Practical Programming, Validation and Verification with Finite-State Machines: a Library and its Industrial Application

Paulo Salem
São Paulo – Brazil
paulosalem@paulosalem.com

ABSTRACT

Finite-state machines (FSMs) are among the oldest models employed in the formalization and analysis of both software and hardware. Owing to their simplicity, there exist various implementations to support their practical application in mainstream programming languages. Through such software libraries, programmers can explicitly define states, events and transitions in order to delegate the machine's execution to an underlying engine. However, as far as we know, no such library provides formal verification capabilities alongside its execution engine. That is to say, once an FSM is defined, the resulting program cannot be used as a formal specification to be subject to formal verification, thereby not making the analytical tractability of FSMs available. Formal verification, if any, is conducted in an independent model separate from the program, thus duplicating the information and creating the possibility of discrepancies between both. In this paper we show how to overcome this limitation. To this end, we present the VERUM library, which allows the specification, execution and verification of FSMs in the Ruby language, largely bypassing the need to explicitly employ an additional modeling language. Formal verification is achieved by automatically translating the source program of the FSM into a Timed Automaton (TA) specification for the UPPAAL model checker. To illustrate the value of the approach, we present the industrial problem which inspired the creation of this tool and is currently using it, namely, a payment system. Besides the technical aspects of the tool, a number of practical lessons learned with this application are explored. Although we describe very concrete artifacts and applications, the overall approach is quite general.

CCS Concepts

•**Software and its engineering** → **Formal software verification;** *Frameworks; Software libraries and repositories;* Model checking;

ICSE '16 Companion, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2889160.2889226

Keywords

finite-state machines, lightweight formal methods, runtime verification, Ruby, UPPAAL, payment systems

1. INTRODUCTION

Finite-state machines (FSMs)[1] are among the oldest models employed in the formalization and analysis of both hardware and software (e.g., Parna's 1969 Terminal State Transition Diagrams [18]). Incredibly, as far as we know there is no support for the use of FSMs' verification capabilities in mainstream languages and frameworks (e.g., C, Java, .Net, Ruby). At best, there are libraries to *implement* FSMs, hence allowing programmers to benefit from their simplicity, but which provide no direct support for their formal verification or even simpler kinds of analyses. This is surprising because a lot of the value of the model lies precisely in its formal and analytical tractability. Indeed, this tractability is at the heart of Model Checking [3], a successful and well-know formal verification technique. This method, however, depends on the prior definition of a formal specification, which is independent of the program that must be developed to satisfy it later (i.e., a top-down approach). Our concern, here, is in making such verification capabilities available directly at the program level, without a separate specification, to be used as any other library that a programmer would normally employ in developing a system (i.e., a bottom-up approach).

FSMs can be used when modeling simple, but critical, systems. For this reason, we chose to employ an FSM when implementing the payment system of a commercial application. At first, we developed just an FSM execution engine, but as errors were found later in the specification of the payment systems, it soon became clear that it would be useful to enrich the FSMs with verification capabilities as well.

In this paper we present our solution to this problem and an actual application in a commercial software. The solution consists in a library, called VERUM[2], which allows the programmer to define FSMs directly in the source code. Once defined, these FSMs can be both executed and transformed in formal specifications. The current implementation is capable of generating a visualization of the machine and, more importantly, a specification in the form of Timed Automata

[1]Until Section 3, the term *finite-state machine* refers broadly to any model that consists mainly in explicit finite sets both of states and transitions among these states. There are many historic variations of this basic idea. Thenceforth, we introduce and formalize a version used in our tool, to which the term then refers.

[2]As a convenient abbreviation of *Verifiable Ruby Machines*.

(TA)[2], suitable for verification with the UPPAAL model checker [4]. The latter includes not only the various states and transitions, but also the actual preconditions that guard these transitions, including real-time constraints, thereby creating a faithful formal representation of the FSM. The library can be downloaded and used under an open-source license.[3] It is implemented in the Ruby language [17], but of course the principles on which it is based can – and should – be ported to other languages.

The value of this contribution lies considerably in the fact that we developed it as a response to a practical, commercial, need. It therefore incorporates mechanisms sufficient to a useful industrial application, a purpose whose satisfaction would otherwise be unclear. Importantly, we show that while a little sophistication is required, it is overall a simple approach, and therefore a cost-effective effort.

This text is organized as follows. Section 2 comments on related work. Section 3 presents VERUM, including the translation strategy employed to render TA specifications. Section 4, in turn, outlines the methodology that best suites the presented technology. To show the relevance of the approach, Section 5 explores our industrial case study. Finally, Section 6 concludes.

2. RELATED WORK

The use of FSMs in software development has been essentially of two kinds. First, the description of an abstract FSM as a specification (usually graphical) which is later implemented as an actual program. This is the approach taken notably by the Harel's Statecharts [12], which today are found incorporated in the UML standard [14]. There are tools, such as Apache Commons' SCXML [10], that can directly execute these charts and connect them to other software. There are also simpler FSM compilers such as SMC [20]. Another way to use such specifications is to derive other validation and verification artifacts automatically from them (e.g., test cases [21]).

