airbus, Renault, Nokia, ST, ABB, the French Army, the European Space Agency, Toshiba, Korean Telecom, or LG Electronics”. [38] Unfortunately, it requires to install some 32bit linux libraries that does not ship with Ubuntu 16.04 LTS by default.



```
pLocals(index) := OFFSPRING;
index := index + 1
```

Figure 3: PragmaDev Studio SDL State Machine Editor [33]

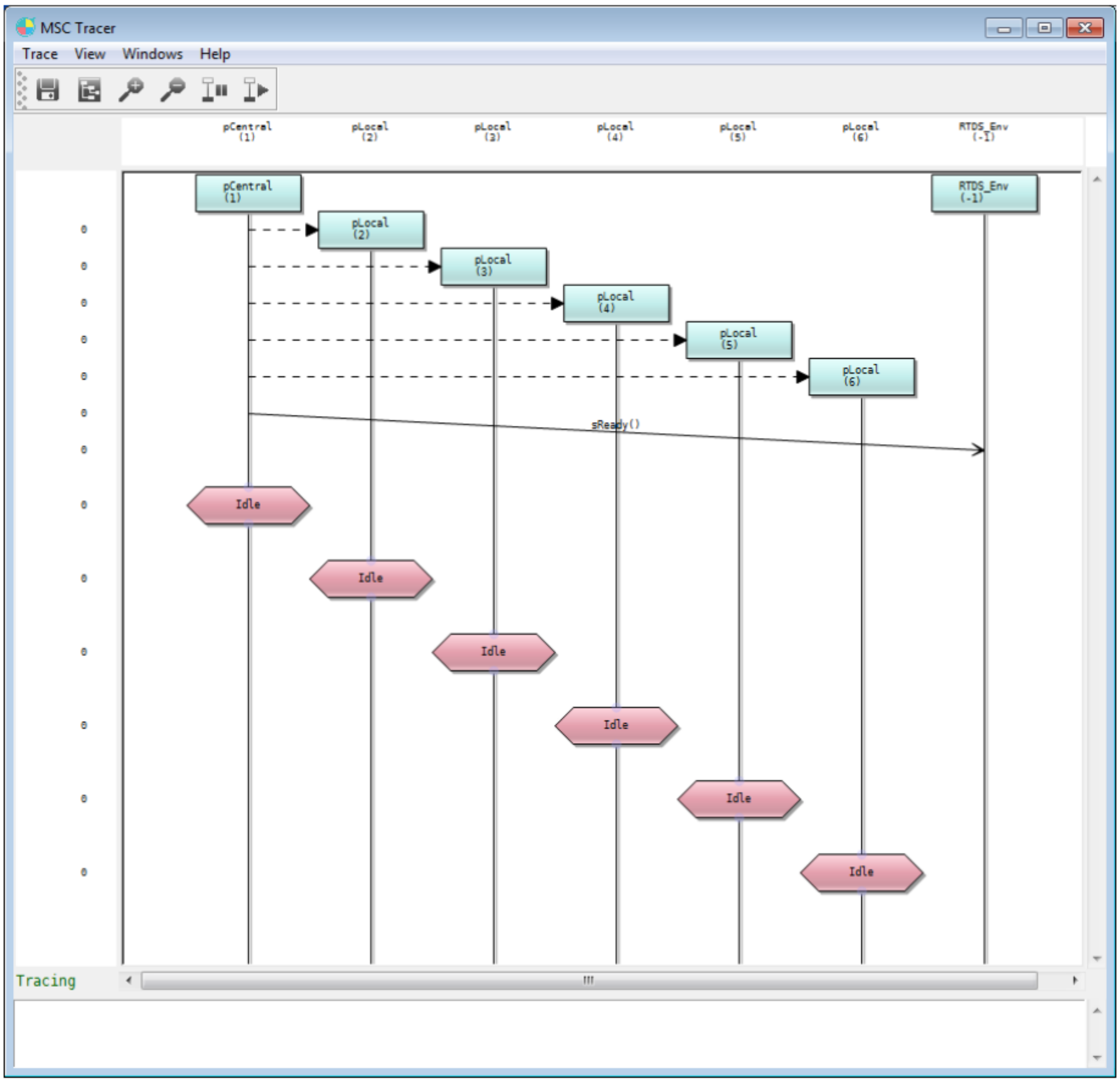


Figure 4: Simulation in PragmaDev [33]

“Use the Freemium version that comes with all the features but that is restricted in size: 50Kb per file and 200Kb per project (this includes all diagrams except the MSC, declaration files such as SDL-PR and ASN.1 files, and TTCN-3 source files).” [34] Regular price is euro 290 per month [34]. It is a limitation that creates a necessity to search for a free solution.

There exists also an open source SDL editor called OpenGEODE [42]. It is a tool that was originally developed for the TASTE suite [40], but can also be used standalone. TASTE allows to create and simulate interactive modules. Fortunately, it is a open source product and free.

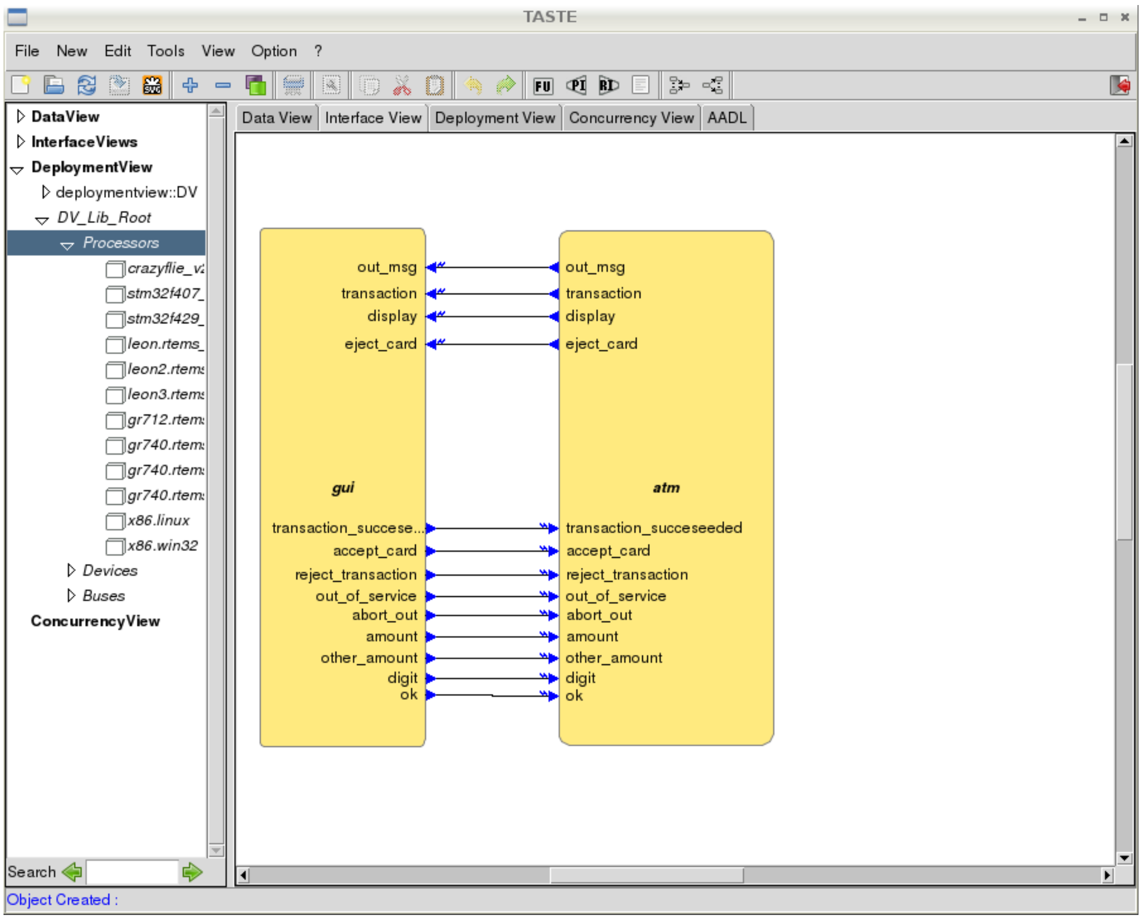
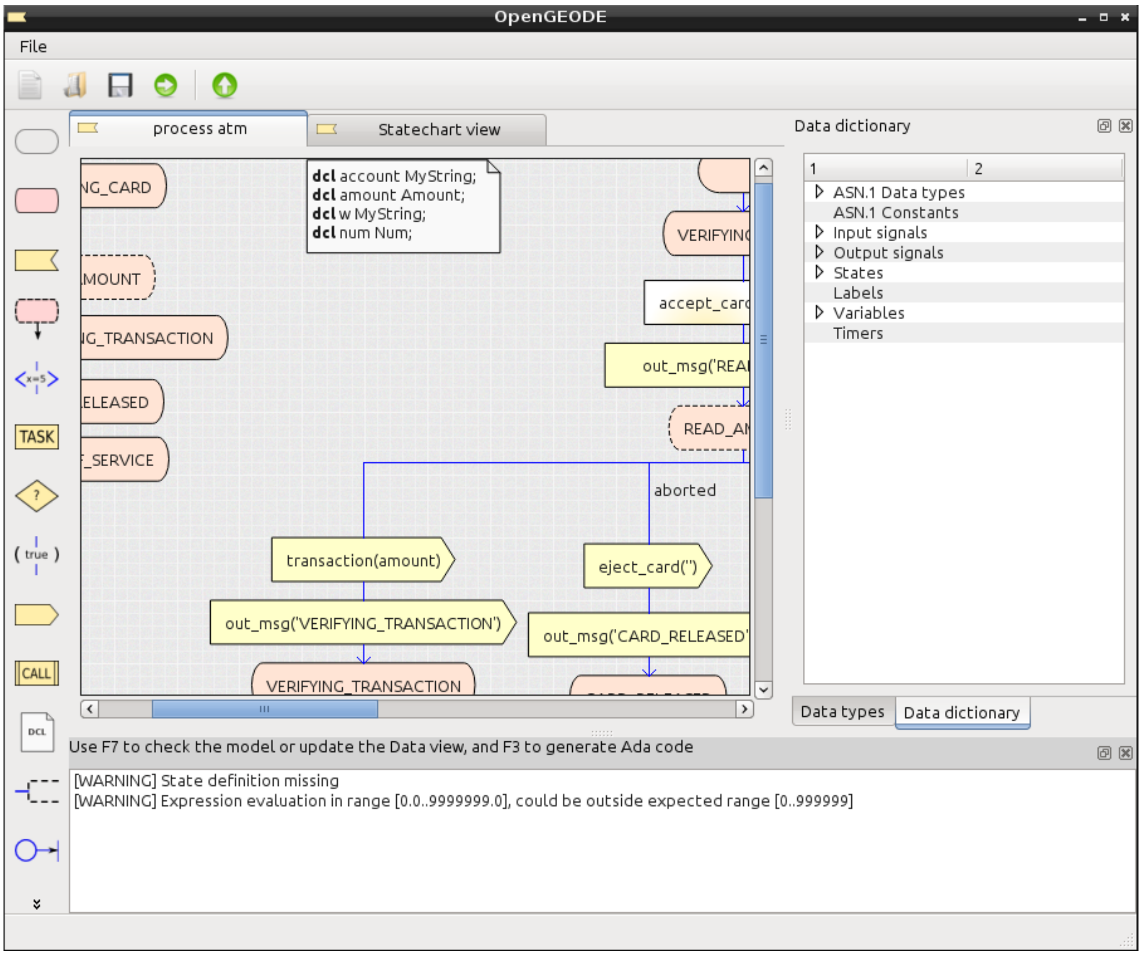


Figure 5: TASTE Inteface View

“TASTE is a set of freely-available tools dedicated to the development of embedded, real-time systems. It is developed by the European Space Agency together with a set of partners from the space industry.” [40] “TASTE also comes with two built-in (free) SDL editors that allow graphical description of state machines, and automatic code generation.” [40] After considering all these benefits, TASTE is the tool of choice for this study.



```
dcl account MyString;
dcl amount Amount;
dcl w MyString;
dcl num Num;
```

```
Use F7 to check the model or update the Data view, and F3 to generate Ada code
[WARNING] State definition missing
[WARNING] Expression evaluation in range [0.0..9999999.0], could be outside expected range [0..999999]
```

Figure 6: SDL Editor OpenGEODE

1.1.9 Stateflow

Stateflow is a state machine dialect. It is used in the popular MATLAB/Simulink tool suite. “Stateflow lets you combine graphical and tabular representations, including state transition diagrams, flow charts, state transition tables, and truth tables, to model how your system reacts to events, time-based conditions, and external input signals.” [12] It is widely used in the industry and education. Stateflow combines state machines and flow-diagrams diagrams to a unique and expressive formalism. Subsets of the Stateflow language have been-formalized by independent researchers, e.g. in [43] and [47].

1.1.10 Problematic Example of Transformation

There is a chance of getting results by conversion process that does not work, the converted model may not show the same behavior as the original model. Such an example is explored by Danial et al in their work “Polyglot Modeling and Analysis for

Multiple Statechart Formalisms" [1]. They have experimented if properties hold on converting Stateflow and UML (Rhapsody) state charts into a Java. Relational Rhapsody is a modeling environment based on UML. They modeled two users to access shared resources. The communication between the Arbiter and the users is modeled using Simulink signals [1]. A user may request or cancel a resource; the arbiter may grant or deny the resource, and it can also rescind the resource after it has been granted [1]. The arbiter prevents potential conflicts between resource requests coming from different users and it enforces priorities. For example, it does not make sense to start a communication session with Earth while the rover is driving [1].

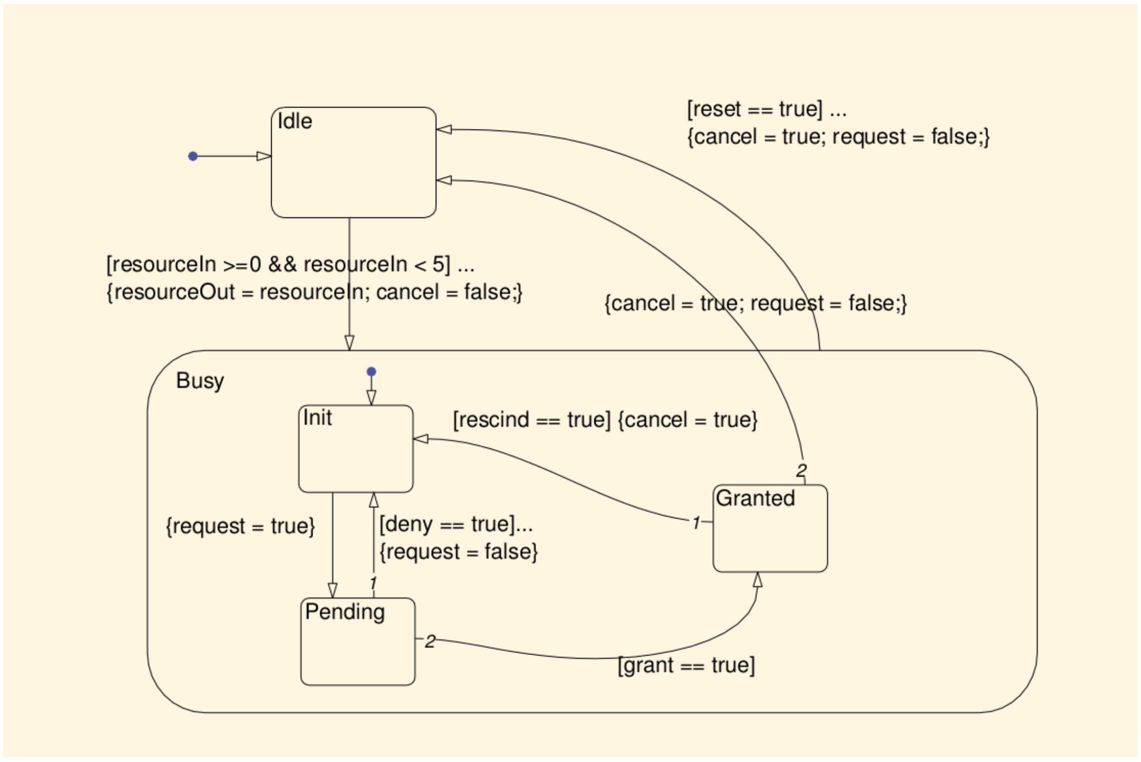

Idle
[reset == true] ...
{cancel = true; request = false;}
[resourceIn >=0 && resourceIn < 5] ...
{resourceOut = resourceIn; cancel = false;}
{cancel = true; request = false;}
Busy
Init
[rescind == true] {cancel = true}
2
Granted
1
{request = true}
[deny == true]...
{request = false}
1
Pending
2
[grant == true]

Figure 7: State machine for user [1]

The result shows that property holds when both users are uses Stateflow semantics, but property fails when one user uses UML. The reason why this property fails in the UML case while it holds in the Stateflow case is that outer transitions (e.g. see the transition enabled on reset==true from Busy back to Idle) have higher priority over inner transitions in Stateflow, but have lower priority in UML Rhapsody. [1]

1.2 Motivation

State machines can be designed using different tools such as Stateflow, OpenGEODE, PragmaDev Studio, YAKINDU Statechart Tools, Rational Rhapsody. Once a state machine is done, later it can be reused in another project. It can be reused as a small part of new state machine or as a starter tool, depending on relevance. But, it is not possible when an existing state machine is designed with the tools that use a modeling language different from new project. To make this possible a proper conversion tool is required. This study is aimed for making this tool for two practical state machine dialects: Stateflow and SDL.

1.3 Goal

The current study should compare the semantics of the Stateflow and the SDL modeling languages and identify the subset of either language that can be converted to the other language in an unambiguous and clear way without altering the semantics.

Secondly, provide guidelines for creating a tool for automatic bidirectional conversions between the two dialects, so that one could automatically convert models (that fall into the above subset) to the other language and vice versa.

1.4 Objectives

1. To specify mappings for a subset of the modeling languages
2. To provide guidelines for making an automatic conversion tool

1.5 Research Design

Objective	Steps	Outcome
Semantic mapping	Study the semantics of the above modeling languages	Literature review report
	Familiarize with the respective modeling tools	
	Perform a literature review and familiarize oneself with existing research in the field	
	According to the findings choose/adapt an existing approach or define a mapping yourself, if necessary.	Semantic mapping specification for a subset of the above modeling languages
		Set of models in either language demonstrating the mapping in concrete cases

Objective	Steps	Outcome
Find an optimistic way for Automated translations	Propose data structure to store both SDL and Stateflow model that facilitate conversions flow.	Data structure
	Give an algorithm to Implement the semantic mapping rules identified in first objective using a chosen model transformation approach.	Pseudo code
	Suggest an optimistic approach to go for coding.	Programming languages, API documentation, demo code

Table 1: Research Objectives

1.6 Validation

- Running converted and original state machines side by side and comparing their behavior.
- Comparing MSC's to find behavioral similarity.

1.7 Related Work

Dony et al in their paper "Programming Language Inter-conversion" presented a new approach of programming languages inter-conversion which can be applied to all types of programming languages. The idea is about implementation of the intermediate language for inter-conversion. [7] They also said "Hence code conversion becomes even more challenging because the features of the source language need to be somehow simulated into the destination language. Hence, this imposes a limitation on code conversion." [7]

Sendall et al in their paper "Model transformation: The heart and soul of model-driven software development." [9] state that "A transformation is typically only meaningfully applied against certain model configurations. Thus, it would be desirable in many cases to describe the conditions under which the transformation produces a meaningful result, which can then be enforced by a tool at execution time."

Rodionov in his thesis "Implementing TTÜ Nanosatellite Communication Protocol using TASTE Toolset", worked with SDL and indicated further usability of the state machine created there. "The created system can be used in the communication process of the TTÜ-Mektory Student Satellite both on ground and on the satellite. The result of this thesis will be used as a case study in ESA project that integrates TASTE Tool set with QGen.” [3] Here nothing mentioned about possible uses of the state chart in another formalism such as Stateflow.

