SIMULATOR FOR THE STATECHART BASED LANGUAGE

ADVAIT LONKAR

AMOGH JOHRI

LASYA

June 7, 2020

STATECHART

Complex systems in modern day essentially need specification models. The existing specification models need to evolve alongside the development of coding practices like incremental development and modularization. These specification languages provide the benefit of automatic verification, where defects can be detected very early in the development cycle.

A statechart is a visual formalism for the specification of complex systems. The system described here is basically an 'object' with a life-cycle which goes through a finite number of discrete states.

But the Statechart is a semi-formal language which makes it a hurdle to make automatic verification of Statechart models. Due to it being semi-formal, there has been a lot of research to define the Statechart's formal semantics.

However, Statechart is widely accepted in the software engineering community, since the growing complexity of softwares has called for an increased level of engineering and consequentially, increased levels of specification and analysis.

But why should one use statechart?

- It is easier to understand statecharts than other forms of code.
- The behaviour of the system is separated from the components of the statecharts, so it is easier to:

  - Make changes to the behaviour of the system.
  - Reason about the code.
  - Test the behaviour of the code independently.
- The process of building statecharts makes the developers explore all the possible states.
- Exceptional situations can dealt with the help of statecharts, due the ability of the statecharts to lend themselves to dealing with them so easily, which might be overlooked otherwise.
- Statecharts scale well with the increasing complexity of the systems.
- Statecharts can be a great way to communicate the complicated software systems to non-developer audience.

A simple statechart can be seen in figure 1.

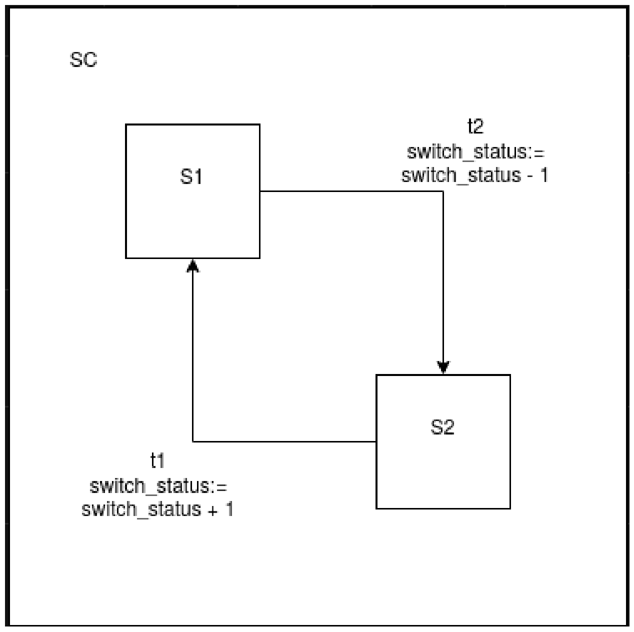

Figure 1: A simple statechart

STABL

Rationale

In an attempt to understand StaBL, it is essential that we take the following into consideration: Statecharts are a part of Unified Modeling Language (UML) suite. Hence, the entire approach should have Object Oriented Design principles and practices, at its very core.

As Statecharts aim to model systems with complex life-cycles, more often than not such Statechart models can span up to contain hundreds of states and transitions. Moreover, such complex systems are often developed by a number of teams working together in co-ordination. In order to facilitate the same, Statechart modeling should employ the concepts of abstraction, modularity, etc.

However, all current implementation of Statechart modelling languages have a severe shortcomings with respect to our above defined goals, they all utilize variables which are global to the entire model. This can lead to a design which hinders collaborative work, is tedious to maintain and difficult to reuse.

StaBL is an attempt to introduce local variables to the Statechart specification system. StaBL stands for Statechart Based Language. It has nuanced semantics for states, hierarchical states, transitions and even advanced features like history states.

Key Points

Before proceeding to the project, we shall discuss the important points as well as the similarities and differences between StaBL, and the abstract notion of Statechart modeling as discussed in the earlier section.