The second kind concerns the direct implementation of FSMs, without a prior separate model. Execution engines have been implemented so that programmers can directly specify states, events and transitions, and have the resulting program to behave as an FSM (e.g., [9, 19]). As far as we could determine, however, none of the existing tools provide verification capabilities in addition to their execution engines.

There is, of course, a plethora of more powerful formal methods that can be used to achieve similar aims as far as expressiveness and verification are concerned. Approaches such as the Z Notation [23] and the B Method [1] allow abstract logical specification and gradual proof-based formal refinements until an executable program is produced. Model checking tools, such as NuSMV [7], PRISM [13] (for probabilistic systems) or UPPAAL [4] itself, can be used directly to model and study the system of interest, although the actual implementation must then be developed independently and shown to be conformant. Most of these tools, however, assume a strict top-down design and require different artifacts in each step: an abstract specification is defined and then it is refined until an executable program is reached; the specification and the program are two different artifacts that must be managed independently, which means

[3] `http://github.com/paulosalem/verum`

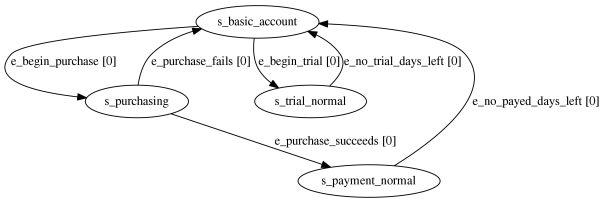

Figure 1: A simplified version of the main case study presented later. States (bubbles) and events in transitions (arrows) are labeled. The numbers after event names indicate their priority. Other constraints (i.e., invariants, preconditions and updates) are not shown, but are defined in the corresponding VERUM program.

that changes in one may not be reflected in the other. In contrast, VERUM is designed so that the barriers between a program and its formal specification become minimal and easily handled by common, non-specialist, programmers in a bottom-up manner. A VERUM FSM is completely defined in the program itself, reuses many of its components both for execution and for verification, and can be gradually enriched with more information that can be used in its formal verification. For example, a fully executable FSM can be defined without specifying any state invariants, which can be gradually added later according to the practical needs and possibilities of the programmers.

VERUM can be seen as a lightweight formal method. In this sense, it has some similarities with Runtime Verification (RV) techniques [11, 15], and in particular with Monitoring-Oriented Programming [6]. Like in RV, a VERUM program has information that allows its verification during execution, and violations of the specification (e.g., invariant violations) are reported. However, unlike RV, more sophisticated properties about the overall behavior of the system (e.g., temporal logic formulas) are not verified during execution in VERUM; rather, programs are supposed to be translated into the input of a model checker and be subject to traditional, exhaustive verification. Therefore, the types of problems and guarantees that each technique can handle are different.

Finally, although a central design principle of VERUM is to reduce the amount of non-executable specifications that must be defined, at certain parts this is not fully achieved and the programmer is advised to add non-executable information to aid in verification. In this respect, the approach is similar to the Java Modeling Language (JML) and similar techniques [16], in which a program is annotated with a behavioral specification that is used as the basis for its verification (that can be carried out either during runtime or statically).

3. THE VERUM LIBRARY

Although in its current implementation VERUM ultimately uses TA as the formal model for verification, in principle the technique could be adapted to other formalisms. Moreover, as it is explained in Section 3.2, the desired behavior for the FSM can be slightly different from what would normally occur in a TA. Therefore, we have opted to: (i) separate the formal definition of the Finite-State Machine (FSM) used

in the execution engine from the formal model used in verification; and (ii) handle any discrepancies between the semantics of both in the translation. In VERUM, an FSM is formally defined as follows:

DEFINITION 1 (FINITE-STATE MACHINE). *Let* Exp(X) *be the set of all Boolean expressions that can be composed with the variables in* X. *Then a* Finite-State Machine (FSM) *is a tuple* $\langle V, S, s_0, inv, F, E, pre, update, T\rangle$ *such that:*

- V *is a set of* formal variables *with values given in some set* D. *Both sets can be partitioned in subsets* $V_1, ..., V_n$ *and* $D_1, ..., D_n$ *such that* D_i *contains the possible values of variables in* V_i *(i.e., variables are typed);*
- S *is a set of* states *and* $s_0 \in S$ *is the* initial state*;*
- $inv : S \rightarrow Exp(V)$ *is an* invariant labeling function *that assigns a Boolean invariant to each state;*
- $F \subseteq S$ *is a (possibly empty) set of* terminal states*;*
- E *is a set of* events*;*
- $pre : E \rightarrow Exp(V)$ *is a* precondition labeling function *that assigns a Boolean precondition to each event;*
- $update : \bigcup_{i=1}^{n}(V_i, E) \rightarrow D_i$ *is an* update function *that assigns new values to each variable at each event;*
- $T = S \times E \times S \times \mathbb{Z}$ *is a set of* transitions *with* priorities.

This definition is an abstraction of the structure of VERUM machines (which are Ruby programs), intended to allow one to think about them in mathematical terms whenever convenient. In practice, however, a programmer simply writes an FSM as a Ruby program, which is then either executed or translated to the formal model of the underlying model checker (currently UPPAAL's TA).