Danial et al in their work “Polyglot Modeling and Analysis for Multiple State chart Formalisms”, wanted to compare various state charts. “To verify these safety-critical systems, a unified framework is needed based on a formal semantics that captures the variants of State-charts. We describe Polyglot, a unified framework for the analysis of models described using multiple State chart formalisms. In this framework, State chart models are translated into Java and analyzed using pluggable semantics for different variants operating in a polymorphic execution environment.” [1] This work supposed to consider available conversion tools, but their report does not signal any existence of such a tool.

Czarnecki and Helsen state in [8] that “While there exist some well-established standards for modeling platform models, there is currently no matured foundation for specifying transformations between such models.”

Through these studies and through web exploration, nothing is found that can convert SDL to Stateflow state machine or vice versa.

1.7.1 Language Conversion Approach

The paper on “Programming Language Inter-conversion” by Dony et al discussed a conversion approach. They also performed a theoretical case study on the conversion of code written in the C++ programming language to Java. [7]

Code conversion should also preserve the structure or should modify it to make it even better by removing redundant codes. Hence, it will be better if a compiler is designed for this purpose. And the purpose of compilation is to convert the given program into its corresponding Intermediate-language. This Intermediate-language can be converted into any programming language using another compiler. [7]

All the programming languages have some common features such as logical, arithmetic operators, looping and so on. Based on these facts a new language can be defined having all these features to represent the characteristics of the programming language. [7] - Dony et al explained in their writing. Their main idea is show in the next figure.

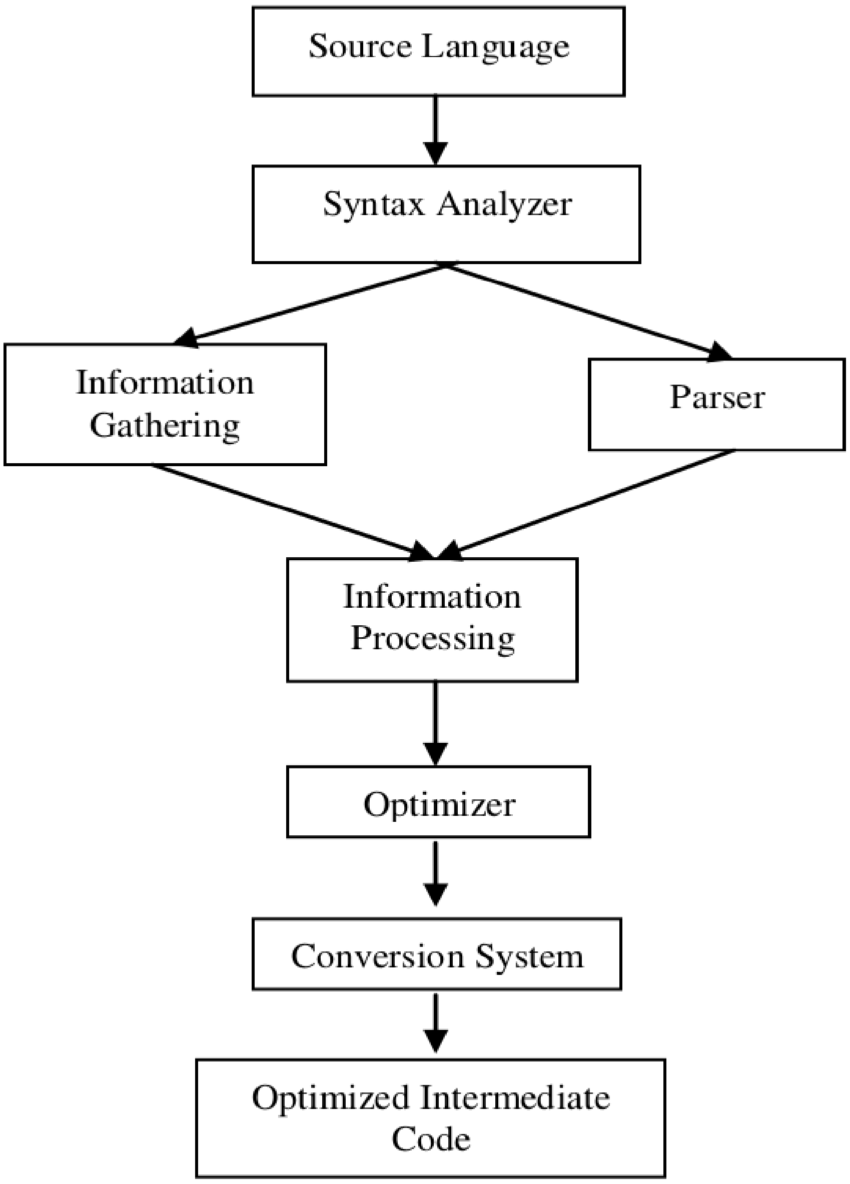
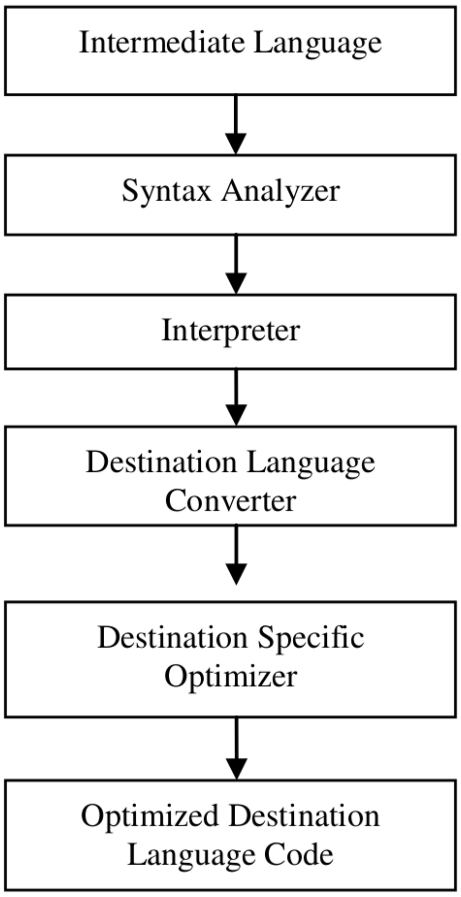


Figure 8: Source to Intermediate Code [7]



Intermediate Language
Syntax Analyzer
Interpreter
Destination Language Converter
Destination Specific Optimizer
Optimized Destination Language Code

Figure 9: Intermediate Code to Source [7]

Their work was done kind of successfully, but with limitation such as “Pointers in C++ cannot be completely represented in Java but can be approximated by using references, operator”, “Multiple Inheritances: This is impossible to achieve. The only option is to use interfaces.” The conclusion is very interesting - “Achieving the maximum efficiency of conversion without compromising the quality of converted system is the programmers' dream. Even though language conversion might seem to be easy, it is actually a Herculean task with many different complications.”[7]

Despite of some limitation, this method helps to formulate the approach for the transformation between state machine dialects.

2 Mapping

The semantics of SDL continuous signals and continuous time Stateflow were not investigated in this study, because the study was primarily oriented for model development for embedded systems that more typically operate in a discrete time manner.

2.1 Variable

In SDL one can use either SDL predefined data types or the ASN.1 formalism for declaring data types. OpenGEODE only supports ASN.1 type definitions. In ASN.1 all numeric data types get an upper limit and lower limit of allowed values at the time of declaration. In Stateflow range can be limited by assigning minimum and maximum value. While converting Stateflow to SDL, the range of the data type itself is to be assigned. SDL has no global variable. But it is not touching the scope of this study. For a state machine, read and writes are done on containing block variables. Variables of other blocks are not relevant here.

2.2 State Machine Initialization

SDL requires an unnamed entry point with a transition to initial state. Also there has to be a separate state with same identifier as initial state that will work as state definition. Stateflow requires just an initial (default) transition to initial state for inception of a state machine. If a Stateflow state machine has events, then it needs the option "Execute (enter) Chart At Initialization" to be enabled. It is explained in the "Transition" section below.

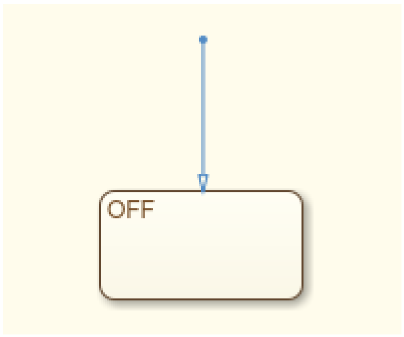
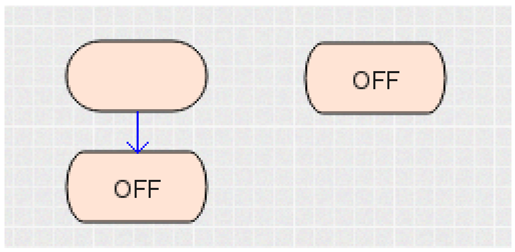
Stateflow	**SDL**
	

Table 2: Initial Transition - Stateflow vs SDL

2.3 State

A minimal state looks similar in both Stateflow and SDL. But Stateflow state can have entry, during and exit actions. Composite and Parallel state is possible in both case. They are explained below under the headings “Action”, “Composite State” and “Parallel State”.

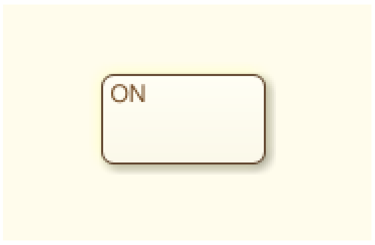
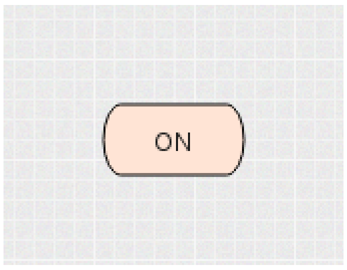
Stateflow	SDL
	

Table 3: Basic State - Stateflow vs SDL

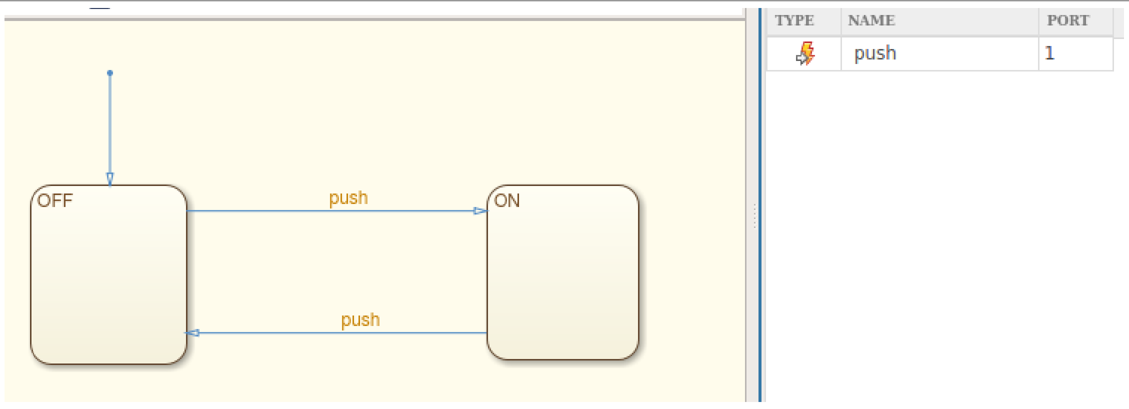
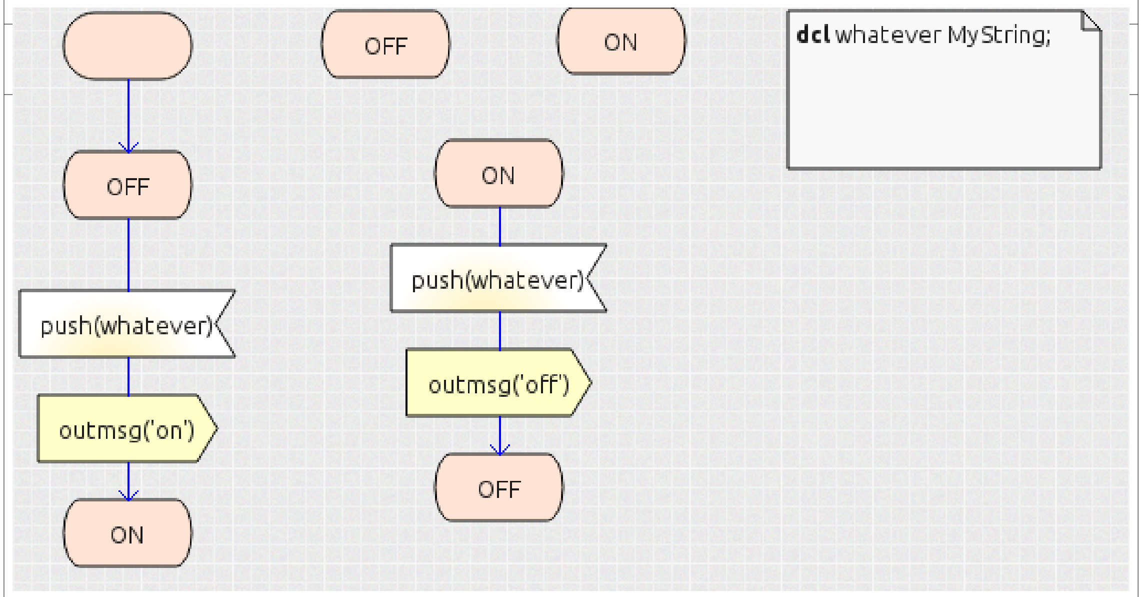
2.4 Transition

The most contrasting element is transition. SDL requires an incoming message signal to enable transition. Stateflow uses guard or event to trigger a transition as well as an event can be triggered without anything (guard, event). A Stateflow chart can be activated either by event or input data. An event based Stateflow state machine stay put until it receives an event. When it receives the first event, it triggers the initial transition. This event should not be consumed by the initial transition while converting SDL to Stateflow. Because, it is for a transition other than initial one. In this case “Execute (enter) Chart At Initialization” option is to be enabled. By default, the first time a chart wakes up, it executes the default transition paths [41].

SDL message signal can contain more than one parameters. In Stateflow message can trigger transition but it can contain just one parameter [23]. Therefore, SDL signal can not be converted to Stateflow message. But, Stateflow messages will be converted to SDL signal that will have one parameter only.

TASTE requires signals to contain at least one parameter. Since, we are simulating SDL models in TASTE, signal without any message can not be sent.

For converting SDL to stateflow, the message sending needs to be emulated using an input event. Say, we have a signal “push()” in SDL. It will be converted to an input event “push”. When the event will be called by outside agent, transition will be triggered. Event in Stateflow does not carry any argument, therefore, the arguments have to be converted to input variables.

Stateflow



TYPE	NAME	PORT
	push	1

```
dcl whatever MyString;
```

Table 4: Transition - Stateflow vs SDL

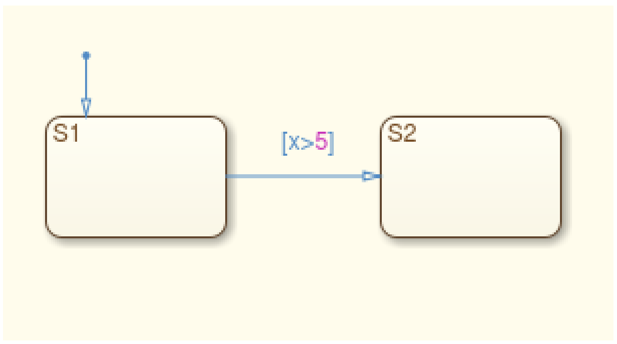
2.4.1 Transition Order

Each transition in Stateflow has a value for transition execution order. SDL does not store such value and it does not have this concept. SDL has decision branches. These branches are mutually exclusive, overlapping conditions are not allowed. Such as if a decision has two branches with “x>5” and “ELSE”, then another branch “x=6” can not be added, because last one is overlapping with first one. But, Stateflow can have two transition with “x>5” and “x=6” sourcing from same state. Therefore, conversion of Stateflow transition order into SDL decision branch require implementation of complex algorithm. This study is not digging that much.

2.5 Guard

2.5.1 Stateflow Guard

To convert a Stateflow guard to SDL, firstly have to convert the input variables to the parameters of of incoming signal. An input signal will be a created to intake all the input variables, as “get_data(x)” in the example. There will be an extra transition per state machine that will start from initial state and go to initial state. It will be triggered by the input variable intake signal. Secondly, we have to create a input signal to check the value of the variable, as “check(whatever)” in the example. Because, dislike Stateflow, SDL can not trigger transition without input signal. Finally, have to place “decision” as a replacement of guard condition. Each transition with a variable will have a “decision” branch of this variable.

Stateflow to SDL
Stateflow


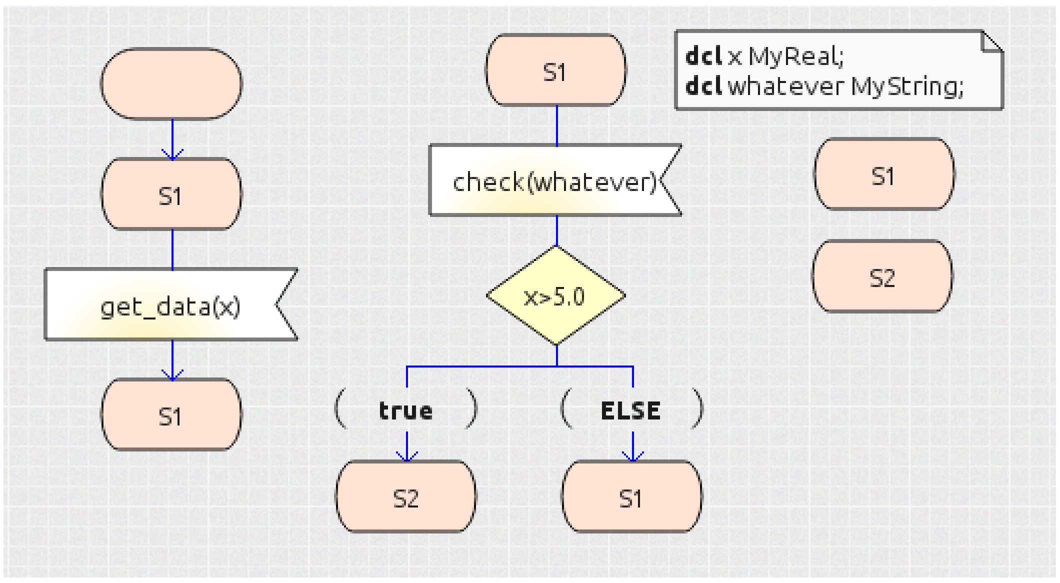
SDL
 S1 get_data(x) S1 S1 check(whatever) x>5.0 (true) (ELSE) S2 S1 dcl x MyReal; dcl whatever MyString; S1 S2

Table 5: Conversion of Stateflow Guard to SDL

2.5.2 SDL Decision of Signal Parameter

For the other direction, SDL to stateflow, first part of a transition, input signal will be converted to input event and value checking (of the parameter of the signal itself) will be converted to a guard that checks the value of the corresponding input variable.