1. A Statechart is a model of a complex-system where its life-cycle is represented through states, transitions, etc. StaBL is a *Specification Language* for specifying a Statechart model. (For an intuitive approach, we can consider Discrete Finite Automaton, which represent models to identify Regular Languages, and consider a programming-language X which allows you to input the 5 tuple-definition corresponding to a particular DFA in order to simulate the same. Here, Statechart corresponds to DFA and StaBL corresponds to the language X).

2. Every **Statechart model** contains of 4 elements :
   a) **S**: Set of one or more states
   b) **sc**: Unique top-level state (This is the single unique state which will contain all the elements (states, transitions, declarations,etc) for the Statechart. This shall be the case for every Statechart model that we consider here-after)

c) **A**: Action Language - Statechart models are allowed to add fragments of code on transitions and states (this interaction with action language effectively makes them an infinite state machine). There are three cases when these transitions are executed:
   - State entry (*Entry Action*)
   - State exit (*Exit Action*)
   - Making a Transition (*Transition Action*)

d) **E**: Events - these model external interaction of the Statechart which trigger the *transitions*. These have not been included in the scope of our current project, and for all practical purposes we have considered a single-global infinitely occurring event which occurs throughout the simulation and triggers all the transitions at all times.

3. Every **State** is represented by **9** elements:

   a) a **name** n for the state,

   b) a unique **parent** p, which is undefined for the Statechart **sc**,

   c) a set of **variables** V, there are three types of Variables (however, for the present project, all variables are considered to be *static*):
      - **Local**: these variables have their scope within the containing state (and all of its sub-states). In addition, these have their life-time strictly aligned to the duration the machine spends in the particular state (or its sub-states).
      - **Static**: these are the same as local-variables except for that their life-time exceeds the time machine spends in the corresponding state. These retain their values until redefined later. For the present project, all variables are considered to be static variables.
      - **Parameter**: these are the same as static variables in terms of life-time however, their scope if larger than static variables as these are (write-)visible to all the incoming transitions as well (discussed in detail later).

   d) an **entry action** a_N, this is the *fragment of action-language code* executed every-time the state is entered,

   e) an **exit action** a_X this is the *fragment of action-language code* executed every-time the state is exited,

   f) a child **state set** $C \subset (S \setminus \{s, sc\})$, (if $C = \phi$, then the state is referred to as an *atomic-state*),

   g) a set of **transitions** T, and

   h) $i \in C$ called the **initial sub-state** (in practice, this has been considered to be the first state in C (C.[0]) for every non-atomic state),

i) and a **history marker** *h*. At any point of time during the execution of the Statechart the *State History* maps a state to its child state which was exited most recently. For a state with a history marker, upon re-entering the state the child-state as mapped by the State History is chosen as the initial sub-state.

4. Every **Transitions** is represented by **7** elements:
   a) a **name** *n* for the transition,
   b) a unique **parent** state *p* for the transition (for of visual representation, this is the smallest-state in the Statechart which fully contains the transition arrow),
   c) a **source state** *src*,
   d) an **event** *e* which triggers the transitions,
   e) a **guard** *g* statement for the transition: these are Boolean expressions which perform a gating mechanism for the transition, the transition can be taken only when *g* evaluates to *True*,
   f) an **action** *a*: this is the *fragment of action-language code* executed every-time the transitions is taken, and
   g) and a **destination state** *dest*.

5. Scoping Rules
   We define **Environment** as the mapping between values and their variables.
   When an action is being executed/type checked, we look up for value/type bindings to variable in two different environments:
   a) **Read Environment** (E_R): This is the environment corresponding to a variable *being read* from.
   b) **Read Environment** (E_W): This is the environment corresponding to a variable *being written* to.