In Section 5 we will present a case study in which a real payment processing system is modeled and verified. In the present section, however, we employ a simplified version of this system as a running example in order to introduce the main concepts of the library. In this simple machine, a user begins with a free basic account for a service and is allowed to purchase a premium version, with the possibility of a free trial for a limited time before the purchase (see Figure 1).[4]

3.1 Architecture

The main architectural principle adopted is to allow the same program structures to act both as executable programs and as formal specifications. The programming language is seen as a general expression substratum, which allows both imperative commands and abstract constraints to be defined.[5] When the program is running, the library merely executes the FSM and monitors whether invariants hold. The interesting part, though, is that the same definitions that allow such an execution can be transformed in the input of a model checker (or other similar verification or validation tool). In this manner, the need to provide a separate formal specification is largely eliminated. The most significant cost left becomes that of writing the properties to be checked. Below we present each of the relevant program structures that implement the elements of Definition 1 to achieve this design. The FSM is assembled by creating a subclass of the provided `Verum::FiniteStateMachine` class and then, within its constructor (i.e., the method `initialize`), making the appropriate method calls to establish these structures. How the FSM thus created is translated to UPPAAL's formal model is left for Section 3.2.

3.1.1 Variables

The translation from VERUM programs to formal specifications depends crucially on the definition, within the programs, of *formal variables* (i.e., the set V of Definition 1) using VERUM's `let` method. These create a bridge, as it were, between the execution of a program and its formal specification. Consider the following declaration.

```
let :never_payed, type: :boolean, init: true
  do
    # Calculation of the actual value of the
    # variable comes here. E.g., check a
    # database to see the user's purchases.
  end
```

On the one hand, during execution, a method named `never_payed` is made available to the programmer and computes whatever is specified in the `let` block. On the other hand, during translation to a formal specification, the name, type and other declared information are used to generate appropriate TA variables. VERUM currently support four types of variables, namely:

- *Boolean.* Declared just like in the above example. If no initial value is specified, VERUM ensures that both true and false are considered during verification.
- *Integer.* Also requires the definition of lower and upper bounds (e.g., `let :v, type: :integer, min: -20, max: 20`).
- *Enumeration.* Defined by an array of possible values (e.g., `let :v, type: :enumeration, values: [:a, :b, :c]`)
- *Chronometer.* Defines a TA clock (e.g., `let :v, type: :chronometer`). In practice, this can be used to model variables such as, in our working example, how many days have been used from a monthly subscription.

In relation to Definition 1, these types correspond to the set D partitioned in four subsets.

3.1.2 States

Each state is declared with the `state` method, through which it is named (e.g., `s_trial_normal`) and an optional customization block can be passed. Within this block, a *builder* object provides the programmer with customization facilities.

```
state :s_trial_normal do |s|
  s.on_entry do
    # Executable program comes here. E.g.,
    # update the database and send an email.
  end

  s.formal_invariant do
```

[4]The program for this simpler version is included as an example with the library and, therefore, can be freely downloaded and examined more fully.

[5]Ruby, incidentally, proved to be an excellent medium, since the resulting programs are very readable.

```
    # Formal invariants come here. E.g., days
    # in trial are bounded by a maximum.
    trial_days_used <= trial_days
  end

  s.formally_force_progress = true

  # Not used in this example:
  #  s.submachine = another_machine
  #  s.terminal
end
```

States thus defined correspond to the set S of Definition 1.

State effects.

The builder's `on_entry` method allows the programmer to define what is to be executed when the machine enters in the state. This is how the machine is programmed to have an effect on the rest of the system and does not affect formal verification in any way. Note that, accordingly, this part of the program does not have a representation in Definition 1 and is thus visible only to the execution engine.

Invariants.

The builder's `formal_invariant` method, in turn, specifies a Boolean expression that must be always true. It affects both the execution of the machine and its formal verification. In the former case, invariants work as safeguards against potential errors introduced through the execution of `on_entry` blocks: the VERUM engine checks, at every state entry, whether the state invariants holds after the corresponding program execution. In the latter case, the expression is translated to a UPPAAL expression and is defined as the invariant of the location in the resulting UPPAAL formal specification. There is no need for the programmer to write a separate specification: it is the very same program that is used for both execution and verification. However, to do this, the expressions allowed are actually a subset of Ruby expressions (see Section 3.2 below). Notice also that although invariants strengthen the reliability of the machine, they are optional. Invariants thus defined correspond to the function *inv* of Definition 1.

Force progress.

Normally, states will be required to progress (see discussion on *Progress* in Section 3.2). If this is to be avoided, for instance because the user can stay forever in a state, then `formally_force_progress` should be set to `false`.

Sub-machines.

Finally, the builder's `submachine` method allows the specification of a sub-machine to run in this state. This effectively allows the composition of larger machines out of smaller ones. During execution, this means that a state with a sub-machine will only be able to advance if the sub-machine has reached one of its terminal states (declared via the `terminal` method in the builder; correspond to the set F of Definition 1). For verification purposes, however, such sub-machines are abstracted away in the present version of VERUM: each machine used must be validated and verified separately.