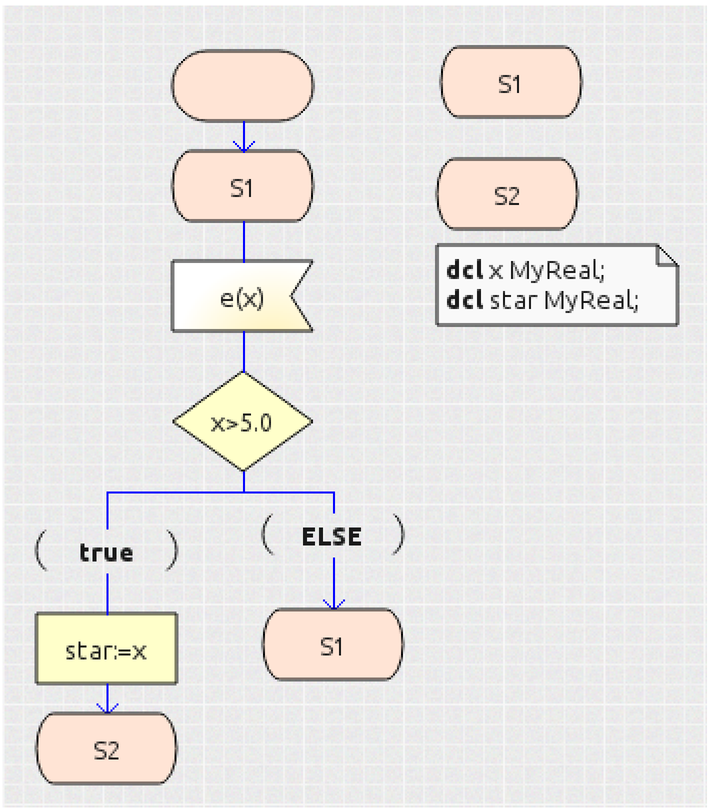
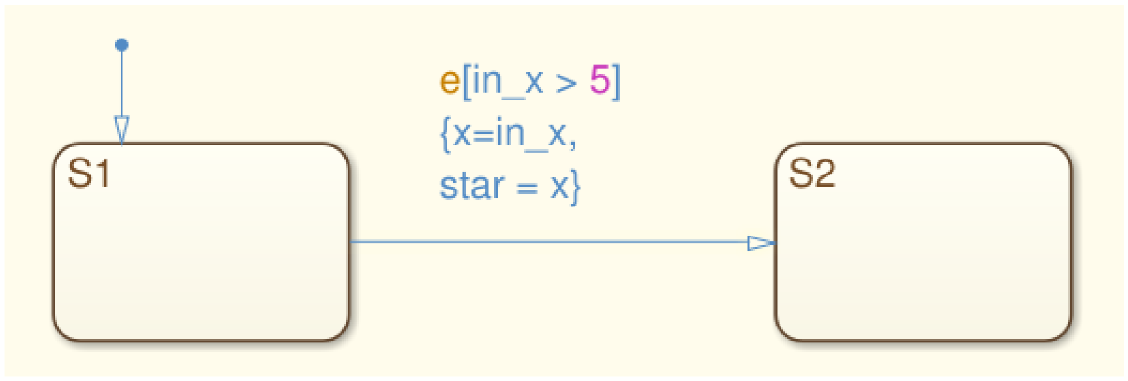
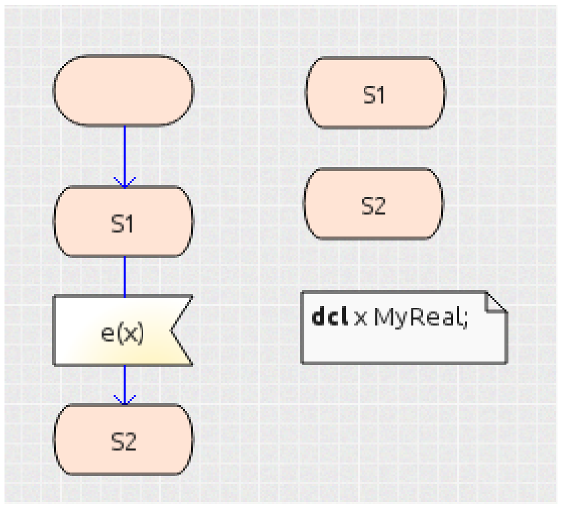
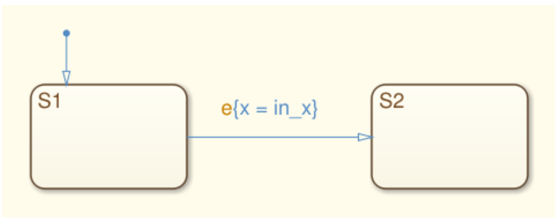
SDL to Stateflow
SDL
 S1 S2 S1 e(x) x>5.0 true ELSE star:=x S1 S2 dcl x MyReal; dcl star MyReal;
Stateflow
 e[in_x > 5] {x=in_x, star = x}
in_x is a double type input variable. x is a double type local variable. star is output variable.

Table 6: Conversion of SDL Guard to Stateflow

Each branch of the decision symbol in SDL will be converted to a transition in Stateflow. ELSE in SDL covers all other possible values. Covering all possible values is compulsory in SDL, otherwise SDL will give error. There can be many branches along with ELSE branch in SDL. For converting ELSE, a transition will be used that will have the highest execution order. Because, it has to be triggered only when all other guard conditions are false. But, when the destination of this ELSE transition and the source are same state, it will not be converted to Stateflow transition. In this case number of transition in Stateflow will be number of decision branches minus one. Secondly, the value must be saved because in Stateflow value of input variable can change before occurrence of next event. This is not possible in SDL. For the sake of intuitiveness, local variable name will be same as SDL signal parameter name but the input variable will be in_[variable name]. Local variable will be used for all further read and write purpose until an event brings new value. When an event brings new value, the local variable will be written and get continued.

2.5.3 SDL Signal

When in SDL transition is not checking the value of the argument, no need check the value of the corresponding input variable in Stateflow. But the value must be saved because in Stateflow value of input variable can change before occurrence of next event that is not possible in SDL.

SDL to Stateflow	
SDL	**Stateflow**
	
	x is a double type local variable.

	in_x is a double type input variable.

Table 7: Conversion of SDL Event Based Transition to Stateflow

2.5.4 Stateflow Event and Parameterless SDL Signal

This is the simplest case, similar to transition conversion, a Stateflow transition with just an event will be converted to just a SDL signal. Same for SDL to Stateflow, a signal without any parameter will be converted to an event.

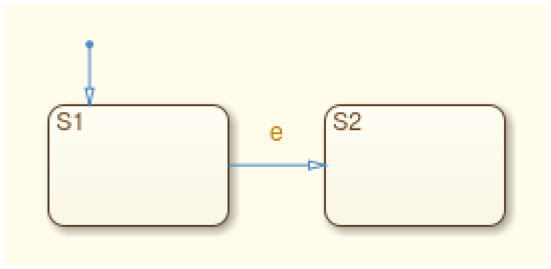
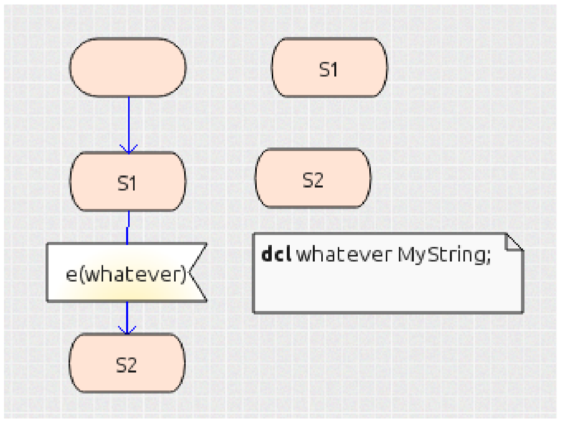
Bidirectional	
Stateflow	**SDL**
	 S1 S1 S2 e(whatever) S2 dcl whatever MyString;
	Since, TASTE does not work with signal without parameter, “whatever” used as a parameter placeholder, e(whatever) instead of e().

Table 8: Bidirectional Conversion of Parameterless Event Triggered Transition

2.6 Output Signal/Event

SDL output signal is equivalent to Stateflow output event when it has not parameter. In SDL it can have any number of parameter. This parameters will be converted to output parameters.

2.7 Messages

Stateflow message is very similar to SDL signal, but it can have only one parameter. So, Stateflow receiving message will be converted to SDL input signal with a parameter and Stateflow message sending will be converted to SDL output signal with a parameter.

2.8 Action/Task

2.8.1 Entry Action

A Stateflow state can have assignment/task on entry of that state. This entry actions will be converted to SDL task and will be placed just before the state. It sounds wrong, because assignment will be done first, then the state will be reached. But in SDL nothing can happen other than going into connecting state. Therefore, it will work.

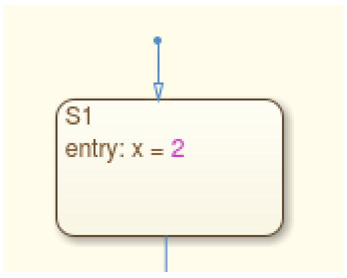
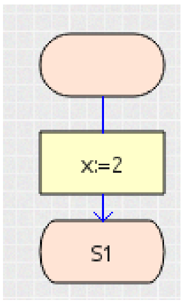
Stateflow to SDL	
	

Table 9: Conversion of Stateflow Entry Action to SDL

2.8.2 During Action

During actions are executed when a state is active, an event occurs, and no valid transition to another state is available [35]. A During action will be converted to SDL task and will be placed just before state. This state will not have any outgoing transition. Since, the task is just before the state nothing can happen between the task and state other than getting in the state. Therefore, it will give same behavior.

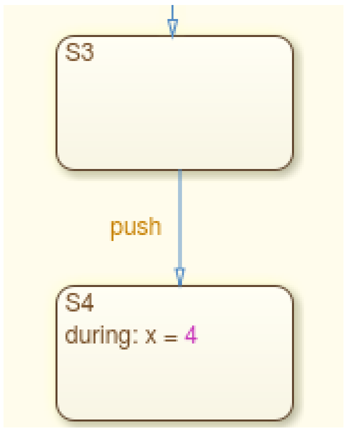
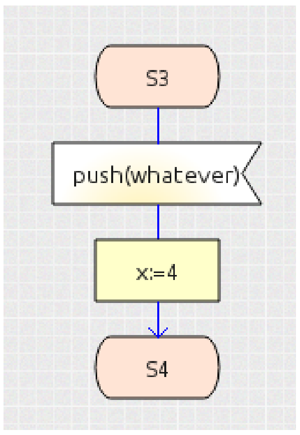
Stateflow to SDL	
	

Table 10: Conversion of Stateflow During Action to SDL

2.8.3 Exit Action

Exit actions will be converted to SDL task and will be placed just after input signal connected to the state. On getting an incoming message state can be exited, so exit actions will be placed just after catching the incoming message. It has to be done for each of the state exiting input signal.

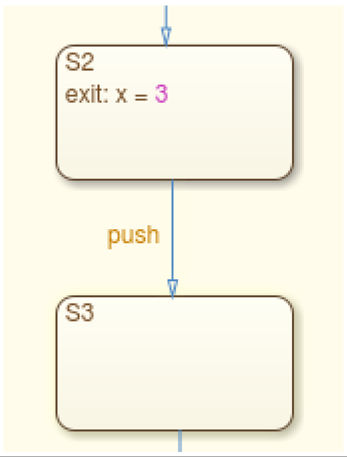
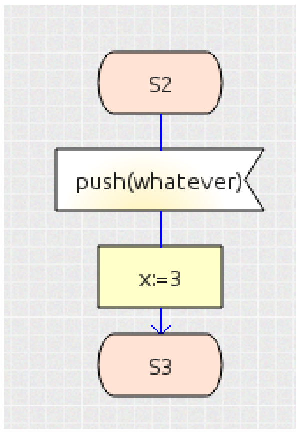
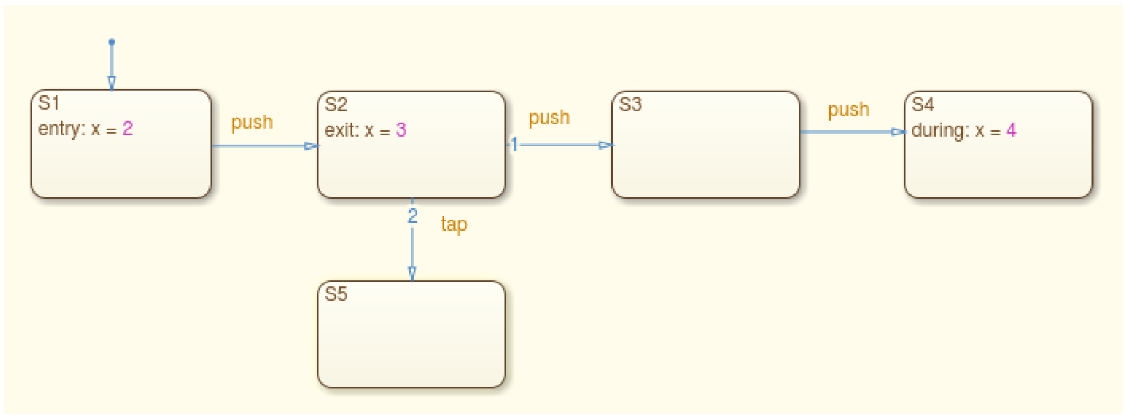
Stateflow to SDL	
Stateflow	**SDL**
	

Table 11: Conversion of Stateflow Exit Action to SDL

For better understanding, a complete Stateflow state machine with Entry, During and Exit actions is converted to SDL state machine.

Stateflow
 S1 entry: x = 2 push S2 exit: x = 3 push 1 S3 push S4 during: x = 4 2 tap S5
SDL

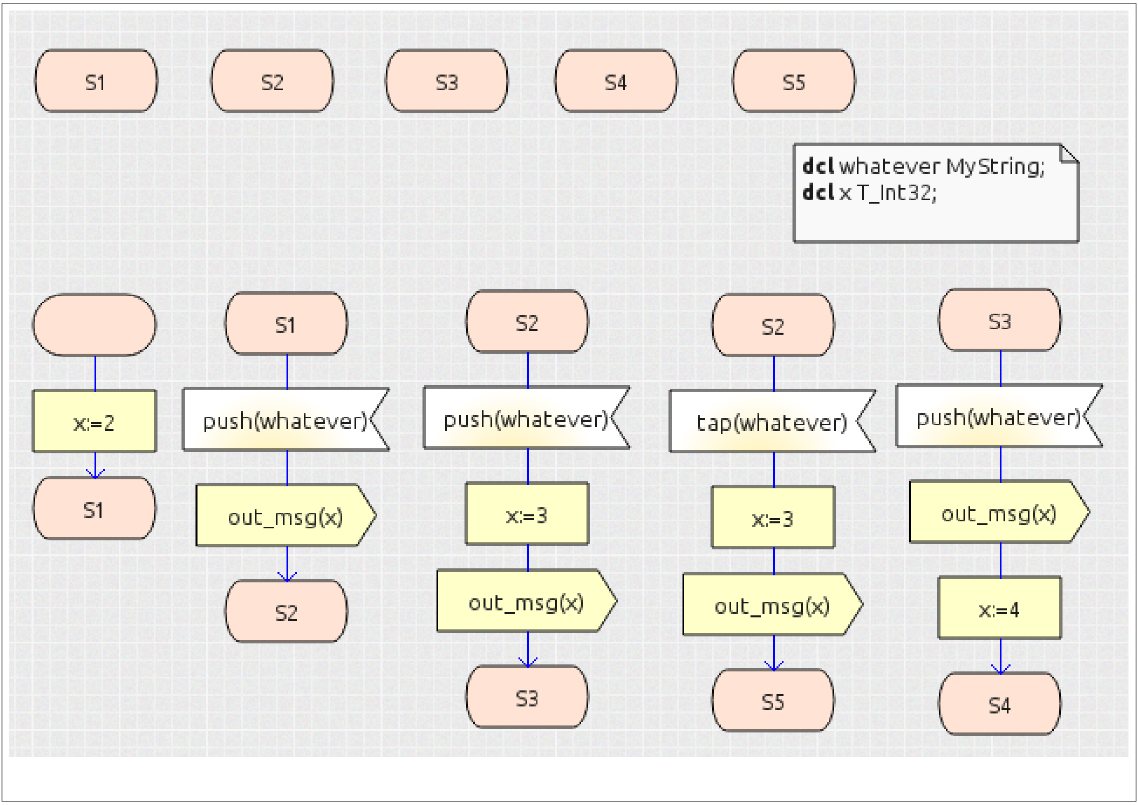


Table 12: Stateflow Actions Converted to SDL - Complete State Machine

2.8.4 Task

Tasks are placed in task symbol in SDL that is curly bracket in Stateflow.

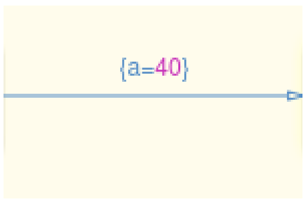
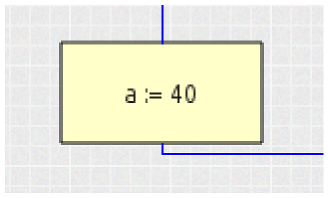
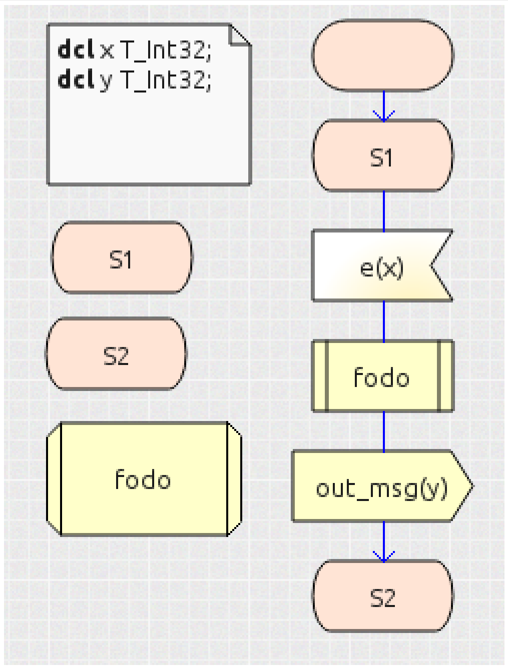
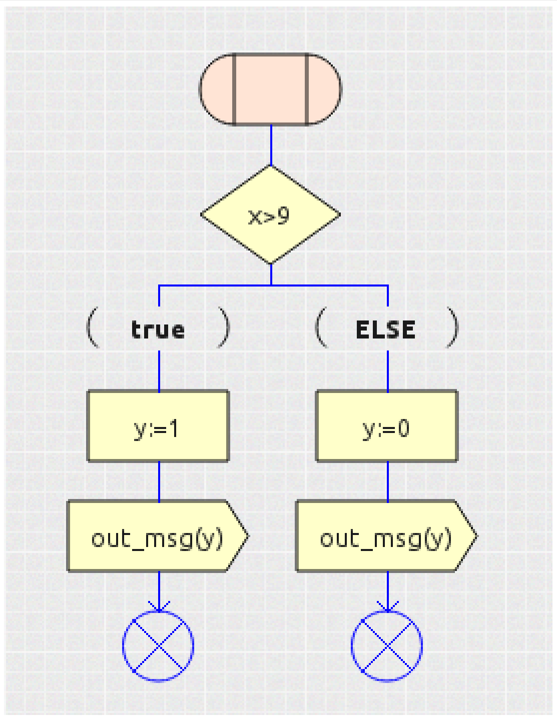
Bidirectional	
Stateflow	**SDL**
 {a=40}	 a := 40

Table 13: Bidirectional Conversion of Stateflow Assignment and SDL Task

2.9 Procedure

Procedure is used in SDL to group a sequence of instruction. The counterpart of procedure in Stateflow is Graphical function. These two contains mappable components. Some components are already discussed that are also available in a SDL procedure such as “Decision”, “Task” and mappings for this components will remain same. Some components are exclusively for SDL Procedure such as "ProcedureStart" maps to "Default transition", "ProcedureStop" maps to "Junction".