   For example: Suppose we have an integer variable *x* and two environments *A* and *B*. *A* contains x bound to the value 2, and *B* contains x bound to the value 3. Now, we have a statement: $x \ := \ x + 4$ where the E_R corresponds to *A* and the E_W corresponds to *B*.
   One execution of the statement, the following happens:
   1. Since E_R is *A*, the value of *x* being read corresponds to the binding present in *A*. The value is being read to evaluate the expression *x* + *4* on the R.H.S hence, the R.H.S evaluates to: 6, as x get the value: 2.
   2. Since E_W is *B*, the value being written to will correspond to the environment *B*. Hence, after execution, environment *A* will contain $x = 2$ and environment *B* will contain $x = 6$.

The inter-play between environments, states and transitions in StaBL is as follows:

a) Environment corresponding to a state contains the declarations made in that state, as well as the environment of it's parent state(this as we can see, is a recursive definition). In the case of conflict in declarations, the precedence goes from highest to lowest, from the inner-most state to the outer-most state (this can be considered similar to *shadowing* as in most programming languages).

b) Every state has: $E_R = E_W$.

c) For transitions, E_R corresponds to the source state's environment where as E_W corresponds to that of the parent state's environment(parent for the transition) (unless we have a *parameter* variable, in which case E_W will correspond to the destination state's environment).

SIMULATOR FOR STABL

Work Previously Done

The available StaBL compiler code had the following 3 features:

- Lexer
- Parser
- Type-checker

The StaBL code can be written in a text file. The available compiler would then proceed to do the following:

- Lexer would retrieve the relevant tokens from the text-file and provide it to the parser.
- Parser would receive the tokens and incorporate the type-checker to return a type-checked abstract syntax tree for the Statechart.

The type-checker provides significant convenience to the front-end of the compiler as it now requires far less intensive error-handling. *For example: if a piece of StaBL code has been type-checked, we can be sure that the* ***guard expression*** *for every* ***transition*** *will evaluate to a* ***Boolean Constant****.* If this was not taken-care of by the type-checker, there would be a number of erroneous cases which would have to be explicitly taken care of by the simulator.

Constraints

We were not supposed to make changes to the lexer, parser, type-checker or the available assets, hence, for features like *taking an input*, *incorporating history marker*, etc we have resorted to a functional work-around. This helps us towards our goal of studying the behavior of Statechart models and StaBL with the additional functionalities.

Current Work

For our current project, the action language for StaBL is *Java*. The available code had assets existing for the different abstract syntactic elements. With the provided StaBL compiler code-base and the Statechart modeling knowledge, we were required to implement a simulator for StaBL.

The simulator was expected to be a user-interface for analyzing the *flow* of the specified Statechart model. These are the following deliverable:

- The Simulator deals with three-basic types of data, which are *integer*, *Boolean* and *string*.
- All variables are considered to be *static variables* (in the definitions of StaBL).
- The Simulator can perform basic binary and n-ary expression evaluation, ex: *addition*, *multiplication*, etc for integers, *AND*, *OR* operations, etc for Boolean.
- The Simulator can execute the various statements (Assignment statement, Conditional statement, Loop statement, Expression statement, Halt statement and Skip statement) as defined in StaBL (since the names make these statements self-explanatory, we have decided not to explicitly define them all in the report).
- Using the above mentioned, the simulator simulates the behavior of the Statechart in a **transition-by-transition** fashion. On each key-stroke of the user, the following takes place:
  - Next transition is taken.
  - Prints the state in which the Statechart arrives after the transition completes.
  - Prints all the Statechart variables and their current binding values, and the next atomic-state that the Statechart is present in.
- The Simulator **functionally** simulates the *State History* feature.