3.1.3 Events

Similarly to states, each event is declared with the `event` method. Together, these events correspond to the set E of Definition 1.

```
event :e_purchase_succeeds do |e|
  e.formal_precondition do
    # Restricted Ruby expression here. E.g.,
    # some payment must succeed.
    credit_card_ok || transfer_ok
  end

  e.formal_update = {payed_days_used: 0,
                     never_payed: false}
end
```

Formal preconditions.

Preconditions guard events. When an event's precondition is true, the corresponding transition may happen; otherwise, it cannot happen. As with state invariants, preconditions are Boolean Ruby expressions which are handled transparently both in execution and in verification. For the same reasons, expressions allowed as preconditions are a subset of Ruby expressions (see Section 3.2 below). Preconditions defined programatically correspond to the function *pre* of Definition 1.

Formal variable updates.

TA allow the values of variables to be updated after transitions. This permits one to constrain the values that variables may have along the various execution paths, and therefore make the specification more precise. VERUM provides the `formal_update` builder method by which the programmer can use this feature to improve the formal specifications of his or her machines. The importance of the feature in the technique proposed here is that VERUM cannot determine what the machine is actually doing when it enters a state (i.e., what is being done within the various `on_entry` methods of states, and in particular how program variables are being changed), since the programmer is allowed to specify any program to run within states (i.e., VERUM's engine enforces the transitions among states, not what happens once a state is reached). Unlike invariants and preconditions, which are specified only once and automatically translated to the underlying formal model, here the programmer does have to duplicate the information. That is to say, besides the actual assignments within various `on_entry` methods, the program can be further enriched by explicitly declaring some of these assignments through the `formal_update` method. This duplication of information is an important limitation of the proposed method, but nevertheless allows more precise formal specifications. Formal updates thus defined correspond to the function *update* of Definition 1.

3.1.4 Transitions

Transitions are declared through the `transition` method and specify that a state may reach another through an event with a certain (optional) priority, for example:

```
transition :s_basic_account, :e_begin_trial,
           :s_trial_normal, 0
```

The priority is given as an integer (zero by default, made explicit above). VERUM transitions are deterministic, which means that if two transitions are enabled at once with the same priority an exception is raised. Interestingly, we noticed that subtle errors can be corrected simply by adjusting priorities of certain transitions. Transitions thus specified correspond to the set T of Definition 1.

3.2 Translation

Given an FSM program, VERUM can translate it to the input notation of UPPAAL, which implements the theoretical construct of Timed Automata but also introduces idiosyncrasies specific to the model checker (e.g., priorities) [4]. Here, it suffices to say that in UPPAAL a TA is a process that defines *variables* (including *clocks*), communication *channels*, *locations* and *transitions* among these locations. Locations can be labeled with *invariants* and transitions can be labeled with various constructs, including *guards*, *assignments* and *synchronization* clauses. The minute details of the translation procedure can be examined directly in the implementation available for download and require understanding of the UPPAAL conventions. Here we present only its main characteristics.

The translation begins with the definition of a main UPPAAL process to represent the FSM, called simply *Process*. As far as syntactically possible, names of states, variables and other elements are unchanged. Hence, a variable called x can be referred to as *Process.x*.

States.

Each state of the FSM is translated as a TA location enriched by the invariant specified in the state (see *Preconditions and invariants* below for details). For the sake of illustration, let us consider what the UPPAAL translation of the the state `s_trial_normal` seen in the example above looks like:

```
<location id="s_trial_normal">
  <name>s_trial_normal</name>
  <label kind="invariant">
     (d &lt;= 100000000) and
     ((trial_days_used &lt;= trial_days))
  </label>
</location>
```

It is an XML[6] element with the same information and an extra invariant (using a variable `d`) which is explained in *Progress* below. In the next subsections the actual XML translations are not shown, but the reader interested in these details can refer to the source code available for download.

Events and Transitions.

Transitions are modelled as TA edges between locations and enriched by the following constructs:

- The *precondition* of the event in a transition is transformed in a *guard* of a TA edge (see *Preconditions and invariants* below for details);
- The *update* specified by the event in a transition is transformed in an *assignment* clause of the TA edge;
- The *priority* of the transition is transformed in a *synchronization* clause of the TA edge, in which a special channel is used (see *Transition priorities* below).

Preconditions and invariants.

In order to translate an event's precondition or a state's invariant, VERUM accesses the related source program and builds its abstract syntax tree (AST). Then, it recursively translates the AST to UPPAAL notation elements. In order to make this procedure feasible, the invariants and preconditions must be defined as expressions using a limited subset of the Ruby language, which at present consists of the following elements:

- References to the formal variables defined previously, which are actually method calls;
- Logical constructs: *or* (||), *and* (&&), *if expressions* (`if ... else ... end`), `true` and `false`;
- Comparators: `<`, `>`, `<=`, `>=`, `==`, `!=`;
- Arithmetic operations: `+`, `-`, `/`, `*`;
- The special method `any()`, which can be applied to formal variables, and makes it explicitly that the variable can take any value (in execution this method simply returns `true`).