SDL	

 dcl x T_Int32;
dcl y T_Int32;
S1
S2
fodo
S1
e(x)
fodo
out_msg(y)
S2
	 x>9
	(true)
	(ELSE)
	y:=1
	y:=0
	out_msg(y)
	out_msg(y)

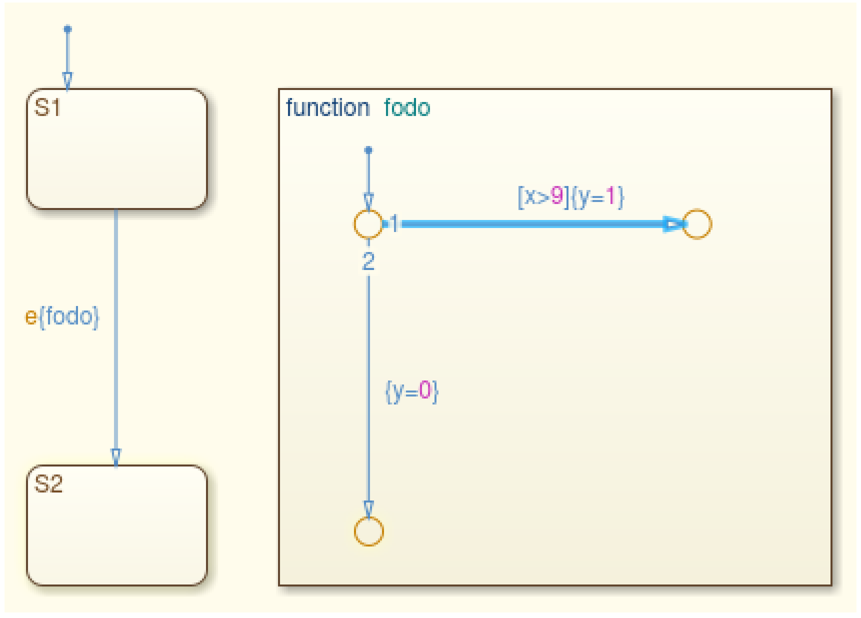
Stateflow	
 S1 e{fodo} S2 function fodo [x>9]{y=1} 1 2 {y=0}	

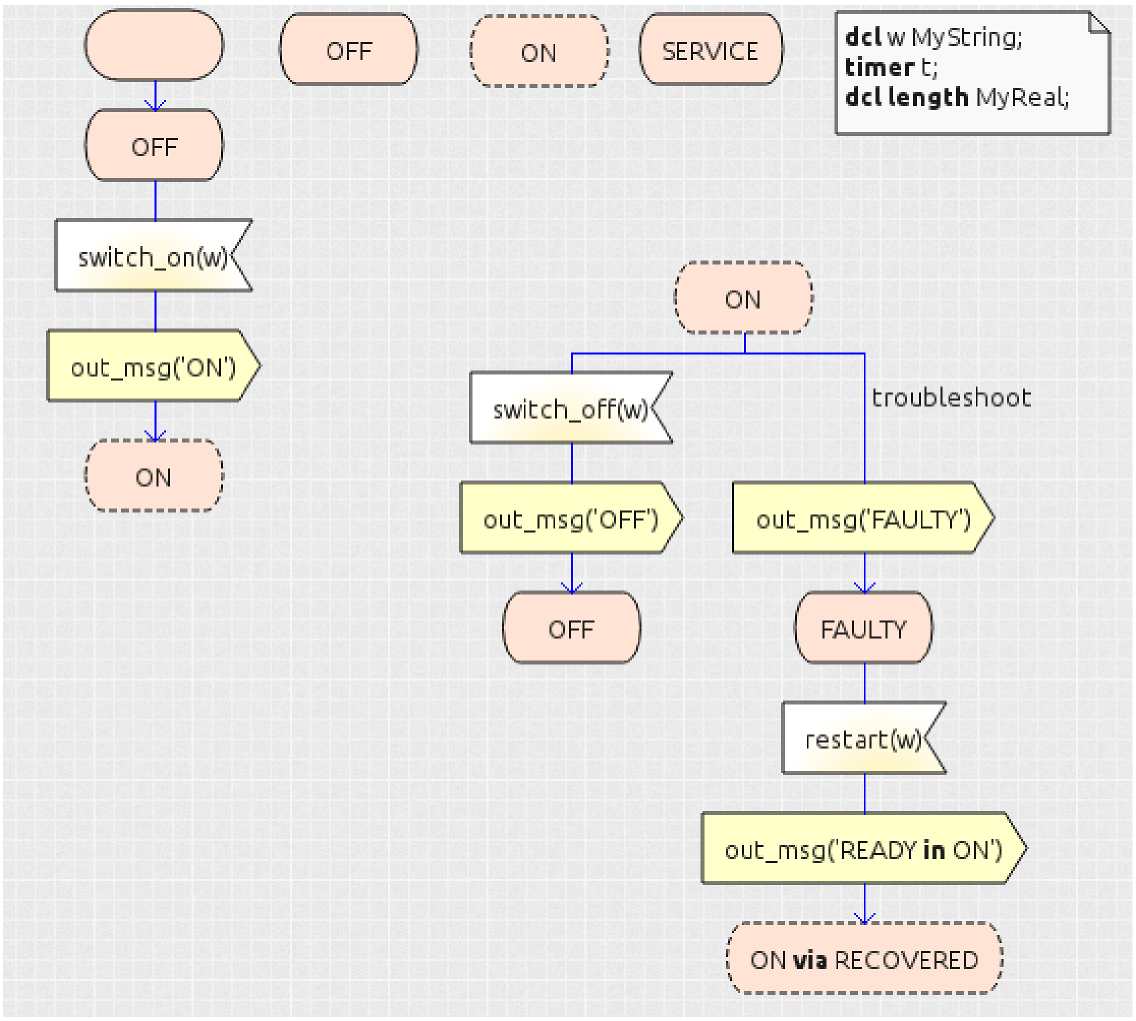
Table 14: SDL Procedure and Stateflow Graphical Function

2.10 Procedure Call

Almost similar to task, procedure calls are placed in ProcedureCall symbol in SDL that is curly bracket in Stateflow.

2.11 Composite State

Variables declared in parent is available in sub-state in SDL. Pointing to an exit point using return statement (exit symbol), for at least once is compulsory for SDL nested state. It is also the way to define a transition from inner state to outer state. Exit symbol without a label means the default exit to outer state. To make a transition from outer state to inner state, an entry point is required. This entry point connects to the destination state. Transition from outer state points to this entry point (using "via" keyword). Entry point without label works as initialization.



```
dcl w MyString;
timer t;
dcl length MyReal;
```

Figure 10: First Level of SDL State Machine With Nested State

Composite state use “Connect” that allows inner to outer transition in SDL. In this example “troubleshoot” is the connect statement that is used on “fail” signal to move from “RECORDING” state to “FAULTY” state. Stateflow it is direct and simpler to connect inner state to outer state.

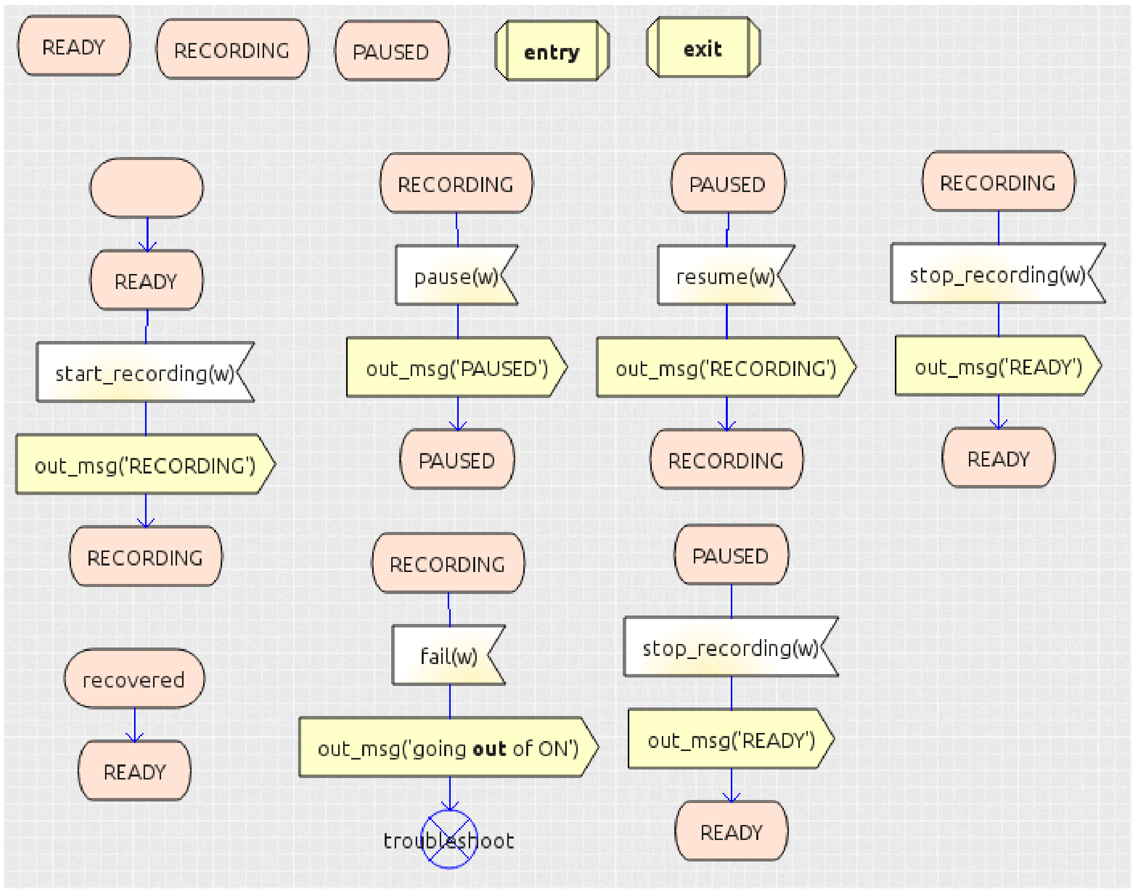


Figure 11: Nested State "ON" of The SDL State Machine Above

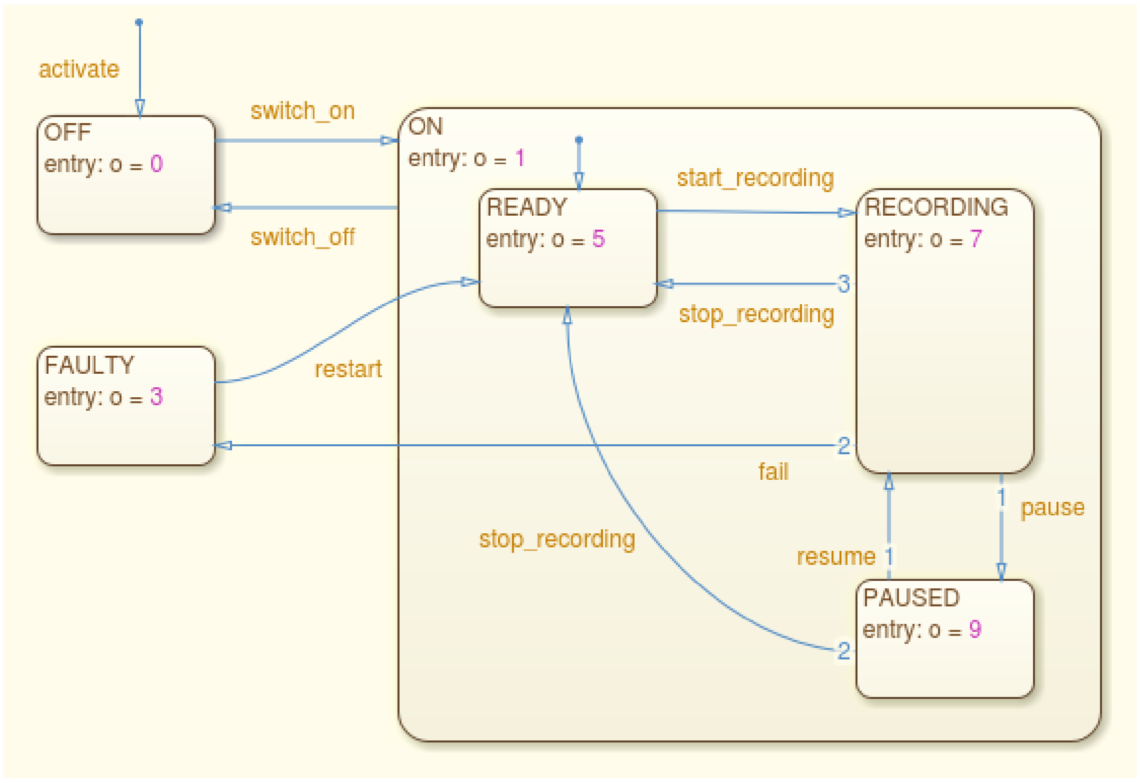


Figure 12: Same State Machine in Stateflow

2.12 Parallel State

In Stateflow parallel state is defined by selecting decomposition of an state to “AND” that is by default “OR”. In SDL parallel state is just two or more nested state inside a nested state without initial transition.

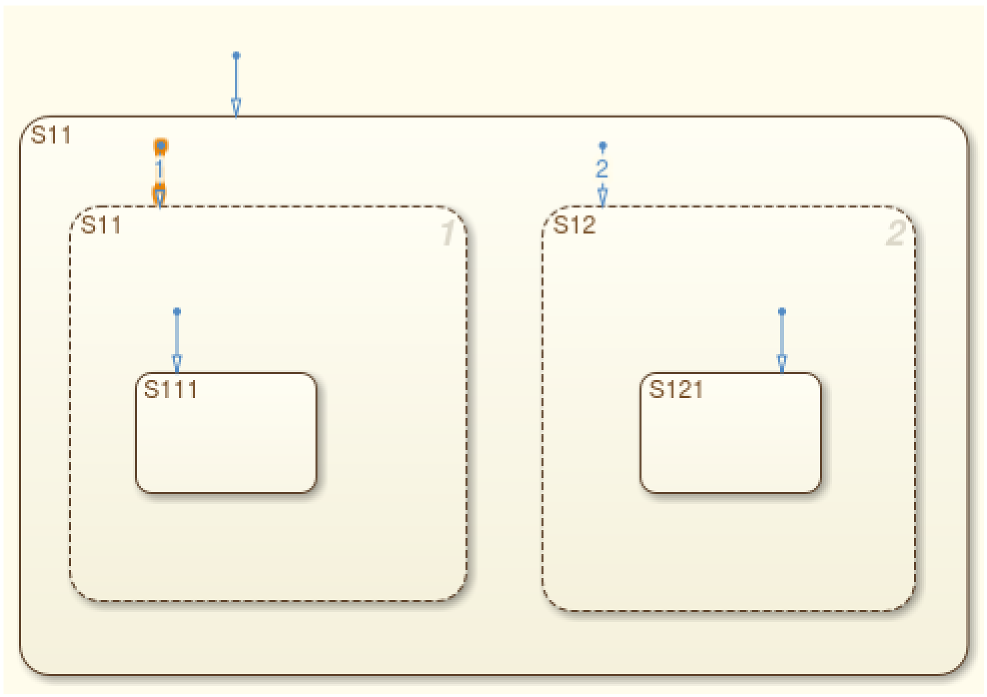


Figure 13: Stateflow Parallel States

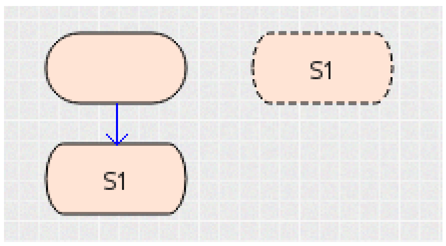
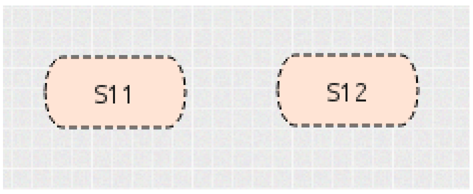
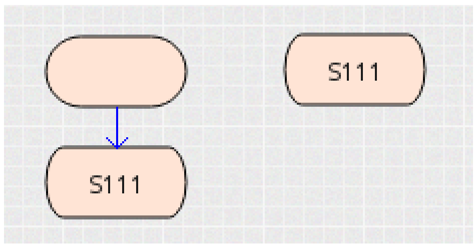
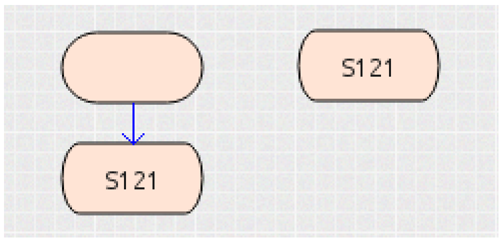
	
	

Table 15: SDL Parallel States

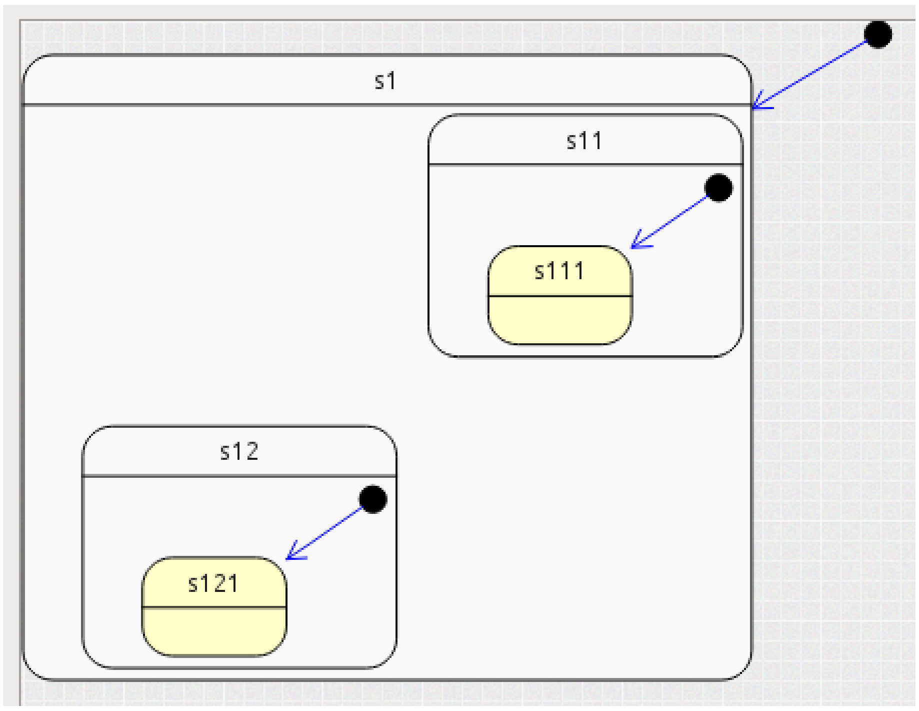


Figure 14: SDL State Chart View of Parallel States

2.13 Temporal Logic

2.13.1 After

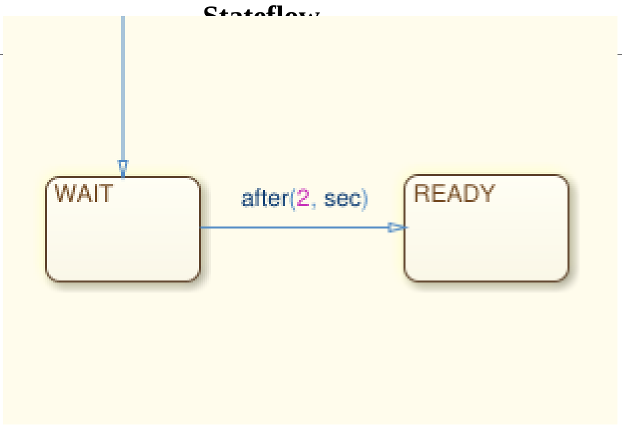
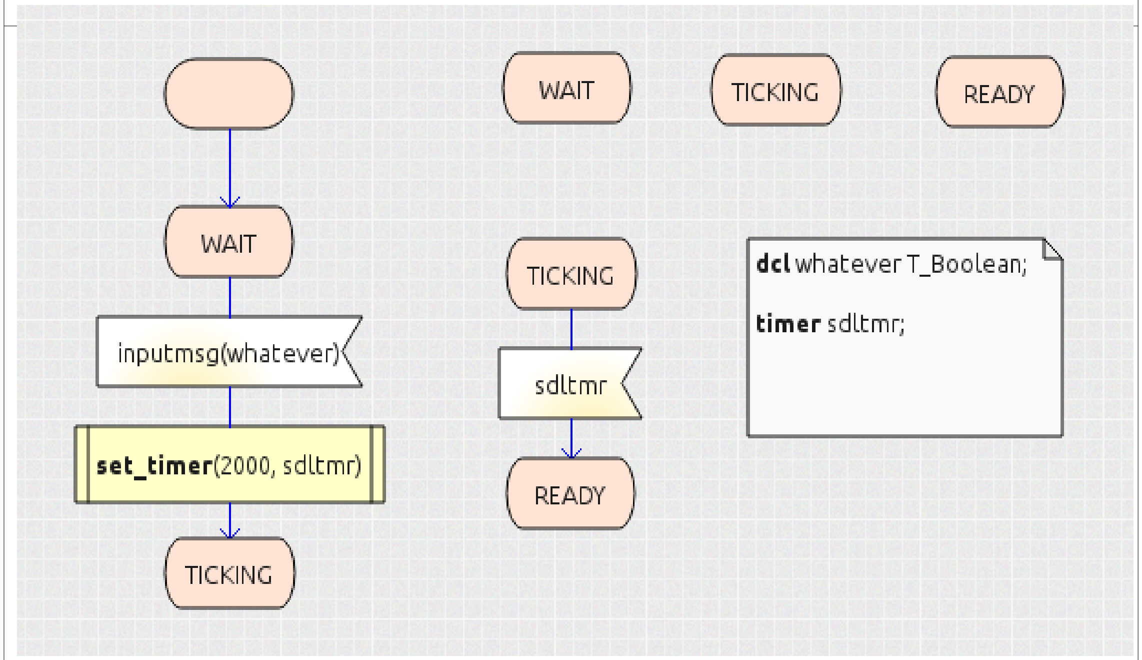
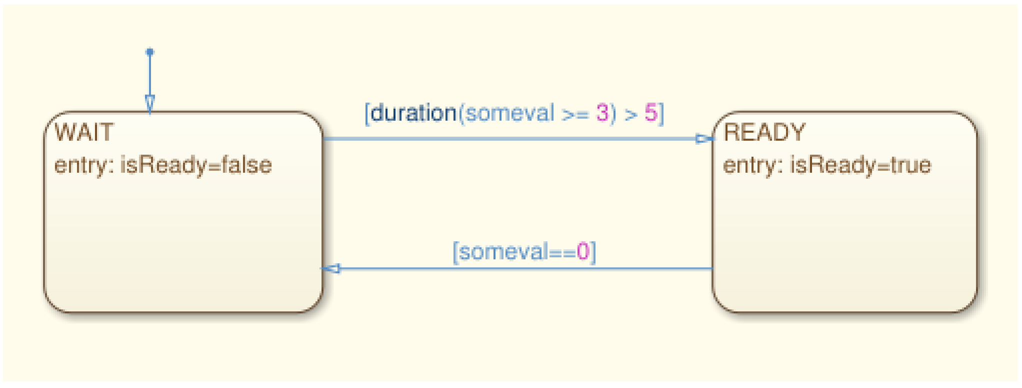
Bidirectional
Stateflow

SDL
 WAIT TICKING READY WAIT inputmsg(whatever) set_timer(2000, sdltmr) TICKING TICKING sdltmr READY **dcl** whatever T_Boolean; **timer** sdltmr;
In stateflow, if WAIT state has other outgoing transition that can trigger during this 2 seconds then ticking state has to have that transition as well. This case is not considered in this mapping.

