Cloud Certification Process Validation Using Formal Methods

Maria Krotsiani(✉), Christos Kloukinas, and George Spanoudakis

City, University of London, London, UK
{Maria.Krotsiani,C.Kloukinas,G.E.Spanoudakis}@city.ac.uk

Abstract. The importance of cloud-based systems is increasing constantly as they become crucial for completing tasks in an effective and affordable manner. Yet, their use is affected by concerns about the security of the data and applications provisioned through them. Security certification provides a means of increasing confidence in such systems, by establishing that they fulfil certain security properties of interest. Certification processes involve security property assessments against specific threat models. These processes may be based on self-assessment, testing, inspection or runtime monitoring of security properties, and/or combinations of such methods (hybrid certification). One important question for all such processes is whether they actually deliver what they promise. This question is open at the moment and is the focus of our work. To address it, we have developed an approach that formalises certification processes, by translating them in the language of the Prism model-checker and uses Prism to verify properties of interest on the model of the certification process, under specific environmental assumptions.

Keywords: Cloud certification · Validation · Probabilistic model checking

1 Introduction

Certification of cloud systems security is important for increasing confidence in cloud service provision. Security certification has traditionally been based on standards and certification schemes (*e.g.*, ISO27001 [22], ISO27002 [22], Common Criteria [9]), which define the security controls that a system should implement to be secured under specific threat models. Certification processes tend to be lengthy and costly, reducing their use [14]. A number of certification schemes focusing on cloud systems and services has also emerged. Some of these schemes are based on self-assessment [12,15,16]. Other certification processes use testing [13] or a combination of formal analysis and testing [8]. Most current certification schemes do not involve a continuous assessment of security, leading to proposals incorporating continuous monitoring of cloud systems and services security, as for example in the CUMULUS project [19,20]. In CUMULUS, security properties are expressed in Event Calculus [27] and are continuously monitored using the EVEREST [28] monitoring platform.

M. Maximilien et al. (Eds.): ICSOC 2017, LNCS 10601, pp. 65–79, 2017.
https://doi.org/10.1007/978-3-319-69035-3_5

In this paper, we present an approach that enables the analysis and validation of cloud certification processes themselves – a necessity for both certifiers and those seeking certification, so that they can better understand what they are committing to when agreeing to a particular certification process. It is based on formalising the process, in order to enable the formal analysis of its consequences under different environment assumptions, *i.e.*, different probabilities for the occurrence of specific environment events (*e.g.*, monitoring results and/or outcomes of the testing process).

Our approach allows a Certification Authority (CA), *i.e.*, the stakeholder, who establishes and oversees the operation of a certification scheme, to define a Certification Model (CM) as an input for the certification process. These CMs are specified in an XML-based language and then translated into probabilistic timed automata, in the language of the probabilistic model-checker PRISM [4,21]. Thus, one can verify different properties, *e.g.*, find the probability of issuing a certificate after monitoring a cloud system for a given period of time, or the probability of revoking a certificate within a given period time after it was issued.

In the following, Sect. 2 presents the overall framework and Sect. 3 gives a running example of a CM. Section 4 presents the certification process and how this is mapped at a high-level into a PRISM model. In Sect. 5 we present the translation from our CM language to PRISM. Section 6 presents the outcomes of experiments that we have conducted, while Sect. 7 reviews related work and, finally, Sect. 8 provides concluding remarks and future directions.

2 Framework Overview

A certification process starts when a *Certification Authority (CA)* submits a *Certification Model (CM)* to the CUMULUS Certification Platform. The Certification Platform in Fig. 1 has three main components. The *CM2Monitor Translator* component translates the CM into an executable format for the platform. It also extracts the operational monitoring specification (*i.e.*, monitoring rules) for the *Monitor* to check against the systems events. The *Certification Manager* component manages the overall certification process. It receives the translated CM and it communicates with the monitor (sending rules and receiving monitoring results). Finally, the *Anomaly Manager* component detects *anomalies* by using the monitoring results. Monitoring rules in CUMULUS are divided into anomaly and assertion rules, *i.e.*, soft constraints requiring further inspection (anomalies) and hard constraints that must be met always (assertions).