The set of all such expressions correspond to the set $Exp(V)$ of Definition 1. Note that this includes complex expressions formed by nested logical constructs and employing multiple variables, comparators and operations.

If the programmer employs an unsupported construct, an exception is raised. Though limited, in our experience this proved to be a sufficient vocabulary for useful applications (see Section 5).

Progress.

The normal TA semantics allows an automaton to stay forever in a state, even if there are transitions available. However, we designed VERUM to always take a transition when one is available. That is to say, the system always tries to *progress*. To approximate this behavior in a TA, the translation to UPPAAL ensures that any non-terminal state eventually reaches a timeout and the automaton is forced to progress to another state (unless the programmer explicitly disables this provision). This is achieved by: (i) defining a special clock, d, and assigning the invariant $d <= max\_delay$ to every non-terminal state, where max_delay is a user-defined constant; (ii) and resetting d to 0 on every transition.

Transition priorities.

A separate UPPAAL process called *PriorityEnforcer* is defined to run in parallel with the main *Process*. This auxiliary process is comprised of one state, loop transitions and corresponding synchronization channels, one for each priority number used in the FSM. Channels are ordered according to their priorities using the `chan priority` UPPAAL command. Transitions in the main *Process* then obey the priority ordering because they synchronize with a corresponding loop transition in *PriorityEnforcer*.

Properties to be checked.

Finally, the properties to be checked must be translated to UPPAAL notation. The simplest manner in which VERUM allows this is through the method `uppaal_spec(spec)`, which simply takes the given UPPAAL specification `spec` and inserts it in the proper place in the final TA file. This, of course, requires special knowledge of UPPAAL itself, which is an extra burden for the programmer. To mitigate this problem, a number of higher-level methods are also provided. These are explored in Section 3.4 below.

[6]Extensible Markup Language.

3.3 Execution

The FSM object does not change state on its own. To advance, the method `next!` must be called. Optionally, a `context` Hash parameter can be specified. As the name implies, this parameter provides a way to inject contextual information in the machine's execution, which can then be accessed by the machine's states and events. Conversely, during the evaluation of states, the method `output` is made available in order to compute outputs, which can later be accessed by the program employing the FSM object. Finally, the FSM class may override the base method `synch`, which is always (automatically) called before a state is entered. This method is supposed to synchronize the internal state of the FSM with external sources (e.g., a database). Other convenience methods are also provided, but we do not explore them here. If at any point in all this an invariant is broken, an exception is raised – hence a form of Runtime Verification is also performed.

3.4 Verification

As any program, the FSMs thus obtained can be subject to testing: they can be programmatically instantiated, put under certain conditions, and have their behavior monitored. Such is the approach largely adopted by the software industry. Our focus here, however, is in the formal verification capabilities provided by VERUM, which can supplement such traditional testing.

The programmer is kept as close as possible to his or her native programming language. As we have seen above, this is achieved by producing a TA from the program itself that can be subject to Model Checking. The verification thus performed has a number of important characteristics:

- Boolean variables used in preconditions can automatically and systematically be assigned all possible values, which would be tiresome to do manually in testing;
- The results of formal verification are, in reality, over-approximations. Because the verification considers *all* possible paths of the machine, it can find paths that in real execution would never happen. Therefore, the result of the verification must be used as a diagnostic tool, not as a final verdict;
- Verification can be partial. It is possible to gradually enrich the model with more information, even though it is already working normally in execution. This happens because not all necessary information for verification can be derived automatically. Preconditions, it is true, are translated automatically. But updates that represent internal changes caused by the software must be added manually.

These considerations suggest that neither testing nor formal verification alone is enough. Each one has a role and must be allowed to play it.

Temporal properties.

A major advantage of translating FSM programs to the input of a model checker is the reuse of the temporal logic used to make assertions about the FSM. UPPAAL provides a version of *Timed CTL* that allows the expression of reachability, safety and liveness properties.

However, it must be noted that these are subject to severe limitations in the present version (4.1.19) of UPPAAL. The first and most important one is the impossibility of nesting temporal operators. The second problem lies in liveness properties: assertions of the form $A\Diamond\psi$ ("eventually ψ happens") and the derived form $\phi \rightsquigarrow \psi$, called *leads to*, which asserts that whenever ϕ happens, ψ will eventually happen. The latter can be useful, for example, to assert that every purchase has a proper conclusion (e.g., by succeeding or failing). This kind of formula is supported by UPPAAL, but only if transition priorities are disabled. Since our translation requires the use of priorities for a faithful representation of the machine behavior, it follows that we do not have this important kind of property available through UPPAAL. Nonetheless, it is possible to instruct the translation to disregard priorities, which produces an abstraction of the actual behavior of the FSM that can then be checked using liveness properties.