Figure 15: Conversion of Stateflow Temoral Logic After to SDL

2.13.2 Duration

Stateflow



SDL

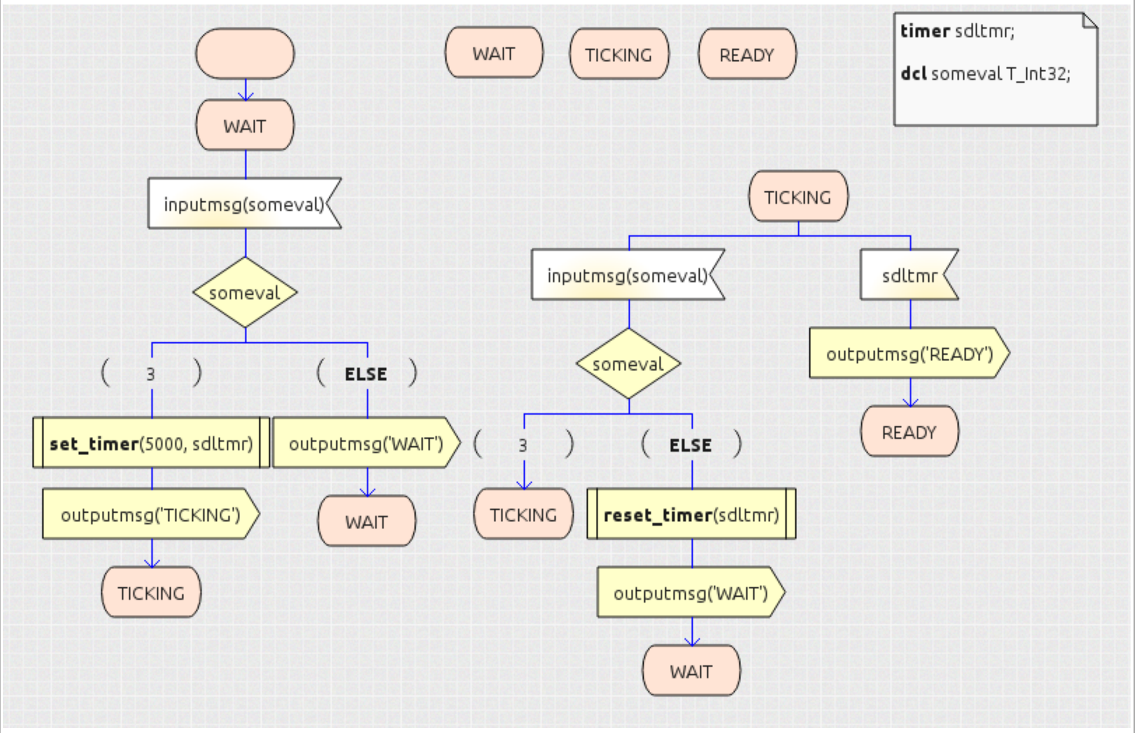


Figure 16: Conversion of Stateflow Temoral Logic Duration to SDL

2.13.3 Timer

SDL timer is a simple timer that can be set and reset. It is a service available in SDL design and simulation package such as TASTE. To convert this timer to Stateflow an external timer service has to be used. Also easy to create a timer state machine to help getting value and making guard on this value.

2.14 Limitations

All the discussion so far is about what is possible. But easy to bump on a state machine that is way too difficult to convert. Secondly, TASTE has some limitation. For example -

- If a Stateflow transition triggered by both variable value guard and an event then it a bit difficult to convert.
- SDL can have parameterless signal but TASTE does not work with parameterless signal, it requires at least one parameter for each interface.
- SDL signal can have more than one parameter but TASTE does not work with more than one parameter for sporadic interface that generates the SDL input signals for OpenGEODE.
- In entry and exit procedure of nested state “output” signal does not work with TASTE.
- Output signal between entry point (both named and unnamed) and connecting state does not work with TASTE.
- History concept is not available in TASTE. History means saving status of a composite state and resume from this saved status next time this composite state get activated. In Stateflow it is done by the symbol “H” with a circle around. This feature in not available in OpenGEODE, although in SDL a similar concept called “save” exists.

3 Validation of Mappings

Behavioral comparison of the converted state machines and the original state machines can testify the correctness of the mappings. First a SDL state machine of Automated Teller Machine (ATM) is converted to Stateflow. Then they are simulated and their behavior is compared to check if they behave same. This SDL machine is a slight extension of the state machine described by Birger et al in their work “Scalable and Object Oriented SDL State(chart)s.” [4]

Secondly, a Stateflow coffee machine is converted to SDL and checked for behavioral similarity.

3.1 SDL to Stateflow

The main behavior of this state machine is to represent a ATM. Once a card entered, First it verifies the card, then reads the amount to transact. If the amount is valid, it makes the transaction, otherwise shows “Limit exceeded” and take the amount again. Anytime, it aborted. Also it can become out of service anytime.

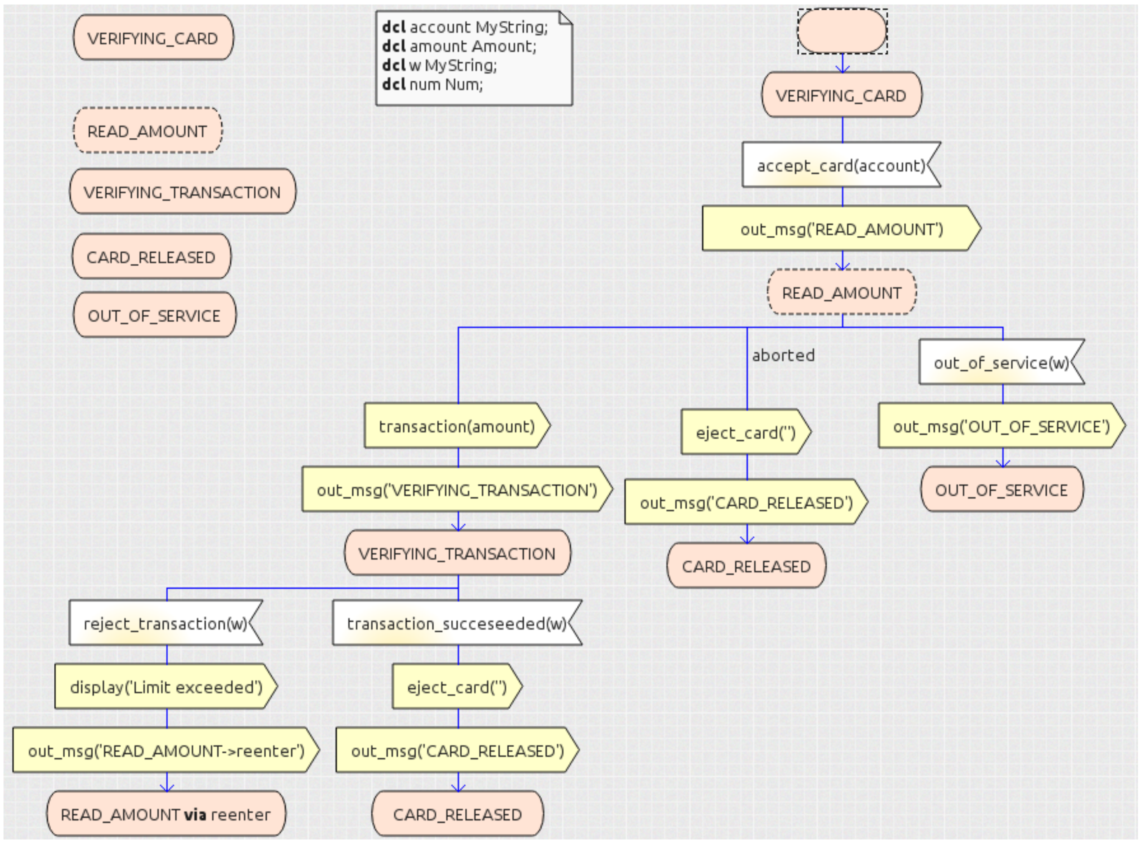


Figure 17: ATM SDL State Machine

It has a nested state (or composite state) READ_AMOUNT that encapsulates the logics for getting the amount to transact for cashing out from bank account.

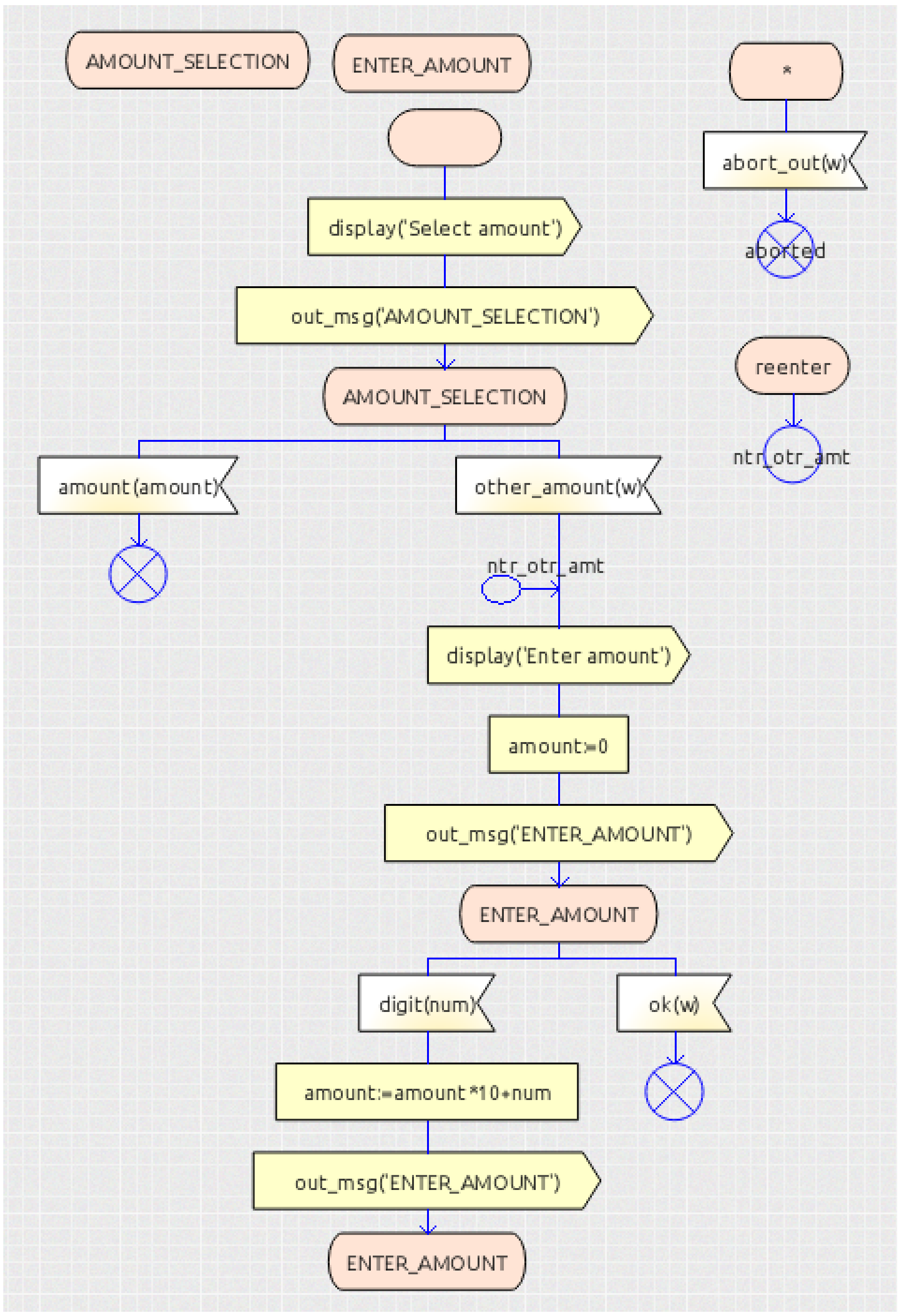


Figure 18: Nested State READ_AMOUNT of ATM SDL State Machine

It's behavior has been tested using Message Sequence Chart (MSC). Here is a happy path being discussed and others are placed in annex. This MSC shows that amount is taken three times, 5000, 500 and 50. For 5000 and 500 transaction rejected, finally for 50 transaction is successful. Amount 500 and 50 is entered using digit press and 5000 by amount button.

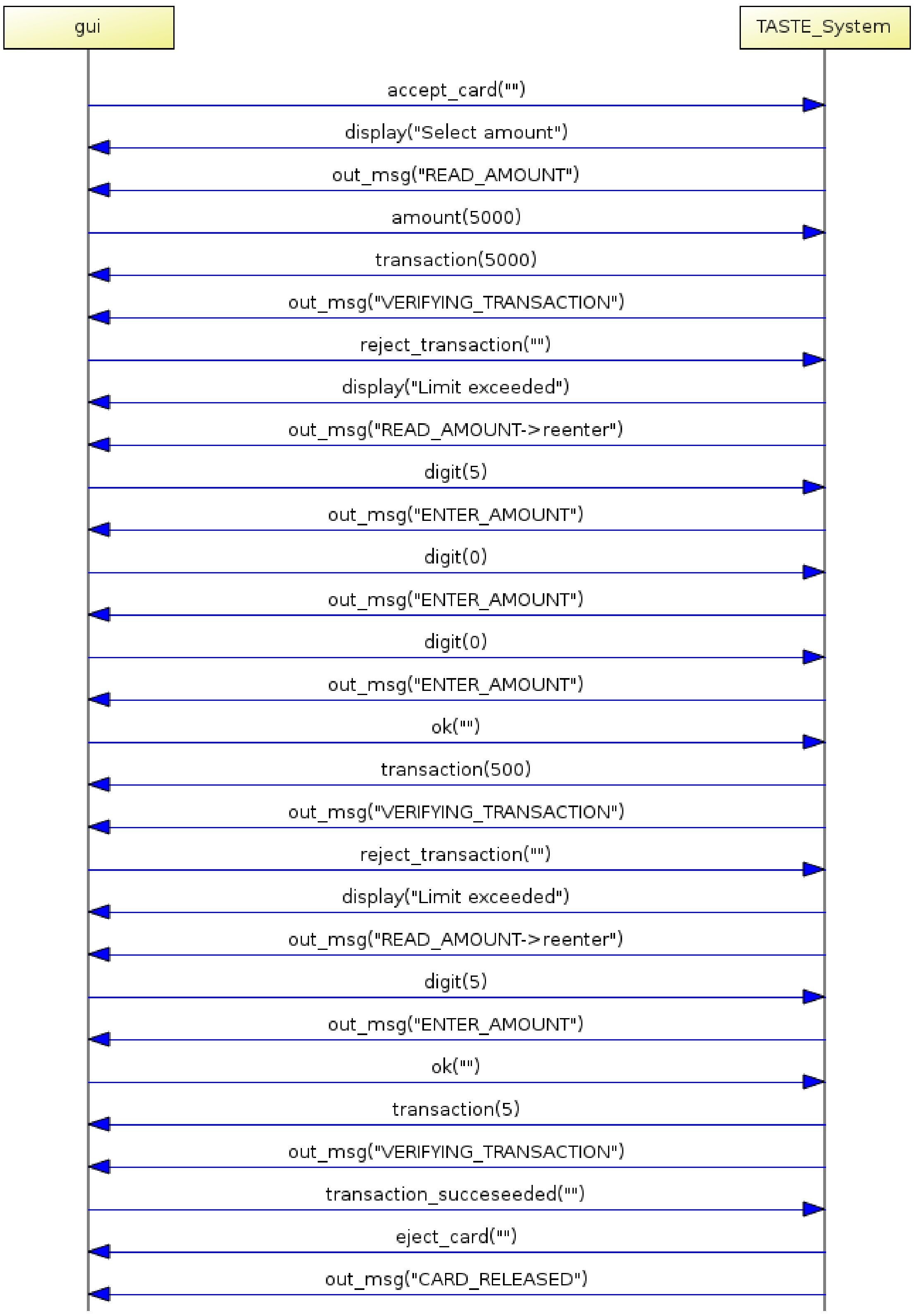


Figure 19: MSC of A Happy Path of ATM SDL State Machine

TASTE does not show output signal in MSC that is between entry point (named and unnamed) and immediate next state. Therefore, “ENTER_AMOUNT” after “reenter”, display(‘Select amount’) and out_msg(‘AMOUNT_SELECTION’) on initial transition are missing.

To convert this SDL state-machine to Stateflow, first the states are placed. Once states are in place including sub-states, it looks like getting the shape. Secondly, declaring the variables and events. An event for each signal and two variable for each input signal parameter. One of intaking data and the other one for saving data when the event triggers. But no variable to declare for output signal parameter, because they are using variable that is already available. Finally, converting the transitions by applying the mappings.