The Deployment Infrastructure builds on a general-purpose monitoring architecture [17], and it consists of *event sensors*, a *monitor*, and a *CM2Monitor Translator*. This infrastructure automates the certification process at run-time, but since CMs and system behaviours can be complex, CMs themselves must be validated. For this purpose, we have extended the framework with a tool-chain for formal validation of CMs. We have also re-implemented the Certification Manager execution engine to follow the formal semantics. Validation is based on translating CMs to the Prism model-checker's language [4,21]. In order to explore

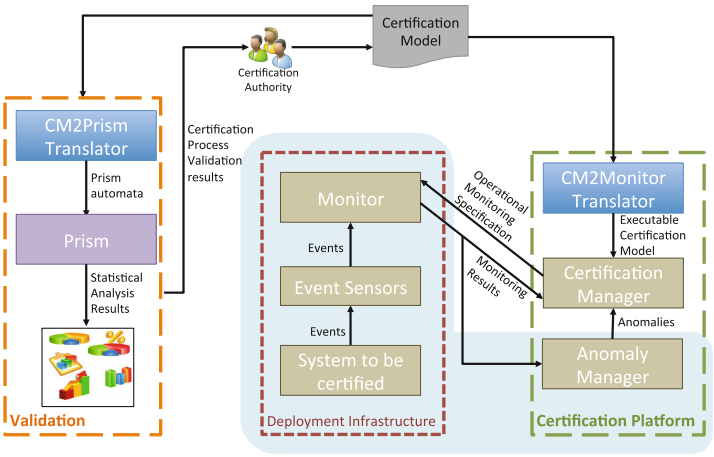


Fig. 1. Overall Architecture (Color figure online)

the CM consequences for specific systems, it also produces a formal model of its environment. This environment is inside the light blue shaded area in Fig. 1, comprising the Anomaly Manager, the system itself, and the system's monitor. Since the environment is too complex to be described formally in details, it is abstracted away and represented as a source of stochastic monitoring results. By using Prism, involved parties can check for properties such as:

- Is the CM respecting the high-level certificate life cycle?
- What is the probability of having to revoke an issued certificate?
- How is this probability affected by other parameters through time?

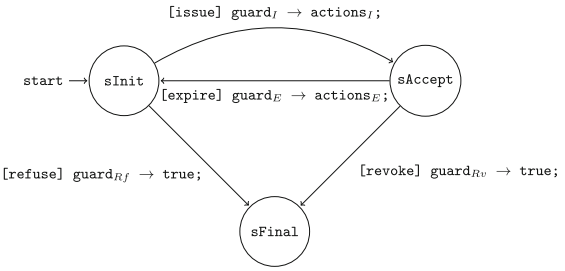
The outcomes of this validation are passed back to the responsible CA to adapt the CM accordingly, *e.g.*, adjust terms to reduce the possibility of a revocation.

3 Running Example

Figure 2 presents an example to demonstrate the different aspects of our approach. In this example, a certificate starts in the `sInit` state. The transition `issue` to the state `sAccept` occurs when $\texttt{guard}_I$ is satisfied. This guard states that: *(i)* violated assertion rules should not be excessive (`ViolatedAssertions() < TooManyVio`); *(ii)* unresolved anomalies should be below their threshold; *(iii)* accumulated evidence should be at least `EnoughEvents`; and *(iv)* the cloud service should be monitored for at least the defined `monitoringTime`.

The satisfaction of the $\texttt{guard}_{Rf}$ in the state `sInit` fires the `refuse` transition that leads to the final state `sFinal`. The satisfaction of the $\texttt{guard}_{Rv}$ in the `sAccept` state fires the `revoke` transition, which also leads to the `sFinal` state. Finally, the satisfaction of the $\texttt{guard}_E$ in the `sAccept` state fires the `expire` transition that leads back to the `sInit` state, where the whole process starts again.

```
1  intp TooManyVio 1                              intp TooManyUnresolved 1
2  durationp monitoringTime secs(20)              durationp expiryTime secs(10)
3  intp EnoughEvents 2                            clock localClock
4  int usedevents min 0 max MaxInteger init 0     enum state sInit sAccept
       sFinal init sInit
```



where:

guard$_I$: (ViolatedAssertions() < TooManyVio) & (UnresolvedAnomalies() < TooManyUnresolved) & ((SeenEvents() - usedevents) ≥ EnoughEvents)) & (localClock ≥ monitoringTime)
actions$_I$: (localClock'=0) & (usedevents'=SeenEvents())
guard$_E$: (ViolatedAssertions() < TooManyVio) & (UnresolvedAnomalies() < TooManyUnresolved) & (localClock ≥ expiryTime)
actions$_E$: (localClock'=0)
guard$_{Rf}$: ((ViolatedAssertions() ≥ TooManyVio) | (UnresolvedAnomalies() ≥ TooManyUnresolved))
guard$_{Rv}$: ((ViolatedAssertions() ≥ TooManyVio) | (UnresolvedAnomalies() ≥ TooManyUnresolved))