VERUM also provides methods that simplify the task of defining temporal properties. Instead of writing the UPPAAL query itself, programmers may simply call one of the following methods (among others), providing only the state names of interest as parameters:

- `uppaal_all_reacheable_except(exceptions)`: requires that all states defined in the machine are reachable, except the ones specified. Our case study suggests that this is a useful pattern;
- `uppaal_may_repeat_forever(names)`: requires the specified states may be repeated forever;
- `uppaal_inevitable_states(names)`: requires that all states in the array `names` are always reached eventually (i.e., `A<> name`). As pointed out previously, this can only be checked if transition priorities are disabled;

3.5 Visualization

Once translated to the UPPAAL notation, the model can be visualized as a graph within UPPAAL's own user interface. Nevertheless, VERUM also provides a more direct method, independent of any translation to TA, to visualize the FSM. As an example, see Figure 1. Since visualizations provide another perspective to the behavior of the FSM, they are useful mainly as a validation mechanism.

4. METHODOLOGY

The library is meant to be used in two manners:

- *Model to program.* As it is usually recommended, first a model can be developed and only later its programmatic details need to be inserted. VERUM makes this procedure simpler, since the modeling primitives are expressed in the same language as the program that should complement it later, namely, the Ruby language. In this manner, programmers can write the models in the tools they are already familiar with, and only later submit the generated specification to a model checker. Furthermore, the fact that specification and program are all in one single document (i.e., the program file itself) prevents unnecessary duplication of the specification information in, say, a separate file. Therefore, our approach makes maintenance of the system easier, since there is no need to keep two distinct artifacts evolving in parallel.

- *Program to model.* Given an existing FSM already in use, it is possible to gradually enrich it with formal annotations so that a suitable model can be generated and subject to Model Checking.

In both cases, the fact that modeling is achieved programatically is essential. It is interesting to note that such a scheme goes against the traditional workflow for designing FSMs, which consists in doing so first visually, and only later enrich it programmatically. This methodology is implicit, for instance, in UML and related technologies. A library such as VERUM provides the technology for an alternative method.

5. CASE STUDY

The motivation and main application of VERUM is to be found in the payment subsystem of a commercial application we develop, called Liberalis[7], which currently manages more than 2500 users. The application has a free, ad-supported, version, which we call *Basic* and a payed version, which we call *Premium*. Moreover, users may test the Premium offer for free during a *trial* period of 30 days.

From the start we realized that it would be best to design and implement this subsystem as an FSM. But even with this provision, it soon became apparent that the mere design discipline imposed by the use of an FSM was not enough, since errors were later found. Thus, we decided that it would be worth to extract the formal FSM model from the implementation to perform Model Checking.

At first, we tried to extract just states and transitions, without worrying about the preconditions guarding the transitions. However, this simple approach was not very useful, since too many false positives and negatives were generated (i.e., paths that would never happen in the actual machine, owing to the various preconditions, were reported to take place). We were thus led to explicitly model the relevant variables and related preconditions.

The main payment system was implemented as an FSM with 21 formal variables, 14 states, 20 events and 46 transitions. Its full graphical representation, as rendered by VERUM using GraphViz, is shown in Figure 2. This main FSM itself depends on another smaller sub-machine, but formal verification was applied only to the main one, which controls the overall payment process and was the source of our central problems.

5.1 Addressable Problems

VERUM is capable of detecting a number of problems we faced during the development of this payment system. Here are two real errors we encountered and that could have been detected by the tool, as well as another that was actually revealed by it:

- *Unexpected cycles.* The original payment system had an error that would allow a user to receive Premium services without actually ever paying. This error relied on the fact that users are given some extra grace days after their free Premium trial period expires in order to make their purchase. However, if the purchase fails (e.g., because the credit card was declined), the FSM was again sent to the state on which users had grace days and could try to pay again (i.e., *s_trial_grace_period*). As long as payment kept failing, this cycle would continue, effectively providing free services to users. The solution we adopted was simply to add a high-priority transition after a purchase failure that downgrades the user's account to a Basic free one. The following specification now checks that this error no longer happens:
  `uppaal_spec("A[] !(Process.s_trial_grace_period and Process.trial_days_used > 50)")`.

- *Missing transitions.* If the FSM runs late and the user did not wish to make a purchase, he or she could stay permanently in the *s_trial_normal* state, hence violating the limit of 30 free Premium days. To correct this, we added a transition that detects the condition and automatically downgrades the account to Basic (i.e., the *s_basic_account* state). The following specification now checks for this problem: `uppaal_spec ("Process.s_trial_normal and (Process.trial_days_used > 30) -> (Process.s_basic_account)")`.

- *Recovering from failure.* While observing the behavior of the machine at the *s_purchase_failure* state through UPPAAL's simulator, we noticed that, under specific circumstances, the machine would be stuck in this state forever. We traced the problem to an erroneous precondition of an event. After correcting it, the following property was properly verified:
  `uppaal_spec ("E<> Process.s_basic_account and Process.purchase_failed_at_least_once and Process.never_payed and Process.trial_begin_date_nil")`.
  It states that it is possible to reach *s_basic_account* after a purchase fails and other conditions hold. Notice that this property uses a variable called *purchase_failed_at_least_once*, which is an artificial addition to cope with the limited temporal vocabulary available in UPPAAL. As the name implies, this variable is set to `true` whenever a purchase fails.