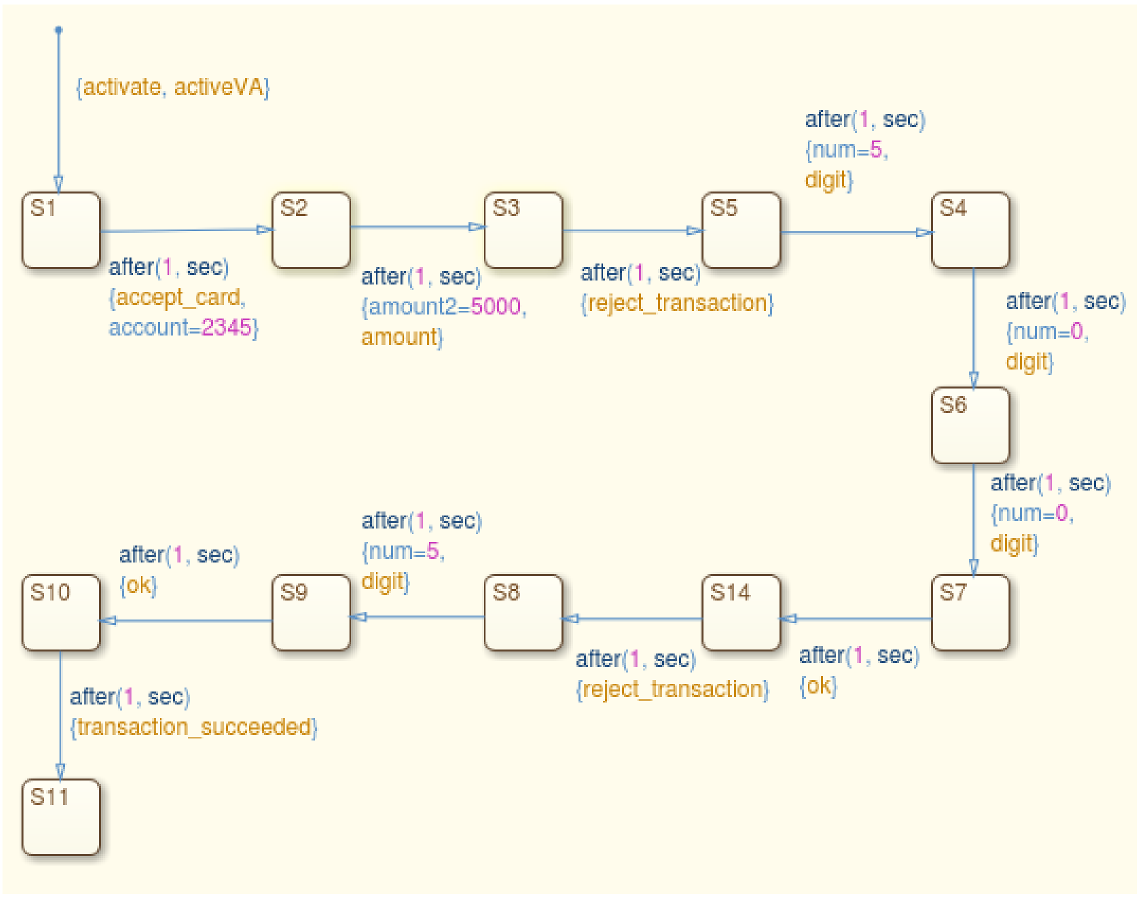
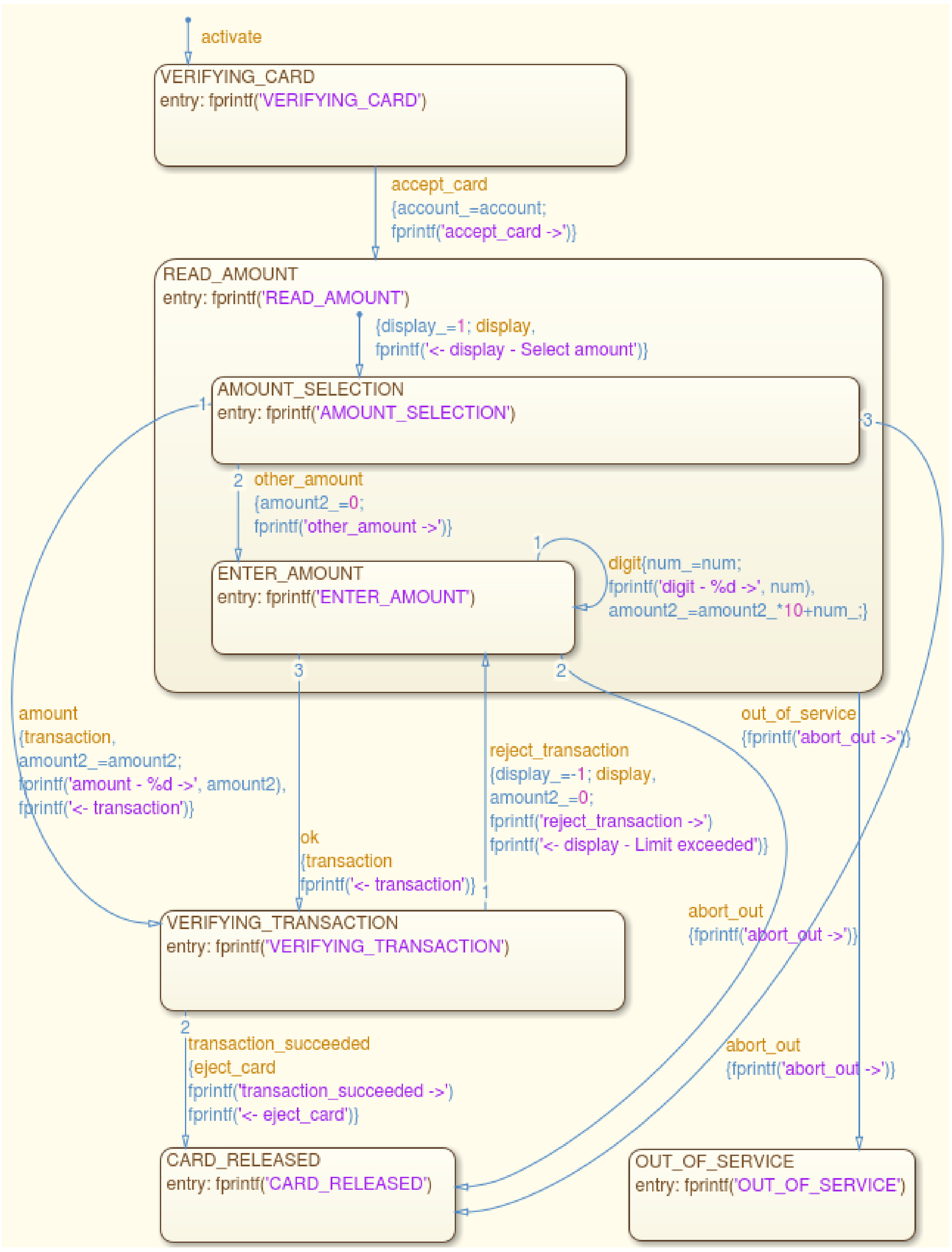

{activate, activeVA}
S1
after(1, sec) {accept_card, account=2345}
S2
after(1, sec) {amount2=5000, amount}
S3
after(1, sec) {reject_transaction}
S5
after(1, sec) {num=5, digit}
S4
after(1, sec) {num=0, digit}
S6
after(1, sec) {num=0, digit}
S7
after(1, sec) {ok}
S14
after(1, sec) {reject_transaction}
S8
after(1, sec) {num=5, digit}
S9
after(1, sec) {ok}
S10
after(1, sec) {transaction_succeeded}
S11

Figure 20: State Machine to Create the Same Sequence of Signal

Though Simulink work differently and it does not facilitate MSC tracing, a simple trick is applied to simulate the same sequence of signal/event calling. A state machine is created to repeat the sequence and some information is printed out to make the execution order exportable and comparable.


activate
VERIFYING_CARD
entry: fprintf('VERIFYING_CARD')
accept_card
{account_=account;
fprintf('accept_card ->')}
READ_AMOUNT
entry: fprintf('READ_AMOUNT')
{display_=1; display,
fprintf('<- display - Select amount')}
AMOUNT_SELECTION
entry: fprintf('AMOUNT_SELECTION')
1
3
2
other_amount
{amount2_=0;
fprintf('other_amount ->')}
ENTER_AMOUNT
entry: fprintf('ENTER_AMOUNT')
1
digit{num_=num;
fprintf('digit - %d ->', num),
amount2_=amount2_*10+num_;}
3
2
amount
{transaction,
amount2_=amount2;
fprintf('amount - %d ->', amount2),
fprintf('<- transaction')}
out_of_service
{fprintf('abort_out ->')}
reject_transaction
{display_=-1; display,
amount2_=0;
fprintf('reject_transaction ->')
fprintf('<- display - Limit exceeded')}
ok
{transaction
fprintf('<- transaction')}
1
VERIFYING_TRANSACTION
entry: fprintf('VERIFYING_TRANSACTION')
abort_out
{fprintf('abort_out ->')}
2
transaction_succeeded
{eject_card
fprintf('transaction_succeeded ->')
fprintf('<- eject_card')}
abort_out
{fprintf('abort_out ->')}
CARD_RELEASED
entry: fprintf('CARD_RELEASED')
OUT_OF_SERVICE
entry: fprintf('OUT_OF_SERVICE')
Figure 21: ATM State Machine Transformed in Stateflow

Events and states are printed while the ATM machine is simulated in diagnostic view. Below is the print outs of the diagnostic view. It shows only the lines printed by fprintf()

and other lines are excluded to make it easy to read. Finally, behavior of ATM state-machine in SDL and Stateflow is ready to put in contrast.

VERIFYING_CARD 14
accept_card -> 14
READ_AMOUNT 11
<- display - Select amount 26
AMOUNT_SELECTION 16
amount - 5000 -> 16
<- transaction - 5000 21
VERIFYING_TRANSACTION 21
reject_transaction -> 21
<- display - Limit exceeded 27
READ_AMOUNT 11
ENTER_AMOUNT 12
digit - 5 -> 12
ENTER_AMOUNT 12
digit - 0 -> 12
ENTER_AMOUNT 12
digit - 0 -> 12
ENTER_AMOUNT 12
ok -> 5
<- transaction - 500 20
VERIFYING_TRANSACTION 21
reject_transaction -> 21
<- display - Limit exceeded 27
READ_AMOUNT 11
ENTER_AMOUNT 12
digit - 5 -> 12
ENTER_AMOUNT 12

ok ->	5
<- transaction - 5	18
VERIFYING_TRANSACTION	21
transaction_succeeded ->	24
<- eject_card	13
CARD_RELEASED	13

Table 16: Lines Printed in Diagnostic View by Stateflow ATM

Comparison of the sequence of these printed lines of Stateflow and MSC of SDL, proves that these two state-machines are behaving same.

3.2 Stateflow to SDL

An event based state is the ideal situation for using this mapping. An example such as the example below is easily convertible using the mapping discussed earlier.

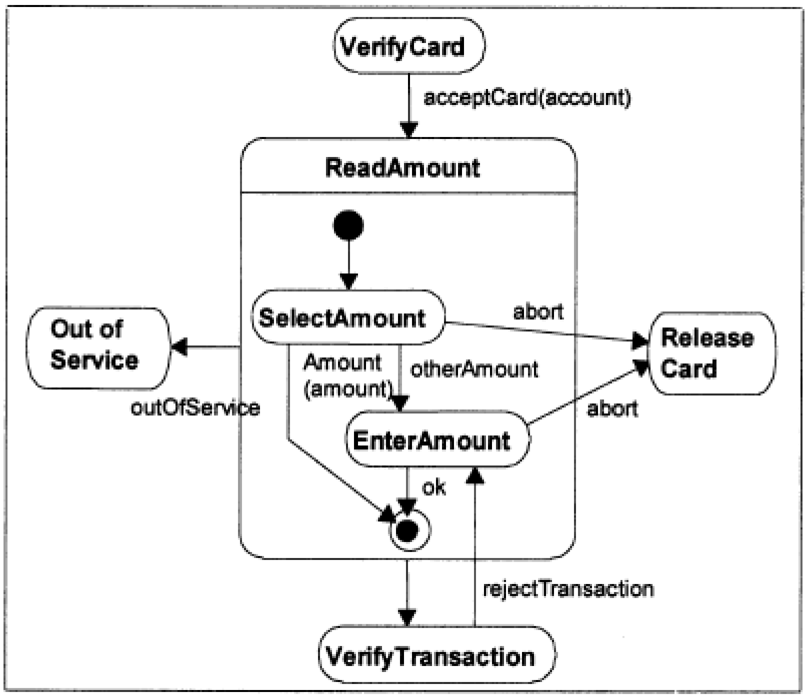


Figure 22: State machine show basic function of a ATM [4]

Guard based Stateflow state machines are more difficult to convert than SDL state machine. A coffee machine state machine is considered for conversion. This state machine has not event, it is completely input and output variable based (these input variables can change anytime). Also this state machine wakes up without any event call.

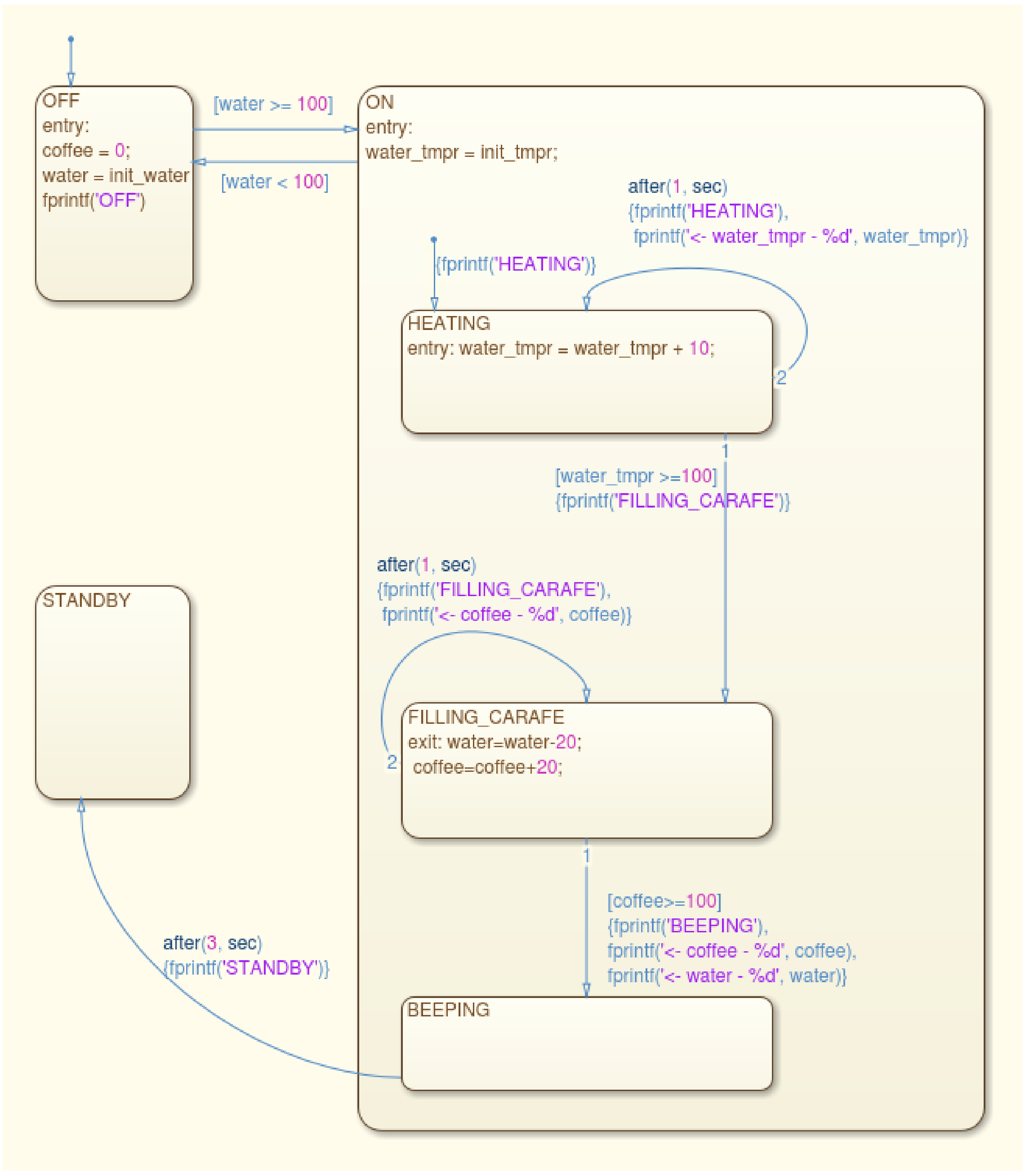


Figure 23: Stateflow Coffee Machine

This coffee machine does two things heat up the water to 100 degree with coffee and sugar, then pour 100 cc coffee in a carafe. It also keep tracks how much water remains in water pot and get turned off when water is less than 100 cc. To make a textual presentation of its behavior, meaningful messages printed on diagnostic view.

water =1400
OFF 3
HEATING 7
HEATING 7
<- water_tmpr - 95 18
FILLING_CARAFE 14
FILLING_CARAFE 14
<- coffee - 0 13
FILLING_CARAFE 14
<- coffee - 20 14
FILLING_CARAFE 14
<- coffee - 40 14
FILLING_CARAFE 14
<- coffee - 60 14
FILLING_CARAFE 14
<- coffee - 80 14
BEEPING 7
<- coffee - 100 15
<- water - 1300 15
STANDBY 7

Table 17: Print out in Diagnostic View of Stateflow Coffee Machine

After converting to SDL state machine it looks quite long because of the way of presentation. Specially, the nested state state has to be displayed in different view that gives SDL a complexer look.