To accomplish the same, the Simulator maintains the following for each simulation:

- Map <Declaration, Expression> (map) - This maintains a mapping for all the variables declared in the Statechart to their current bindings. The variables are identified through the *Declaration*, which is unique for every variable, and their value is given as an *Expression*.
- List <Transition> (transitions) - This maintains a list for all the transitions declared in the Statechart. This is merely a concatenation of all *Transitions* list corresponding to the the individual states of the Statechart. StaBL treats the case of non-determinism (finding multiple transitions which can be validly taken at a single time-instant) as an issue/defect. Hence, we do not need to worry about maintaining the order of transitions in the list, as there is a maximum of one valid transition in all non-erroneous scenarios.

- Map <State, Integer> (history_map) - This is a mapping to provide for the functional implementation of *History Marker*. With respect to every state, if the state has a history marker we maintain its last active child state's position in the *child states* list. This position is maintained as an Integer.

Algorithms

The Simulation algorithm is essentially an infinite loop which stops after each transition to accept a key-stroke, and then proceeds forward. This allows the user to pace the analysis of model behavior.

Algorithm 1: Simulator

```
Result: Simulates the statechart
curr <- Current State;
while True do
    curr <- getAtomicState();
    t <- getValidTransition(curr);
    performTransition(t);
    displayMap();
    curr <- t.getDestination();
    counter ++;
end
```

- At any point of time, the system's position corresponds to an atomic state that the system is present in. However, transitions may correspond to a destination state which is non-atomic. In these cases, we need to bubble down to the atomic state of the same. While doing so we also need to execute the *entry statement* corresponding to all the states that fall in our path. While taking the child state for a parent state we also need to check whether the parent state has history marker, if it does then we need to enter the last exited state of the parent (this information his stored in *history_map*).

Algorithm 2: getAtomicState

```
Result: Returns a Valid Transition (in the case of there being one)
init <- Input State;
execute statement -> init.entry;
if init.child_states.isEmpty() then
    return init;
else
    if history_map.containsKey(init) then
        index <- history_map.get(init);
    else
        index <- 0;
    end
    return getAtomicState(init.state[index]);
end
```

- The *getValidTransition()* function takes a state as input, and returns the next valid transition. In case of no valid transitions, it executes *halt* statement, halting the simulation. In case of multiple valid transitions, it prints out an error message with the current state information, and executes the *halt* statement.
- The performTransition() function performs everything relevant for taking the a transition. This is reached *only if* getValidTransition() returns a non-null transition. It takes a transition as the input, finds the lowest common ancestor between the source and destination states of the transition, and follows the path from *source –> lowest common ancestor –> destination*. While doing so, as it moves from *source –> lowest common ancestor*, it executes the *exit* statements corresponding to the states in the path. It also updates the history_map accordingly for the states being exited(and which have the history marker). After this, it executes the *action* statements corresponding to the transition. Having done that it follows the path from *lowest common ancestor –> destination*, while executing the entry statements for all the states in the path.

Algorithm 3: performTransition Algorithm

```
Result: Performs all the necessary steps related to taking a transition
source <- t.getSource();
destination <- t.getDestination();
lub <- lub(source, destination);
current <- source;
while current is not lub do
    execute statement -> current.exit;
    currParent <- curr's parent state;
    if currParent maintains history then
        set the value of currParent in history_map to correspond to curr;
    else
        do nothing;
    end
    curr <- currParent;
end
execute statement -> t.action;
path <- empty list;
current <- destination;
while current is not lub do
    path.append(current);
    current <- current's parent state;
end
i <- path.len - 1;
while i > 0 do
    execute statement -> path[i].entry;
    i –;
end
```

TEST-CASES

The test-cases are essentially StaBL code which were used to test the simulator. However, the purpose of these were not just for testing the simulator but to also develop a better understanding of Statechart modeling, and StaBL as a language.

Most of the test-cases are simplistic and well documented in the code-itself hence, we shall not be re-iterating over the same. Here, we will discuss 1 test-case provided as *sim10.stb* (*.stb* is just to denote that it is StaBL code).

