Formal Specification with Statecharts

Karthika Venkatesan, IIITB
Sujit Kumar Chakrabarti, IIITB

Abstract

UML Statecharts have enjoyed widespread popularity in software development community as a design and specification notation. It has primarily been employed for specification of object life-cycles. In this paper, we explore the option of using Statecharts in specifying GUI interactions, typically in web applications. A notation like Statechart is very apt to be used as a formal specification language, provided the following features: Firstly, it should have well-defined semantics, so that specifications written in it can not be inherently ambiguous. Secondly, it should have tool support for automatic detection of specification bugs. In this work, we present enhancements and customisations to the Statechart syntax and semantics for specification of modern web applications. We also demonstrate how automatic detection of specification bugs can be performed through well-known model-checking and program analysis. We have used our specification approach in specifying a number of web applications. Our approach enabled early detection of several specification bugs and hence facilitated higher quality specification.

1 Introduction

Statechart was proposed a visual formalism for specification of complex systems (cite Harel). In particular, the 'system' here is an object with a complex life-cycle going through a finite number of discrete *states*. The notation has enjoyed a wide acceptance in software engineering community. It has been a part of the UML suite for modelling object oriented system.

Application of Statecharts as a specification notation for web-based systems was explored in (cite Ricca-Tonella paper). Features of web-based systems have evolved significantly since the time of this paper. Predominantly plain HTML forms have given place to highly interactive user-interfaces, e.g. tabbed panels, with rich client side

validations as well as AJAX which permits partial page updates via asynchronous server interactions. This calls for a further look at how Statechart notation could be adapted to allow specification of richly featured modern web applications.

Another angle to look at is the growing complexity of web applications. This calls for increased levels of automation in engineering these systems, in particular, requirement analysis and verification. To achieve this goal, it is more important for specification formalisms to have a formally defined semantics, than to have rich syntactic features. A formal specification language would open doors to a variety of automated approaches to analysis, design, generation of both code and tests for modern web based systems.

Contributions of this work include:

1. Syntactic and semantic adaptation of Statechart notation for formal specification of GUI intensive applications, typically web applications
2. Illustration of specification defects possible in specification written using this notation
3. Static analysis methods to detect the above defects automatically.

2 Statechart Language

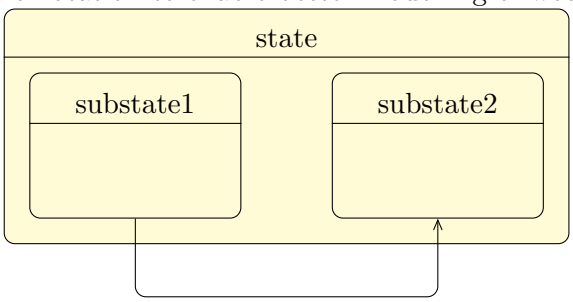
Statechart provides a pictorial notation to describe the life-cycle of a system, usually object-oriented. It is an adaptation of the classical finite state automation with enhancement like hierarchical states, concurrent states, action language, communication etc., all aimed to increase its expressive power as a specification language of industrial strength software systems.

Statechart has been used widely in industry and academia as a part of various UML modelling tools [Rational Rose, Rhapsody, Telelogic, Statemate, Stateflow]. In all industrial tools, the explicitly specified semantics of the language, if at all given, are semi-formal to the best of our knowledge. There has been a lot of research in formal specification of semantics for Statechart(-like) languages [J. Rushby paper in FASE'04 on Stateflow operational semantics etc.]. They mostly target specific subsets of the Statechart language. We take a similar approach, defining formal syntax and semantics of the Statechart language for parts which we have found useful in formal specification of web applications. Mapping various existing features of the Statechart language to specification use-cases and formulating their formal de-

scription and building verification support for the same is a part of our ongoing research agenda.

3 Example

In this section, we informally introduce, through an example, the specific features that have been added or modified to the existing Statechart notation to enable better modelling of web applications.



3.1 Syntax

```
type statechart =
  state      *    (* initial state *)
  state list *    (* other states *)
  transition list (* transitions *)

type state =
    AtomicState of
      name *  (* State name *)
    | block * (* Entry actions *)
    | block   (* Exit actions *)
  | NonAtomicState of statechart
  | ShallowHistory
  | DeepHistory
  | Branch

type transition = Transition of
    id * (* source state name *)
    id * (* destination state name *)
    expr  * (* trigger *)
    expr  * (* guard  *)
    block   (* action *)

block = stmt list (* statements *)

stmt = Assignment of
         expr * (* LHS *)
         expr   (* RHS *)
  | Expr of expr (* independent expressions like
    arithmetic expressions or function calls *)
  | Return of expr (* Returned value *)
  | If of expr  * (* boolean condition *)
          block * (* then block *)
          block   (* else block *)
```

```
  | Loop of expr * (* boolean condition *)
            block  (* loop body *)

expr =
    BinaryExpr of expr    * (* LHS *)
                  binop * (* Operator *)
                  expr      (* RHS *)
  | UnaryExpr of unaryop * (* Unary operator *)
                 expr
  | FunctionCall of expr * (* Function *)
                    expr list (* actual parameters *)
  | Id of string
  | string_literal of string
  | num_const of string
  | PHI (* empty set *)
  | Set of expr list
```

Statechart specifications can be partitioned into their core Statechart part, and the trigger/guard/action part, typically abbreviated as the action part. The core Statechart part of the specification gives the pictorial form and finite-state nature to the specification. The action part, essentially comprised of code fragments in an appropriate imperative programming language, adds arbitrary expressiveness to the Statechart, allowing it to be infinite state in reality. In our adaptation, we have not made any modification to the core Statechart language w.r.t. Harel's notation, except – for the time being – postponing the treatment of some of its advanced features. However, we have provided more detailed specification of the action language part, some of them in variance with existing norms. These specifications have been accommodated to align the notation more closely to the needs of specifying GUI intensive applications in general, and web apps in particular.