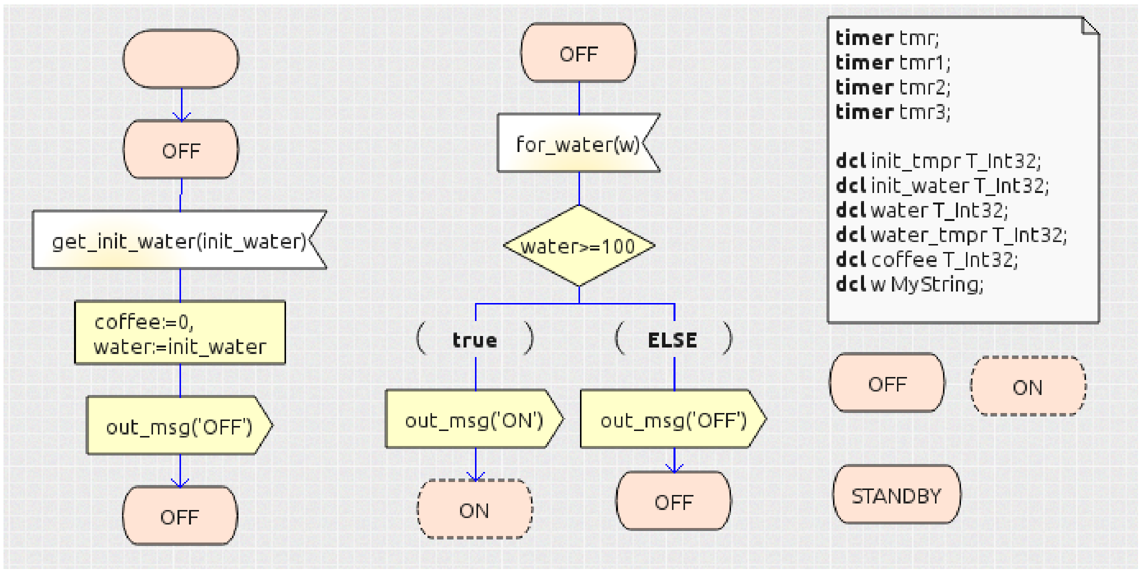


Figure 24: SDL Coffee Machine Part 1/5

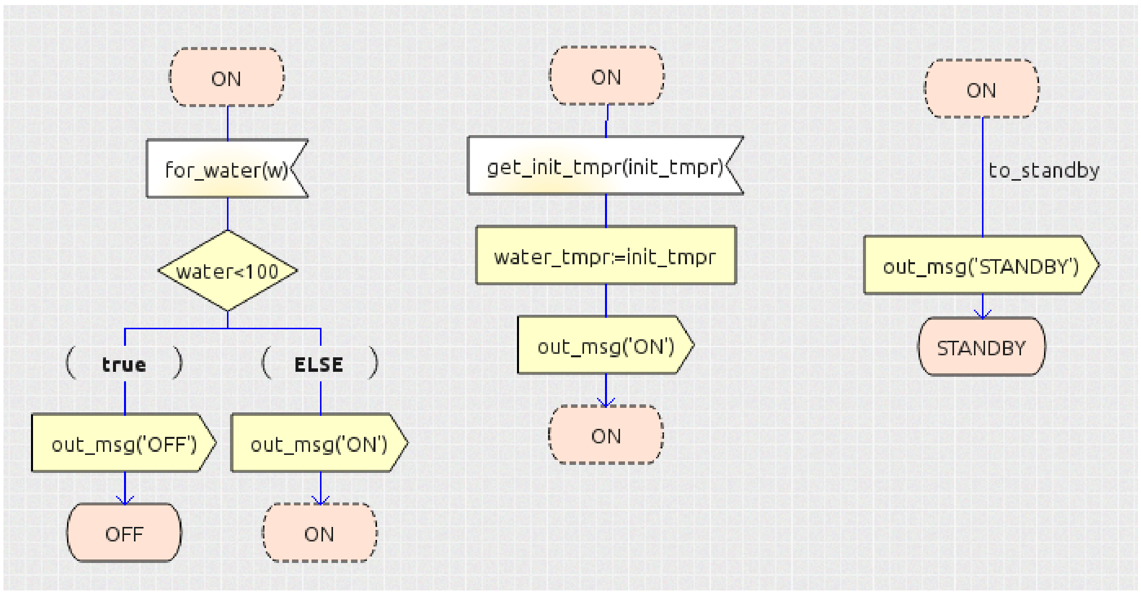


Figure 25: SDL Coffee Machine Part 2/5

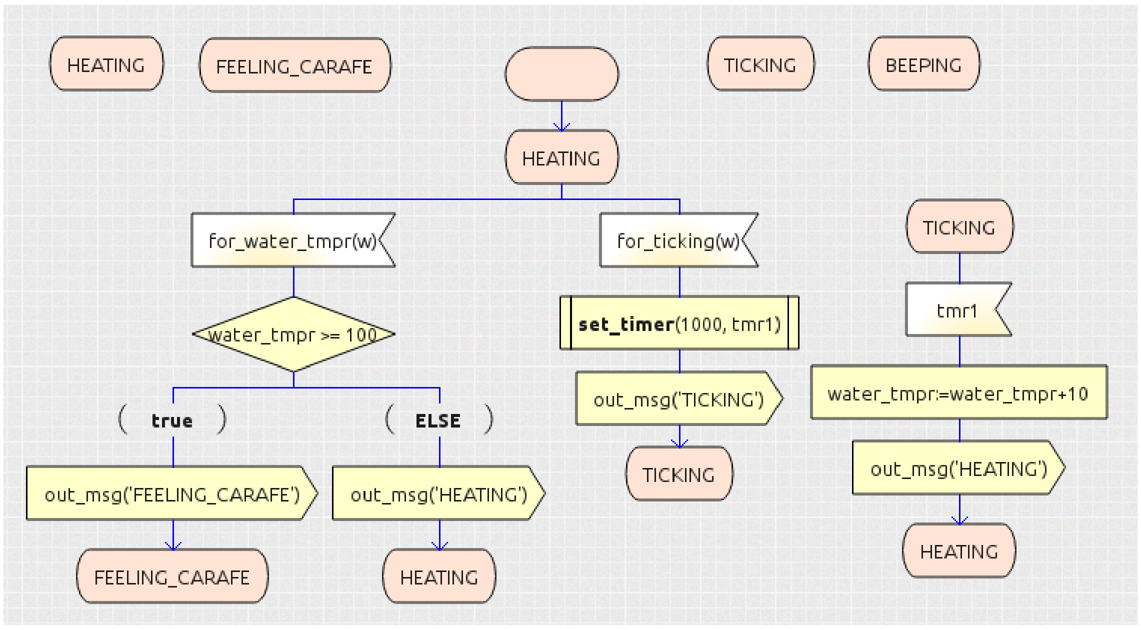


Figure 26: SDL Coffee Machine Part 3/5

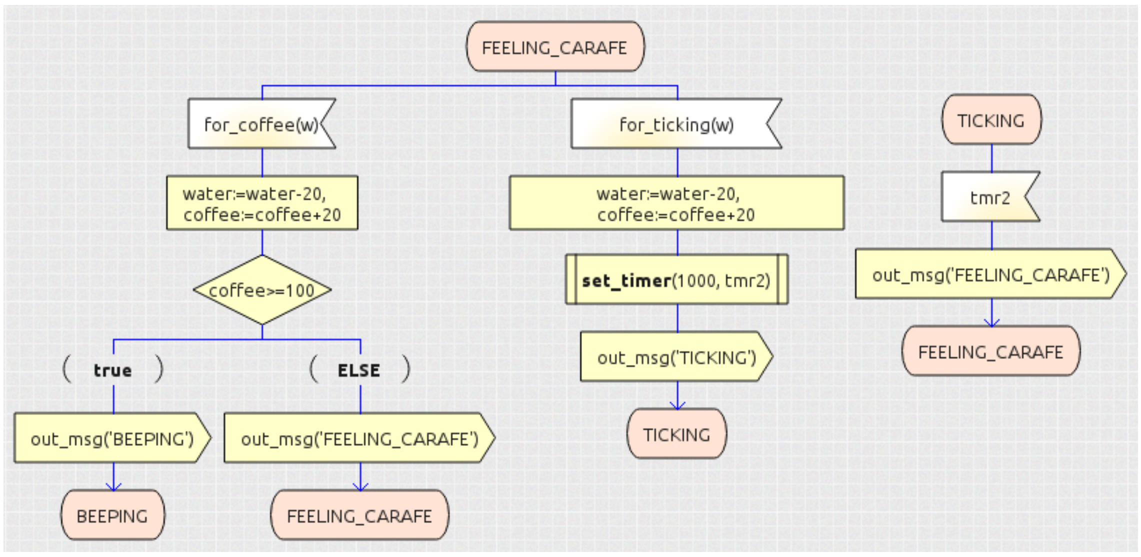


Figure 27: SDL Coffee Machine Part 4/5

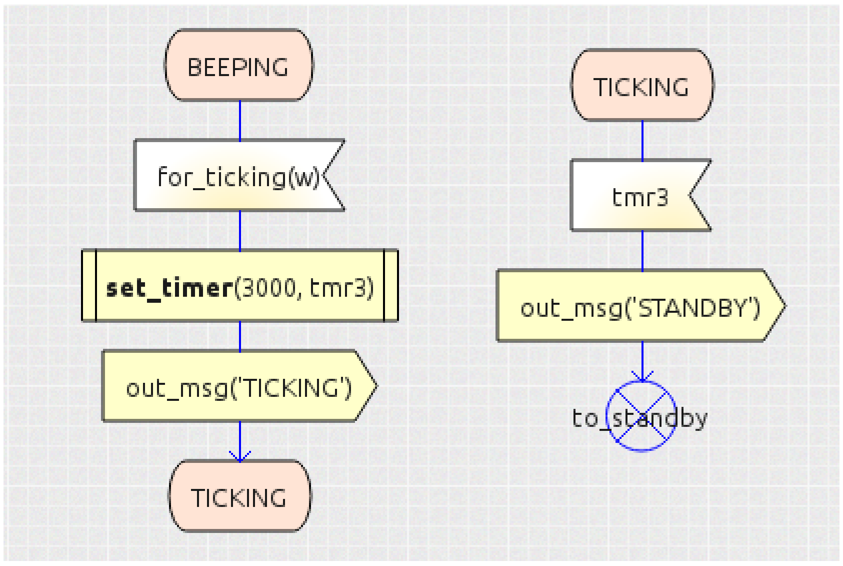


Figure 28: SDL Coffee Machine Part 5/5

SDL transitions can not trigger without input signal (an event or continuous signal). Continuous signals were not considered in the current study. Therefore, to run a SDL state machine, another agent is required to send an input event as a complementary. By the means of TASTE Graphical User Interface (GUI) signals has been sent. This signals are sent in sequence that results in a comparable MSC.

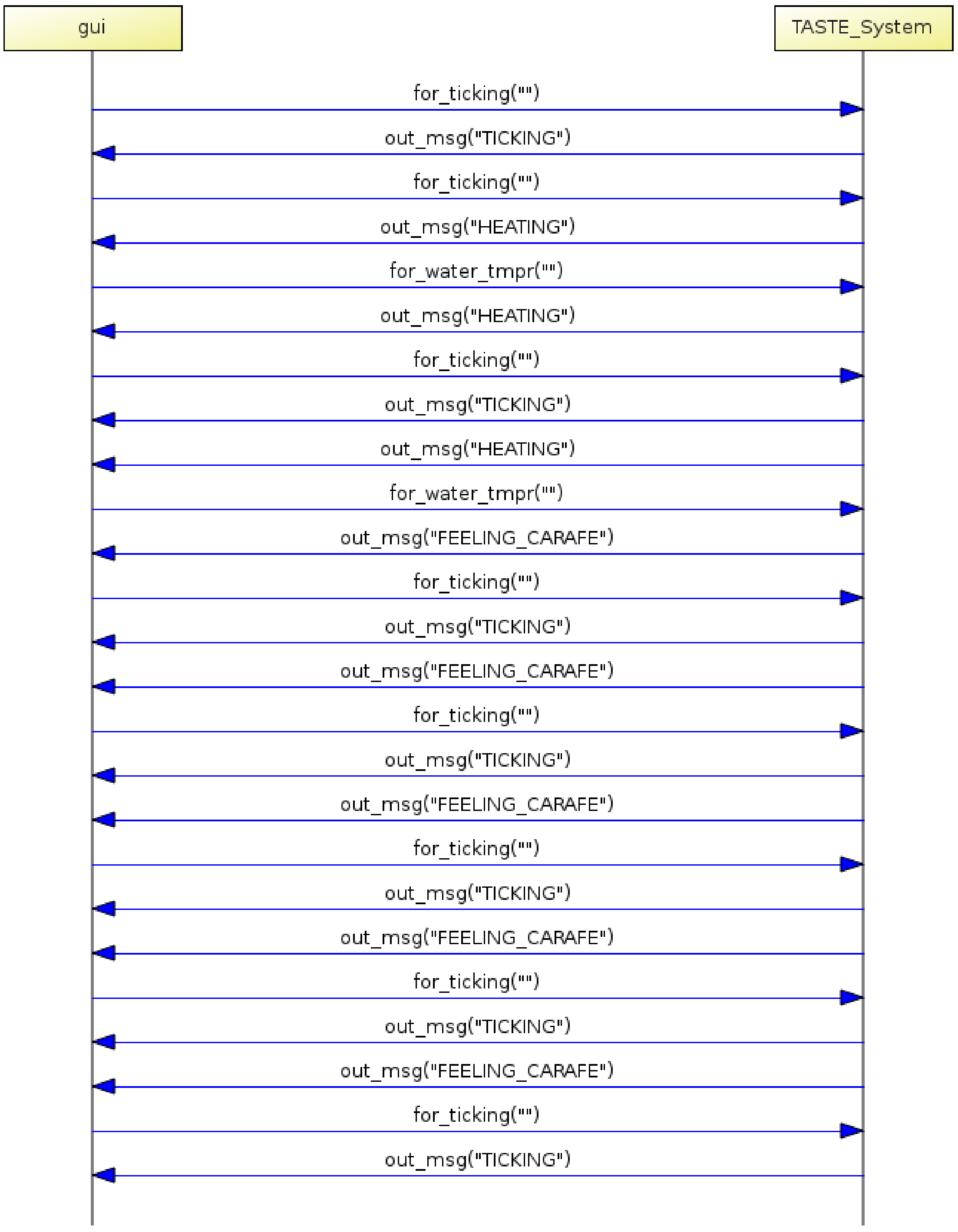


Figure 29: MSC of SDL Coffee Maching Part 1/2

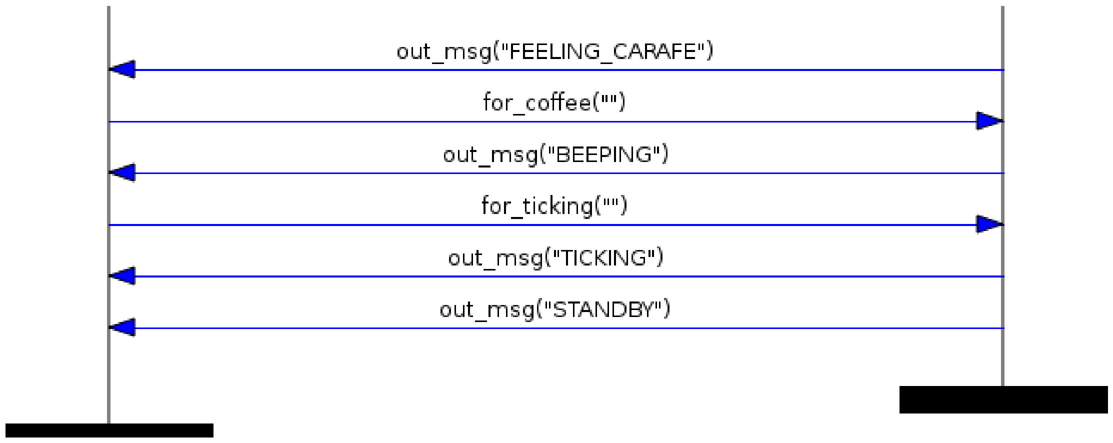


Figure 30: MSC of SDL Coffee Maching Part 2/2

The SDL MSC looks longer than the number of lines printed in Stateflow diagnostic view, because SDL state machine requires more signals to send to trigger transitions. For example to simulate temporal logic "for_ticking()" has to be sent many times, that is never required in Stateflow. The way of interaction is different because of the difference of the definitive characteristics of the two formalisms. This comparison clearly reveals that the behavior of the coffee machine in Stateflow and SDL is same.

3.3 Inner vs Outer Transition Inconsistency Issue

Stateflow provides higher priority for outer transition over inner transition [30]. On the other hand, SDL editor OpenGEODE gives error while adding such an overlapping transition. If an input signal is placed for inner state to outer state then it will not allow to add same input signal to be placed for the parent state to any outer state. In the ATM example, if a transition is added from "READ_AMOUNT" to "CARD_RELEASED" using signal "abort_out(w)" OpenGEODE gives error. Therefore, this study did not face any problem of transition inconsistency upon conversion that was described by Danial et al in [1].

4 Guidelines for Automatic Model Transformation

The basic idea is to convert both SDL and Stateflow state machine to a data structure that is preferably easy to parse and convert into any of these two. For example, for converting a SDL to Stateflow, firstly, SDL state machine will be converted to a structured data, then this data will be converted to a Stateflow state machine. So, it is very important to define a data structure that is simple but complete and extensible. It is foreseeable that having a bit human readable and programatically easy to handle data structure can bring a big success in terms of popularity.

4.1 Data Structure

JavaScript Object Notation (JSON) is the formate of choice because of it's simplicity, popularity and the support by a wide range of programming language for parsing and generating. Matlab has jsonencode() to create and jsondecode() to import JSON. In the proposed implementation of the mapping one file will contain one state machine and will contain the state machine features in a list structure. The first level will include variables, events, states and transitions. The second, level will include their specifications.

```
{
  "variables": {
    "<variable identifier>": {
      "scope": "<input/output/local>",
      "type": "<boolean/integer/double/string>",
      "initial_value": "<null/value>",
      "min": <null/value>,
      "max": <null/value>
    },
    ...
  },
  "events": {
    "<event identifier>": {
      "scope": "<input/output/local>",
      "parameters": <null/[variable_identifier, ...]>
    },
    ...
  },
  "messages": {
    "<message identifier>": {
      "scope": "<input/output/local>",
      "type": "<boolean/integer/double/string>"
    },
    ...
  },
  "states": {
    "<state identifier>": {
      "type": "<basic/composite/composite+parallel>",
      "parent": "<null/state identifier>",
       "entry": "<null/state identifier>",
       "during": "<null/state identifier>",
       "exit": "<null/state identifier>"
    },
    ...
  },
  "transitions": {
    "<!sourseless!/state identifier]": [
      {
        "destination": "<state identifier>",
        "order": <integer value>,
        "trigger_type": "<null/event/guard/event+guard>",
        "event": "<null/event identifier>",
        "guard": "<null/guard expression for truthy
checking>",
         "action": "assign all input variable to output
variable such as x_ = x while converting SDL to Stateflow",
        "contents": [
           {"input_event":["<event identifier>", "<variable
identifier of first parameter>", "<variable identifier of
second parameter>" ]},
         {"action":"water=water+100"},
         {"output_event":["<event identifier>", "<variable
identifier of first parameter>", "<variable identifier of
second parameter>" ]},
         {"procedure_call": "<call expression>"}
        ]
      },
      ...
    ]
  }
}
```

Figure 31: Intermediary Data Structure

An example of the data structure is given in the appendix. Rules to be maintained for this data structure are:-

- Events are not allowed use input variables as parameter
- Initial value of input variables must be null
- Variable type must be a data type
- Parent of a sub state must be a composite or parallel state
- Transition “trigger_type” can be "event" or "guard" or "event+guard"

4.2 Algorithms for Implementing The Mappings

Though, state machines will be saved in an intermediary data structure, for a transformation two algorithms are required. Giving a total of four algorithms -

- For converting a Stateflow to SDL
  1. Stateflow state machine to the JSON formated state machine
  2. JSON formated state machine to SDL state machne

- For converting a SDL to Stateflow
  3. SDL state machine to the JSON formated state machine
  4. JSON formated state machine to Stateflow state machine

4.2.1 Algorithm for Stateflow to JSON

Stateflow state machine is readable in Stateflow API. It has a tree structure. [14][15] But, objects of same type are accessible in an array. [13]

```
Part 1. Getting handle

h = load_system(‘file_name.slx’);
rt = sfroot
m = rt.find('-isa','Stateflow.Model');
ch = m.find('-isa', 'Stateflow.Chart');

sm = {};

Part 2. Reading variables and adding to JSON
```

```
vs = ch.find('-isa','Stateflow.Data')
Loop i=1 To length(vs)
    sm.variables.[get(vs[i], ‘name’)] = {
        scope: get(vs[i], ‘scope’),
        type: get(vs[i], ‘type’),
        initial_value: get(vs[i], ‘initial_value’),
        min: get(vs[i], ‘min’),
        max: get(vs[i], ‘max’)
    }
End Loop
```

Part 3. Reading events and adding to JSON

```
es = ch.find('-isa','Stateflow.Event')
Loop i=1 To length(es)
    sm.events.[get(es[i], ‘name’)] = {
        scope: get(es[i], ‘scope’),
        parameters: null
    }
End Loop
```

Part 4. Reading messages and adding to JSON

```
ms = ch.find('-isa','Stateflow.Event')
Loop i=1 To length(es)
    sm.messages.[get(ms[i], ‘name’)] = {
        scope: get(ms[i], ‘Scope’),
        type: deduce from ms[i]
    }
End Loop
```

Part 5. Reading states and adding to JSON

```
ss = ch.find('-isa','Stateflow.State')
stype = “basic”;
sparent = null;
Loop i=1 To length(ss)
    If (ss[i].up is a state) Then
        parent = get(ss[i], ‘name’)
    Else
        sparent = null
    End If
    If (length(ss.find('-isa','Stateflow.State','-depth',1))>0) Then
        stype = “composite”
    End If
    If (get(ss[i], ‘Decomposition’) is “PARALLEL_AND”) Then
        stype = stype + “parallel”
    End If
    sm.states.[get(ss[i], ‘name’)] = {
        type: stype,
        parent:  sparent
    }
End Loop
```

```
Part 6. Reading transitions and adding to JSON

ts = ch.find('-isa','Stateflow.Transition')
Loop i=1 To length(ts)
    sm.transitions = []
    sm.transitions.push({
        source:  get(get(ts[i], ‘Source’), ‘Name’) If Not null,
        destination: get(get(ts[i], ‘Destination’), ‘Name’),
        order: get(ts[i], ‘ExecutionOrder’)
        contents: {
            input_event: truncate ts[i].LabelString using regular expression
            action:  truncate ts[i].LabelString using regular expression
            output_event:  truncate ts[i].LabelString using regular expression
            procedure_call:  truncate ts[i].LabelString using regular expression
        }
    })
End Loop
```

```
Part 7. Writing json file

write the return of jsonencode(sm) to a file.
```

Figure 32: Algorithm for Stateflow to JSON

4.2.2 Algorithm for JSON to SDL

To create SDL state machine to use with OpenGEODE, the .pr has to be produced. To generate .pr a programming language will be used, that will read the JSON and write the file. Algorithm below gives the logics for getting SDL out of JSON.

```
Part 1. Read JSON file into object

jsm = jsondecode(content of the JSON file);
sdl = “”


Part 2. Reading variables and adding to a dataview-uniq.asn file

type = “”
Loop i=1 To length(jsm.variables) as {key:v}
    type += “convert v to asn variable /n”
    Format Example
    “Num ::= INTEGER (0..9)”
End Loop
```