Model Banking System

*Changes were made in the *Simulation loop* to incorporate for obtaining user-input. However, this was only meant for the purpose of demonstration and we do not aim to model external interaction in this manner.
*In terms of functional implementation, *State History* as discussed above has been implemented through *HISTORY* variable. In a similar manner, user-input has been implemented by storing the value in a variable named *VARIABLE*.

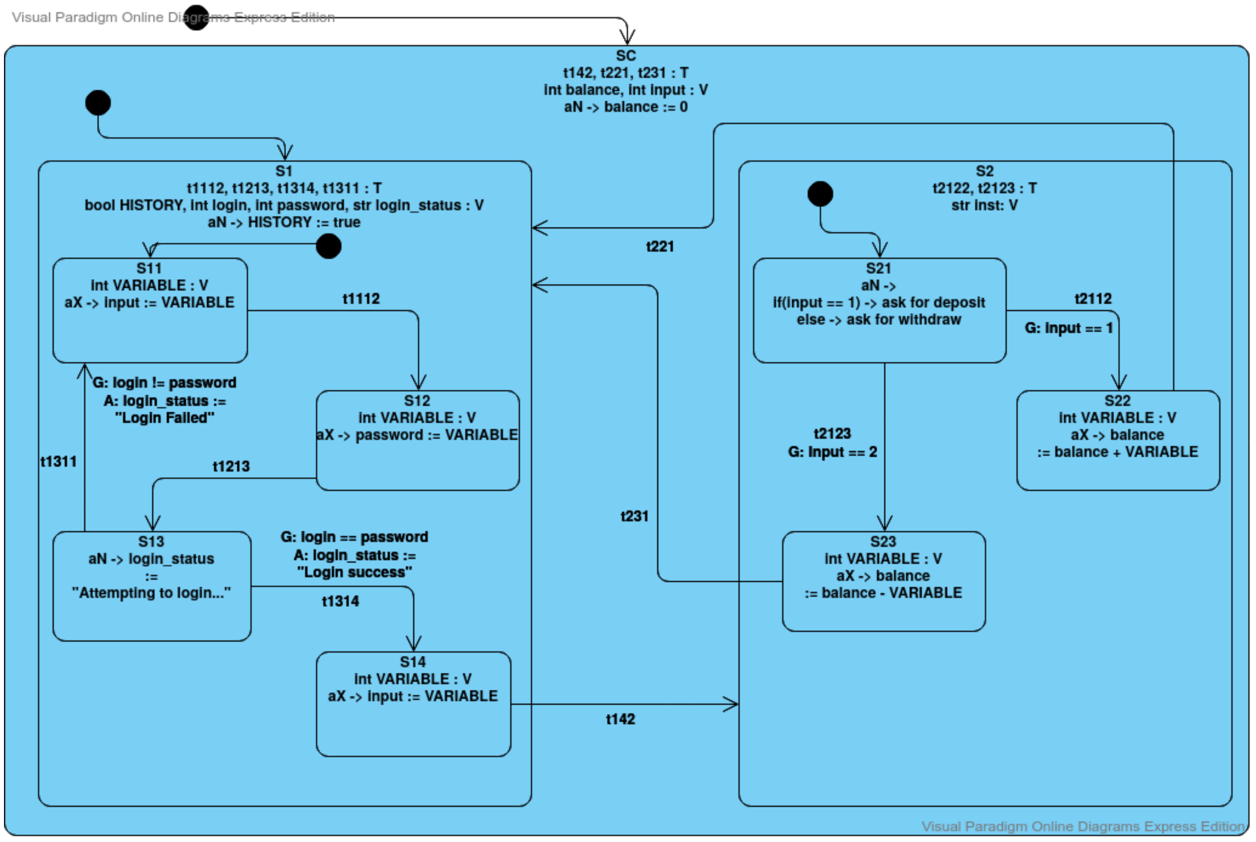

Figure 2: Statechart Diagram for the Model Banking System

The aim was to verify how different Simulator functionalities work together, and to do so while modeling a meaningful (at least to some extent) Statechart.
The following functionalities were modeled:

- The banking system takes in the login and password from the user, verifies them (the verification method here is only to see whether login == password).
- If the verification succeeds the user is allowed to deposit or withdraw into the account. The default balance on every run of the Statechart is set to zero.
- The only data-types used are int, bool and strings. The input taken from the user is an int at all times (balance and deposit are integer amounts, ID and Password

are integers and input is taken to determine whether user has to deposit (input=1) or withdraw (input=2)).