A number of other properties are defined as well in the system. The objective is, of course, not only to ensure their validity at present but also to make sure that they continue to hold as the system evolves. A simple example is that users may remain forever with a Basic account (`uppaal_may_repeat_forever ([:s_basic_account])`). A more interesting example concerns reachability: all states except *s_suspended* and *s_premium_free_account* must be reachable (`uppaal_all_reacheable_except ([:s_suspended, :s_premium_free_account])`). The first exception arises from the fact that when the application was released a user's account could be suspended, but later it was decided that suspension would no longer take place (the account would simply be downgraded to the Basic version instead). Nevertheless, some users were already suspended, and therefore we kept the corresponding state, although it was no longer reachable. The second exception comes from the fact that some users may have a special, permanently free, Premium account, which is never charged nor downgraded. Finally, other states in the machine *may* be unreachable depending

[7]A web application, located at `http://liberalis.biz`, which allows liberal professionals (e.g., lawyers, psychologists, physicians) to easily create their business web sites.

Figure 2: The full set of states, events and transitions of the main payment system. The number after each event indicates the transition's priority. Preconditions and other details are not shown, but are specified in the FSM's program and are verified.

on how the initial configuration is defined, so some care must be taken when interpreting the result of this verification.

Some of the above properties may seem so trivial as to be unworthy of verification. However, we have found that it is easy to disrupt them easily as changes are performed. Consider, for instance, the addition of a new transition from a state that has already numerous transitions coming out of it (such as $s\_purchase\_failure$ we considered above; see also Figure 2). It is possible to mistakenly assign a priority to the new transition in such a way that it becomes impossible to take other previously feasible transitions. Under such circumstances, reachable states can suddenly become unreachable.

These considerations support the hypothesis that the approach provided considerable and useful extra information about the payment system. Both verification and validation aspects were improved, and at little extra cost, which is a good investment for a system seen as particularly critical within the business. It may be argued that a more traditional formal verification approach could have provided similar or better verification benefits. To this we can say that, while it is true that there are more sophisticated and reliable methods available to handle the presented application, they also require much more knowledge and commitment from developers. Such requirements are often not possible to meet in practice, therefore an approach such as VERUM is quite useful. A number of other general lessons learned during the case study are commented in the next and last section.

6. CONCLUSION

In designing VERUM we have deliberately aimed at reusing existing verification technology instead of developing it ourselves. We have considered a number of alternative model checkers, such as NuSMV [7], but ultimately we settled with UPPAAL because TA in general, and UPPAAL's implementation in particular, allowed the simplest and richest translation we could think of. Nevertheless, there is some mismatch between our intended semantics and that of TA. For instance, while a TA does not have to change the current state even if it has enabled transitions, VERUM requires that enabled transitions are always taken. We overcame this and similar difficulties by encoding special patterns in the UPPAAL translation. This shows that the task is not simple, and while the translation is obviously useful, it does carry the disadvantage of not being a simple copy of the original machine, making it harder to understand if one tries to read it directly. It is also important to note that we did not prove the translation to be correct, but merely relied on intuitive arguments used for its construction and the rather direct correspondence with respect to certain elements (e.g., states of the FSM and locations of the TA). A formal proof, however, would be clearly desirable and is left as possible future research.

The practical benefits of the approach do come with drawbacks, some of which have been explained in the text. Perhaps most important, because arbitrary programs can be inserted in the FSMs (i.e., through the `on_entry` method; see Section 3.1.2), properties that are true in verification may be false during execution (e.g., invariants may be violated during execution). For this reason, VERUM should be used as a tool to increase one's understanding and confidence in a system, not to prove its ultimate correctness.

A number of practical lessons can be learned from the presented industrial case study. Besides the most evident ones commented throughout the paper, the following are also worth emphasizing:

- *Proper level of simplification.* For the sake of simplicity, initially we ignored preconditions and invariants during formal verification, thereby analyzing only state transitions. This simple approach, however, was too imprecise to be useful (i.e., too many false positives and negatives), so we had to add the more sophisticated capabilities described in the paper. This experience illustrates well that, like Albert Einstein is reported to have said, things should be made as simple as possible, but not simpler.

- *Enriching existing software.* Our initial FSM implementation was a simple execution engine, with no formal verification capabilities. When adding such capabilities, we aimed at minimizing the changes involved in the existing software. The result was that while some changes were necessary (e.g., the explicit declaration of formal variables; see Section 3.1.1), little extra information was actually needed. Most of the changes involved were, in essence, changes of syntax, to allow the necessary information for the construction of the TA model to be extracted from the program.

- *Gradual specification.* The more information is available (e.g., formal invariants), the better, but the technique allows the gradual addition of such information. This proved useful in the case study because there was too much software to be enriched at once. In general, this capability allows developers to choose the appropriate level of formal modeling needed at any given time. This possibility contrasts with traditional formal verification approaches, which typically require complete specifications before an actual working software can be produced.

- *Validation through formal modeling.* Although the TA model is meant mainly for verification of temporal properties, it turns out that it is easy to simulate and manipulate it in the UPPAAL user interface in a way that increases our understanding of the system. A specific fault was detected in this manner (Section 5.1).