Part 3. Reading events and input variables and write in system_structure.pr file

```
sys = “USE Datamodel;\n
    SYSTEM atm;\n”
incom = []
outgo = []
Loop i=1 To length(jsm.events) as {key:e}
    sys += “convert to consistent format for system_structure file”
    Format example
    “/* CIF Keep Specific Geode PARAMNAMES account */
    SIGNAL accept_card (MyString);”
    If (e.scope == “input”) Then
        incom.push(key)
    End If
    If (e.scope == “output”) Then
        outgo.push(key)
    End If
End Loop
Loop i=1 To length(jsm.variables) as {key:v}
    If (e.scope == “input”) Then
        type += “create signals for input variables
        such get_x(x) for input variable x /n”
        incom.push(“get_variable_identifier”)
    End If
End Loop
sys += “write CHANNEL with specifying signal definition by
concatinating incom and outgo”
sys += “write BLOCK with specifying signal definition by
concatinating incom and outgo”

sys += “ENDSYSTEM;”
```

Part 5. Reading transitions and adding to chart

```
sdl += “process some_name;”
Recursive Function transition_converter(node)
    Loop i=1 To length(jsm.states) as key:s
        If (jsm.states[i].parent == node) Then
            If (Is a destination in jsm.events.”!sourceless!”) Then
                sdl += “START;\n”
                sdl += “NEXTSTATE state_name;\n”
                sdl += “START;\n”
            End If
            If (jsm.states[i].type contain “composite”) Then
                sdl += “substructure \n”
                    transition_converter(jsm.states[i])
                sdl += “endsubstructure; \n”
            End If
            Loop i=1 To length(jsm.transitions.”key”)
                Convert jsm.transitions.”key”[i]
                Add its input event
                    Add its contents - output, task
                    Add destination
            End Loop
        End If
    End Loop
End Recursive Function
transition_converter(null)

N.B. SDL keeps transitions between inner and outer states in a
splitted form, but JSON has them in single transition, these have to
be splitted.

Part 6. Saving sdl to file
```

Figure 33: Algorithm for JSON to SDL

4.2.3 Algorithm for SDL to JSON

SDL can be read by parsing .pr file saved in OpenGEODE. It contains state machine in a tree structure. OpenGEODE provides code generation facility based on custom template. Using this feature .pr file should be converted to a easily parseable formate, such as JSON. Then using a programming language this JSON formate of .pr has to be read into an abject. So, the object, where the SDL will be read in, will have the same tree structure as .pr file. By the means of this object and a programming language the intermediary JSON file is to be generated. Below is the algorithm for generating this intermediary file. Also possible is reading the .pr file line by line in a programming language and creating the object for further processing.

```
Part 1. Read .pr file into an object

sdl = SDL state machine
sm = {}

Part 2. Read in data types into a separate array

Convert dataview-uniq.asn to JSON format using String Template

jdt = jsondecode(content of converted dataview-uniq)

Part 3. Convert variables into JSON
```

```
Recursive Function var_converter(node)
    Loop i=1 To length(node)
        If (type_of(node[i]) == variable) Then
            sm.variables.”node[i].variable_name” = {
                scope: "output",
                type: find in jdt,
                initial_value: null,
                min: find in jdt,
                max:  find in jdt
            }
        Else (If type_of(node[i]) == substructure) Then
            var_converter(node[i])
        End If
    End Loop
End Recursive Function

var_converter(sdl)
```

Part 4. Convert events into JSON

File system_structure.pr contains at least three (while SDL is simulated in TASTE), 1) list of signals with parameters, 2) list of input signals, 3) list output signal. These three will be converted three array – signals, in_sigs, out_sigs.

```
Loop i=1 To length(signals)
    sm.events.”signals[i].identifier” = {
        scope: “input” if in in_sigs, “output” if in out_sigs,
        parameters: {
            Loop param In signals[i].parameters
                param + “_”,
                sm.variables.”param_” = {
                    same to same sm.variables.”param_”
                }
                sm.variables.””.scope = “input”
            End Loop
        }
    }
End Loop
```

Part 5. Convert states into JSON

```
Recursive Function state_converter(node)
    Loop i=1 To length(node)
        If (type_of(node[i]) == state && not_in(sm.states)) Then
            sm.states.”node[i].state_name” = {
                type: "basic",
                parent:
                    If  type_of(node) == state Then
```

```
                        ”node.state_name”
                    Else null
                }
            Else (If type_of(node[i]) == substructure) Then
                sm.states.”node[i].state_name” = {
                    type: "composite",
                    parent: ”node[i].state_name”
                }
                state_converter(node[i])
            End If
        End Loop
    End Recursive Function

    state_converter(sdl)
```

Part 6. Convert transitions into JSON

```
fht = {}
lht = {}
sm.transitions = []
Recursive Function transition_converter(node)
    Loop i=1 To length(node)
        If (type_of(node[i]) == START) Then
            sm.transitions.”!sourseless!”.push({
                destination: node[i].NEXTSTATE
            })
        Else If (type_of(node[i]) == state && contain(input)) Then
            If(sm.transitions.”node[i].state_name” == undefined) Then
                sm.transitions.”node[i].state_name” = []
            End If
            Loop j=1 To length(sm.transitions.”node[i].state_name”)
                input = sm.transitions.”node[i].state_name”[j]
                et = {
                    trigger_type: “event”
                    event: node[i].input
                    guard: node[i].decision
                    contents: []
                }
                If (input signal has parameters) Then
                     assign all input variable to output variable
                     such as x_ = x
                End If
                If (input contain decision) Then
                    ee = et.clone()
                    et.destination = input.true.NEXTSTATE
                    et.contents.push(
                        contents of input.decision.true
                    )
                    ee.destination = input.ELSE.NEXTSTATE
                    ee.contents.push(
                        contents of input.decision.ELSE
                    )
                Else
                    et.destination = input.NEXTSTATE
                    et.contents.push(
                        contents of  input
                    )
                    sm.transitions.”node[i].state_name”.push(et)
                End If
            End Loop
        Else If (type_of(node[i]) == substructure) Then
            transition_converter(node[i])
        End If
    End Loop
End Recursive Function

transition_converter(sdl)

N.B. SDL keeps transitions between inner and outer states in a
splitted form, but JSON has them in single transition, these have to
be joined.

Part 7. Saving JSON to file
```

Figure 34: Algorithm for SDL to JSON

4.2.4 Algorithm for JSON to Stateflow

By the means of Stateflow API, JSON will get converted to Stateflow.

```
Part 1. Read JSON file into object and creating handle

jsm = jsondecode(content of the JSON file);
rt = sfroot;
sfnew;
m = rt.find('-isa','Stateflow.Model');
ch = m.find('-isa', 'Stateflow.Chart');


Part 2. Reading variables and adding to chart

chvs = []
Loop i=1 To length(jsm.variables) as {key:v}
    chvs[i] = Stateflow.Data(ch)
    chvs[i].Name = key
    chvs[i].Scope = v.scope
    chvs[i]. type = v. type
    chvs[i].InitialValue = v.initial_value
    chvs[i].Minimum = v.min
    chvs[i].Maximum = v.max
End Loop

Part 3. Reading events and adding to chart

ches = []
Loop i=1 To length(jsm.events) as {key:e}
    ches[i] = Stateflow.Event(ch)
    ches[i].Name = key
    ches[i].Scope = e.scope
End Loop

Part 4. Reading states and adding to chart

chss = []
temp = null
Loop i=1 To length(jsm.states) as {key:s}
        If (s.parent != null) Then
            temp = ch.find(
                '-isa','Stateflow.State','-and','Name', s.parent
            )
            chss[i] = Stateflow.State(temp)
            chss[i].Name = key
        Else
            chss[i] = Stateflow.State(ch)
            chss[i].Name = key
        End If
        If (s.type contain “parallel”) Then
            chss[i].Decomposition = “PARALLEL_AND”
        End If
End Loop

Part 5. Reading transitions and adding to chart

chts = []

Loop i=1 To length(t)
    chts[i] = Stateflow.Transitions(ch)
    chts[i].Source =  jsm.states.[t.source].inch
    chts[i].Destination = jsm.states.[t.destination].inch
    chts[i].ExecutionOrder = t.order
    chts[i].LabelString = generate from t.compnents
End Loop


Part 6. Saving chart to file
```

Figure 35: Algorithm for JSON to Stateflow

4.3 Prototyping for Demonstration

To demonstrate how to apply the conversion approach a prototype is under development. It is placed on Github as a public repository (https://github.com/taheruddin/transformation-stateflow-sdl). It is expected to be ready very shortly. All the updates from now will be available in the repository.

5 Conclusion

The aim of this work is to find out and validate a possible semantic mapping between subsets of two state machine modeling languages, Stateflow and SDL. Secondly, to propose an optimistic approach for the implementation of this mapping.

Both Stateflow and SDL have been studied and the mappings between a substantial amount of their features are explored. These mappings are simulated using small representative models to focus on a specific semantic equality that results in same behavior. This simulation revealed behavioral equality.

Moreover, two case studies have been conducted, ATM and Coffee Machine, to find out the validity of mappings as well as the quality of the total conversion work. These case studies show that conversion from SDL to Stateflow (ATM) is easier and more efficient. Converting input event based state machine from Stateflow to SDL is also a piece of cake. However, input value based Stateflow state machine is a bit difficult to convert, but it was successful to produce useful output. These models are placed in the same Github public repository (https://github.com/taheruddin/transformation-stateflow-sdl).

Next the approach for implementation of these mappings is explained. Firstly, the source is to be converted to an intermediate JSON data structure, then this data is to be converted to destination. The structure of this data is very simple and very easy understand and hopefully, easy to use. This data structure is self explanatory and an example is also included in the annex.

Four algorithms are provided for implementing the mappings through the intermediate data structure. Required learning materials are referred. A prototype for demonstrating the approach is currently under development. It is placed on Github as a public repository and future updates will be available there.

Finally, it is an honor to be supervised by Andres Toom and Tõnu Näks. Their direction and continuous support made this work possible for me.

References

[1] Balasubramanian, Daniel, et al. "Polyglot: modeling and analysis for multiple statechart formalisms." *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011.

[2] Crane, Michelle L. *On the Syntax and Semantics of State Machines*. Diss. Queen's University, 2005.

[3] Rodionov, Dan. "IMPLEMENTING TTÜ NANOSATELLITE COMMUNICATION PROTOCOL USING TASTE TOOLSET." *TALLINN UNIVERSITY OF TECHNOLOGY*, 2017.

[4] Møller-Pedersen, Birger, and Dagbjørn Nogva. "Scalable and object Oriented SDL State (chart) s." *Formal Methods for Protocol Engineering and Distributed Systems*. Springer, Boston, MA, 1999. 59-73.

[5] Grammes, Rüdiger, and Reinhard Gotzhein. "SDL Profiles–Formal Semantics and Tool Support." *International Conference on Fundamental Approaches to Software Engineering*. Springer, Berlin, Heidelberg, 2007.

[6] Fischer, Joachim, et al. "SDL-2000: A Language with a Formal Semantics." *Rigorous Object-Oriented Methods*. 2000.

[7] George, Dony, et al. "Programming Language Inter-conversion." *syntax* 1.20 (2010).

[8] Czarnecki, Krzysztof, and Simon Helsen. "Classification of model transformation approaches." *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. Vol. 45. No. 3. 2003.

[9] Sendall, Shane, and Wojtek Kozaczynski. "Model transformation: The heart and soul of model-driven software development." *IEEE software* 20.5 (2003): 42-45.

[10] Harel, David. "Statecharts: A visual formalism for complex systems." *Science of computer programming* 8.3 (1987): 231-274.

[11] Lilius, Johan, and Ivn Porres Paltor. "The semantics of UML state machines." (1999).

[12] "Stateflow." *MATLAB & Simulink*, The MathWorks, Inc, www.mathworks.com/products/stateflow.html.

[13] *MATLAB Data Acquisition Toolbox User's Guide*, http://files.matlabsite.com/docs/books/matlab-docs/data_acquisition_toolbox_daqug_r2015a.pdf.

[14] *Overview of the Stateflow API*. The MathWorks, Inc, 2018, se.mathworks.com/help/stateflow/api/overview-of-the-stateflow-api.html.

[15] *Stateflow Hierarchy of Objects*. The MathWorks, Inc, 2018, se.mathworks.com/help/stateflow/ug/stateflow-hierarchy-of-objects.html.

[16] *Access the Properties and Methods of Objects*. The MathWorks, Inc, 2018, se.mathworks.com/help/stateflow/api/accessing-the-properties-and-methods-of-objects.html.

[17] *Create a MATLAB Script of API Commands*. The MathWorks, Inc, 2018, se.mathworks.com/help/stateflow/api/creating-a-matlab-script-of-api-commands.html.

[18] Tutorials Point. *MATLAB Tutorial*. Tutorials Point, 19 Mar. 2018, www.tutorialspoint.com/matlab/index.htm.

[19] *Jsonencode*. The MathWorks, Inc, 2018, se.mathworks.com/help/matlab/ref/jsonencode.html.

[20] Jsondecode. The MathWorks, Inc, 2018, se.mathworks.com/help/matlab/ref/jsondecode.html.

[21] *Properties and Methods Sorted By Chart Object*. The MathWorks, Inc, 2018, www.mathworks.com/help/stateflow/api/properties- and-methods-sorted-by-chart-object.html.

[22] *Integrate Custom C/C++ Code for Simulation*. The MathWorks, Inc, 2018, se.mathworks.com/help/stateflow/ug/procedures-for-simulation.html.

[23] *View Differences Between Stateflow Messages, Events, and Data*. The MathWorks, Inc, 2018, se.mathworks.com/help/stateflow/ug/view-differences-between- messages-events-and-data-1.html.

[24] *Integrate C Code Using the MATLAB Function Block*. The MathWorks, Inc, 2018, se.mathworks.com/help/simulink/ug/incorporate-c-code-using-a-matlab-function-block.html.

[25] *Temporal Logic Operators*. The MathWorks, Inc, 2018, se.mathworks.com/videos/temporal-logic-operators-1487800687476.html.

[26] *Control Chart Execution Using Temporal Logic*. The MathWorks, Inc, 2018, se.mathworks.com/help/stateflow/ug/using-temporal-logic-in-state-actions-and-transitions.html.

[27] *Execution of a Stateflow Chart*. The MathWorks, Inc, 2018, se.mathworks.com/help/stateflow/ug/chart-during-actions.html.

[28] *Activate a Stateflow Chart Using Input Events*. The MathWorks, Inc, 2018, se.mathworks.com/help/stateflow/ug/using-input-events-to-activate-a-stateflow-chart.html.

[29] *How Events Work in Stateflow Charts*. The MathWorks, Inc, 2018, se.mathworks.com/help/stateflow/ug/how-events-work-in-stateflow-charts.html.

[30] *Execution Order in Stateflow*. Gamax Laboratory Solutions Ltd., 27 Aug. 2015, gamaxlabsol.com/execution-order-stateflow/?lang_=EN.

[31] *Technical Topic: Use of Timers in User Code with TASTE*. TASTE, 4 Aug. 2017, 21:02, taste.tuxfamily.org/wiki/index.php ?title=Technical_topic:_Use_of_timers_in_user_code_with_TASTE

[32] *What Is Finite State Machine? - Definition from*. whatis.techtarget.com/definition/finite-state-machine. Accessed 5 Dec. 2017.

[33] *PragmaDev Studio Tutorial*. www.pragmadev.com/downloads/Manuals/Tutorial.pdf.

[34] PragmaDev. Product Prices. PRAGMADEV SARL, www.pragmadev.com/prices.html.

[35] State Action Types. The MathWorks, Inc, 2018, se.mathworks.com/help/stateflow/ug/state-action-types.html#f0-123926.

[36] *Create and Access Charts Using the Stateflow API*. The MathWorks, Inc, 2018, se.mathworks.com/help/stateflow/api/quick-start-for-the-stateflow-api.html.

[37] *Finite-State Machine*. Wikipedia, 7 May 2018, en.wikipedia.org/wiki/Finite-state_machine.

[38] PragmaDev. *PragmaDev - Modeling and Testing Tools*. PRAGMADEV SARL, 2018, www.pragmadev.com/index.html.

[39] *Abstract State Machines*. Wikipedia, 13 Feb. 2018, en.wikipedia.org/wiki/Abstract_state_machines.

[40] *Overview*. TASTE, taste.tuxfamily.org/wiki/index.php? title=Overview.

[41] *Types of Chart Execution*. The MathWorks, Inc, 2018, se.mathworks.com/help/stateflow/ug/types-of-chart-execution.html.

[42] *Technical Topic: OpenGEODE, an SDL Editor for TASTE*. TASTE, 2018, taste.tuxfamily.org/wiki/index.php?title=Technical_topic %3A_OpenGEODE%2C_an_SDL_editor_for_TASTE.

[43] Tiwari, Ashish. *Formal semantics and analysis methods for Simulink Stateflow models*. Technical report, SRI International, 2002.

[44] Keller, Robert M. "Computer science: Abstraction to implementation." *Harvey Mudd College, Claremont, CA, United States* (2001).

[45] Cheng, Kwang-Ting, and Avinash S. Krishnakumar. "Automatic generation of functional vectors using the extended finite state machine model." *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 1.1 (1996): 57-79.

[46] *Extended Finite-State Machine*. Wikipedia, 10 Apr. 2018, en.wikipedia.org/wiki/Extended_finite-state_machine.

[47] Hamon, Grégoire. "A denotational semantics for Stateflow." *Proceedings of the 5th ACM international conference on Embedded software*. ACM, 2005.

Appendix 1 – Data Structure Example

```
{
  "variables": {
    "water": {
      "scope": "input",
      "type": "integer",
      "initial_value": null,
      "min": 0,
      "max": 2000
    },
    "faulty": {
      "scope": "input",
      "type": "boolean",
      "initial_value": null,
      "min": null,
      "max": null
    },
    "amount": {
      "scope": "output",
      "type": "double",
      "initial_value": 55.75,
      "min": 0,
      "max": 999999
    },
    "account": {
      "scope": "local",
      "type": "string",
      "initial_value": "EE87281471849",
      "min": null,
      "max": null
    },
    "msg": {
      "scope": "output",
      "type": "string",
      "initial_value": null,
      "min": null,
      "max": null
    }
  },
  "events": {
    "activate": {
      "scope": "input",
      "parameters": null
    },
    "insert_card": {
      "scope": "input",
      "parameters": ["account",
"another_output_or_local_variable"]
    },
    "verify_transaction": {
      "scope": "input",
      "parameters": ["account", "amount"]
    },
    "cancel": {
      "scope": "input",
      "parameters": null
    },
    "eject_card": {
      "scope": "output",
      "parameters": null
    },
    "out_msg": {
      "scope": "output",
      "parameters": ""
    }
  },
  "states": {
    "verifying_card": {
      "type": "basic",
      "parent": null
    },
    "read_amount": {
      "type": "composite",
      "parnet": null
    },
    "amount_selection": {
      "type": "sub",
      "parnet": "read_amount"
    },
    "entering_amount": {
      "type": "sub",
      "parent": "read_amount"
    },
    "verifying_transaction": {
      "type": "basic",
      "parent": null
    }
  },
  "transitions": {
    "!sourseless!": [
      {
        "destination": "verifying_card",
        "trigger_type": "event",
        "event": "activate",
        "guard": null,
        "contents": [
          {"action":"water=water+100"},
          {"output_event":["out_msg",
"value_of_first_parameter"]}
        ]
      }
    ]
  },
  "verifying_card": [
    {
      "destination": "read_amount",
      "order": 1,
      "trigger_type": "guard",
      "event": null,
      "guard": "amount<200",
      "contents": [
        {"action":"water=water+100"},
        {"output_event":["out_msg",
"value_of_first_parameter"]}
      ]
    },
    {
      "destination": "read_amount",
      "order": 2,
      "trigger_type": "event+guard",
      "event": "verify_transaction",
      "guard": "water<100",
      "contents": [
        {"action":"water=water+100"},
        {"output_event":["out_msg",
"value_of_first_parameter"]}
      ]
```

```
        }
    ]
}
```

Figure 36: An Example of Proposed Data Structure