- **T** determines all the transitions declared in that state.
- **aN** determines the entry statements of that state
- **aX** determines the exit statements of that state
- The state name is right at the top (ex: SC, S1, etc)
- **V** determines all the variable declared in that state, here we have also mentioned the variable type along with their declaration.
- **G** determines the *guard* condition for a transition (for every transition where it is not determined, it has been set as *true*)
- **A** determines the *action* statements for a transition
- **:=** symbol determines variable binding
- Black-fill circles determine the default start-state.
- The name of transitions have been kept according to their source and destination state, ex: t1112 makes a transition from source: t1 to destination: t2.

The banking system has been broken down into two major states, the *S1* state deals with user login and input, the *S2* state deals with the amount of money to be dealt with. The variable sharing has been kept to a minimum and transitions between *S1* and *S2* do not go to an atomic destination, i.e. if *S1* and *S2* are modelled separately, modelling of *S1* can be done without knowing the exact functioning of *S2* but only having an abstracted view of its logic (and vice and versa). Once the user has logged in successfully, we do not want the system to prompt them again, this has been ensured through *State History*. Having succeeded in the login is the only scenario when *S1* state is exited. In our Statechart, *S1* has the history marker, and whenever it is returned to, the simulator is directed to a sub-state which deals with user-input, instead of the default sub-state(which deals with login). The guard statements are used to regulate the transitions which are to be picked by the system. Apart from that, a few variable usage might seem unnecessary (example: login_status, balance as a global variable, etc) but that is to demonstrate the complete working of the Statechart simulator. This has also been the reason behind the division of action between *entry*, *exit* and *transition action* statements, usage of 'conditionals' within statements, etc, as we try to build a model cum comprehensive test-case for our Simulator.

Other Test-Cases

- **sim1.stb**: simplest test-case, models the switching on-off of a switch
- **sim2.stb**: test-case for numeric-binary expression evaluation (addition, subtraction, multiplication, etc).
- **sim3.stb**: test-case for having local-variables with the same name to demonstrate the scoping rules corresponding to states and transitions (for static variables).
- **sim4.stb**: test-case for testing conditional statements.
- **sim5.stb**: test-case for testing looping constructs.
- **sim6.stb**: test-case for testing *State History*
- **sim7.stb**: same as **sim6.stb** but without maintaining history, to see the difference
- **sim8.stb**: test-case for nary expression evaluation (Note: The current expression evaluator does NOT consider expression precedence and hence, requires explicit use of brackets. The current algorithm randomly splits an nary expression into binary expressions and then solves the problemThe performTransition() function performs everything relevant to the current tran- bottom-up.
- **sim9.stb**: test-case for analyzing *State History* along with various inter-level transitions.
- **sim11.stb**: test-case for detecting non-determinism.

FUTURE WORK

Statechart has been widely accepted on the industrial front and it bridges the crucial gap in system-modeling of an easy-to-understand representation, while incorporating various methodologies for verification. StaBL aims to bolster the latter, by providing for formal mathematical verification constructs for Statechart modelling. A crucial step towards the same shall require a well-engineered Simulator. We shall continue the work forward to provide for the same. Our immediate next steps would involve enhancing the Simulator to provide for the remaining functionalities of StaBL (like History Marker, different types of Variables, etc). We shall build upon our long-term goals as we proceed.

REFERENCES

Sujit Kumar Chakrabarti and Karthika Venkatesan. 2020. StaBL: Statecharts with Local Variables.