With respect to scalability, the technique presents the same power and limitations of the underlying model checker, because the translation procedure itself has linear complexity with respect to the program size. The real burden, if any, is left for the formal verification algorithm. In the case of UPPAAL, it has been successfully applied to many industrial systems [5], which suggests that it scales sufficiently well to be useful in industrial practice. In the application reported here, for instance, verification took negligible time.

An alternative approach that we have not yet tried would be to implement a verification method specially designed to handle our specific FSM implementation. In particular, a Model-Based Testing approach [22] could be used in a relatively easy manner. Small machines could be exhaustively tested, larger ones could have a certain number of random paths selected and tested [8]. All this could be accomplished within the library itself, with no need of external tools, and could work as part of the usual regression testing commonly

used in industry today. However, such an approach would bring one complication: how to specify the properties of interest. When translating the problem to an external tool such as UPPAAL, one gains for free a language in which to query temporal properties about the model (in UPPAAL's case, a version of TCTL). If, however, the FSM library is to perform verification alone, it must also provide some means of expressing temporal properties in addition to the mechanism to explore its various execution paths.

Given all this, we can also now point out that it would not be simple to take an existing FSM library and somehow add the verification capabilities we described here. First because, as we have shown through the article, some fundamental software design decisions are crucial to permit later verification (see Section 3.2); for instance, it is easy and sensible for a mere execution engine to allow any kind of programmable precondition and invariant, but this impacts negatively the verifiability of the resulting system. Second, verification techniques and tools are very idiosyncratic as to their semantics and the elements they support, and therefore it would be hard to match the semantics implied by an existing tool, possibly quite powerful, with that of a model checker, possibly quite limited.

The technology and related methodology presented here fill a niche previously ill-served. We have shown that such an approach is useful, technically feasible, cost-effective and practical, thus a promising direction to follow. It would be interesting to further validate this approach through independent applications, specially in areas which are felt to require additional safeguards against errors, such as financial systems. We expect and encourage others to either build upon our tool or to develop similar ones.

7. REFERENCES

[1] J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 2005.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183 – 235, 1994.

[3] C. Baier, J.-P. Katoen, et al. *Principles of model checking*. MIT press Cambridge, 2008.

[4] G. Behrmann, A. David, and K. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg, 2004.

[5] G. Behrmann, K. G. Larsen, O. Möller, A. David, P. Pettersson, and W. Yi. Uppaal – present and future. In *Proceedings of the 40th IEEE Conference on Decision and Control*, volume 3, pages 2881–2886. IEEE, 2001.

[6] F. Chen and G. Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'07)*, pages 569–588. ACM press, 2007.

[7] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In E. Brinksma and K. Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer Berlin Heidelberg, 2002.

[8] A. Denise, M.-C. Gaudel, S.-D. Gouraud, R. Lassaigne, J. Oudinet, and S. Peyronnet. Coverage-biased random exploration of large models and application to testing. Technical Report 1494, LRI, Université Paris-Sud XI, June 2008.

[9] A. H. Dönni. The boost statechart library. `http://www.boost.org/doc/libs/1_60_0/libs/statechart/doc/index.html`. Last accessed: February 8th, 2016.

[10] T. A. S. Foundation. Commons SCXML. `http://commons.apache.org/proper/commons-scxml/`. Last accessed: February 8th, 2016.

[11] M. Geilen. On the construction of monitors for temporal logic properties. *Electronic Notes in Theoretical Computer Science*, 55(2), 2001.

[12] D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[13] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 441–444. Springer Berlin Heidelberg, 2006.

[14] O. M. G. Inc. Unified modeling language. `http://www.uml.org/`. Last accessed: February 8th, 2016.

[15] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a run-time assurance tool for java programs. In *Runtime Verification 2001 proceedings*, volume 55 of *ENTCS*. Elsevier Science Publishers, 2001.

[16] G. T. Leavens, A. L. Baker, and C. Ruby. JML: a java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, 1998.

[17] Y. Matsumoto and K. Ishituka. *Ruby programming language*. Addison Wesley Publishing Company, 2002.

[18] D. L. Parnas. On the use of transition diagrams in the design of a user interface for an interactive computer system. In *Proceedings of the 1969 24th National Conference*, ACM '69, pages 379–385. ACM, 1969.

[19] A. Pfeifer. state_machine. `https://github.com/pluginaweek/state_machine`. Last accessed: February 8th, 2016.

[20] C. W. Rapp. SMC: The state machine compiler. `http://smc.sourceforge.net/`. Last accessed: February 8th, 2016.

[21] V. Santiago, N. L. Vijaykumar, D. Guimarães, A. S. Amaral, and É. Ferreira. An environment for automated test case generation from statechart-based and finite state machine-based behavioral models. In *IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08)*, pages 63–72. IEEE, 2008.

[22] J. Tretmans. Model based testing with labelled transition systems. *Formal methods and testing*, pages 1–38, 2008.

[23] J. Woodcock and J. Davies. Using Z: Specification, refinement, and proof. 1996.