The salient points of our syntax are:

1. *Set theoretic operations.*
2. *Local variables.*
3. *Input and output variables of states*

Some of the features which we retain are:

1. Hierarchy
2. triggers/guards/actions
3. History states

The important elements of Statechart which we have not considered so far are:

1. And states
2. Message passing

3.2 Semantics

As mentioned earlier, our emphasis has been on the specification of the action language. Our action language has the following semantic features.

1. *Lexical scope.* Events and variables declared in a state are visible within it, including the substates. Internal bindings shadow the external bindings.
2. Action on a transition are allowed to use local events and variables of the source state and define any variables of the destination state. This feature allows data-flow between successively visited states without recourse to global variables.

4 Specification Bugs

Like programs, requirement specifications are subject to faults [1]. One of the most important goals of using formal methods is to detect such bugs automatically. Requirement bugs can be further classified as *implicit* and *explicit*.

Implicit bugs are violations of universal properties. In programming, universal properties abound. Examples of violations of universal properties are accessing undefined variables, null-pointer dereferencing etc. Such bugs would most likely lead to catastrophic failures, like a program crash. Hence, modern compilers come with built-in support to detect many of these errors; programmers do not need to specify them explicitly. Such analysis is often based on static program analysis methods.

Explicit bugs are violations of properties explicitly stated by the programmer, often arising out of the business domain, often specific to the system being modelled. For example, for a banking software, the `balance` attribute of a `BankAccount` class may never be allowed to hold a negative value. Such properties need additional analysis typically based on model-checking or theorem proving technology.

We have encountered a number of requirement bugs that could occur in a Statechart specification model which could be classified as implicit and explicit, in a sense very similar to the above. And interestingly, it turns out that the verification technologies applicable in detection of these bugs also correspond to the above: program analysis for implicit bugs, and model checking and theorem proving for explicit bugs.

[1] For our discussion, we use *faults*, *defects* and *bugs* as synonyms.

Figure 1: Undefined variable

Further subsection of this section present some of the properties – both implicit and explicit – which could get violated in a specification leading to requirement bugs. In section 5, we explain how they can be detected using static analysis techniques.

4.1 Undefined Variable Access

Consider the Statechart fragment shown in figure 1. On running static analysis on this model, several warnings are flagged, e.g. `roomNumber` for a student may turn out to be `nil` in a number of expressions where it is used. At several places, this corresponds to a serious flaw in the specification model. Room allocation can be done in two ways: either by first selecting a student and then going to the room allocation, or by first selecting the room and then going to the student selection. Once this process is completed, the composite state `AllocateRoom` terminates. Here, it is possible to complete the room allocation without actually completing the process.

4.2 Non-Determinism

Statechart permits non-determinism. For instance, when multiple outgoing transitions from the current state are enabled simultaneously, the system must make a non-deterministic choice between them. Different flavours resolve non-determinism in different ways (cite various Statechart semantics papers).

Non-determinism is an instance of incompleteness in specification. Whether it is desired in the specification or not depends on the specific case. In any case, it would be desirable to be able to detect their presence automatically. For any configuration, the set of *conflicting outgoing transitions* is nothing but the set of the outgoing transitions from all its states. To statically detect non-determinism, we need to check if for any given configuration, any two conflicting outgoing transitions can get enabled simultaneously.

4.3 Stuck Specification

Another problem is that of *stuckness*: when there is no way left for the system to exit a particular configuration. Consider the simple Statechart fragement in figure 2

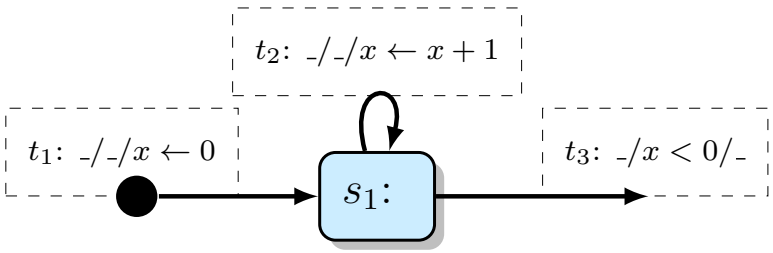


Figure 2: Stuck specification: The system can not exit s_1

4.4 Violation of External Specifications

Based on the specific domain, there may be any number of additional constraints which we may want our specification to follow.

5 Verification

In this section, we outline a number of analyses that can be used to detect the specification bugs presented in section 4.

6 Conclusion and Future Work

Concurrent states (And states) are a very important feature of Statechart notation which we have not considered so far. We realise that it is very relevant for specification of web applications which themselves have a large amount of concurrency built in. In the next stage of our research, we are working on incorporating concurrency into our adaptation of Statecharts. This feature – apart from adding expressiveness – will also make a number of additional specification bugs possible. Our next effort will be to augment our verification tool set for automatic detection of the above bugs.

Wherever there was a translation required for preparing an input to a verification tool, it was performed manually. This is tedious and error prone. There exists a wealth of prior work in this direction (cite Translation of Statechart to Model Checking). We are currently in the process of implementing some of these translators, so that the translation can be automated.

In this paper, we have focused on static verification techniques. However, Statechart specifications are an excellent starting point for automated test generation. In our earlier work, we have reported an end-to-end technique to generate test cases for web applications from Statechart specification (our ModSym 2016 paper). With addition

of further features to the Statechart notation, it will be necessary to extend this test generation method to accommodate these.