Fig. 2. Example Certification

4 Certification Process and Prism Model

4.1 Certification Model

Schema. The certification model is specified in XML, with its BNF equivalent as produced by the $\mathbb{K}$ Framework [26] shown in Fig. 3. The top element of the model of interest to this paper is the `LifeCycle`. `LifeCycle` declares the unit of time that is assumed for the certification process and a list of typed parameters (*i.e.*, constants) of interest to the specifier "*Ps*", such as the `TooManyVio` in Fig. 2. Parameters can be of four different types: *boolp, intp, floatp*, and *durationp* (*i.e.*, integers associated with a time unit). `LifeCycle` also declares typed variables "*Vs*". These can be of one of the following five types: *bool, int, duration, enumeration*, and *clock*. Variables cannot be of type float as Prism does not support such a type [4]. Parameters and variables have a name and an initial value. In addition, *int* and *duration* variables have min and max values, *e.g.*,`usedevents` in Fig. 2. Finally, `LifeCycle` declares a list "*Ts*" of model transitions Each transition has a type, a guard, and a sequence of variable assignments. Type can be a user-defined one (`other`) or a fixed one (`issue`, `expire`, `refuse`, and `revoke`). The guard is a predicate (*Pred* in Fig. 3) over variables and parameters and the predefined Certification Manager functions (`SeenEvents`, *etc.*– *cf.* next section).

A certification model also contains other XML elements of use to the framework that we omit here as they are of no relevance to this paper (*e.g.*, the Target of Certification defining the service to be certified, *etc.*).

```
                                    MODULE CERT-LIFECYCLE-SYNTAX
SYNTAX   CLifeCycle ::= LifeCycle { TimeUnit  Ps  Vs  Ts}
SYNTAX   TimeUnit ::= TimeUnit NExp
SYNTAX   Param ::= boolp Id  Pred  | intp Id  NExp
                 | durationp Id  NExp  | floatp Id  NExp
SYNTAX   Var ::= bool Id  Pred  | int Id min NExp max NExp init NExp
               | duration Id min NExp max NExp init NExp
               | clock Id  | enum Id  IdList init Id
SYNTAX   Transition ::= issue Pred  Actions  | expire Pred  Actions
                      | refuse Pred  Actions  | revoke Pred  Actions
                      | other Id  Pred  Actions
SYNTAX   Action ::= Id = Pred  | Id = NExp  | Id reset
SYNTAX   Function ::= secs  | ··· | weeks | SeenEvents | DetectedAnomalies
                    | ViolatedAssertions | SatisfiedAssertions
                    | ResolvedAnomalies | UnresolvedAnomalies
                                    END MODULE
```

Fig. 3. BNF representation of the Certification Model XML Schema (fragment)

Semantics. The overall model comprises the certification process manager that receives events from its environment, which is represented by five parameters:

`AnomalyP` : Probability of an event to be an anomaly and not an assertion rule;
`ViolationP` : The (conditional) probability of an assertion rule to be violated;
`UnresolvedP` : The (conditional) probability of an anomaly to not be resolved;
`minRuleTime` : Minimum time it takes for a new rule event; and
`maxRuleTime` : Maximum time it takes for a new rule event.

Using these parameters, the environment produces anomaly and assertion events with a temporal distance in [`minRuleTime`, `maxRuleTime`]. Moreover, it updates a set of model variables representing the Certification Manager counter functions:

`SeenEvents` : Events produced since the beginning of time, $t = 0$;
`DetectedAnomalies` : Anomaly events produced since $t = 0$;
`ResolvedAnomalies` : Anomaly events resolved since $t = 0$;
`UnresolvedAnomalies` : Anomaly events not resolved since $t = 0$;
`SatisfiedAssertions` : Satisfied assertion events since $t = 0$;
`ViolatedAssertions` : Violated assertion events since $t = 0$;

Formal Modelling Framework. As we require probabilities and time to represent the environment and express the properties of interest for validating a certification process, our model uses Probabilistic Timed-Automata (PTA) [25], as supported by the Prism model checker [21]. A PTA is a tuple $P = (Locs, l_0, Clocks, Act, Inv, EnabConds, ProbTrans, Lab)$, where [25]:

- *Locs* is a finite set of locations, and $l_0 \in Locs$;
- *Clocks* is a finite set of clocks, and *Act* a finite set of action names;
- *Inv* is an invariant on *Locs* and clock constraints – $Inv : Locs \rightarrow CC(Clocks)$;

Listing 1.1. Environment in Prism

```
1  formula SeenEvents =
2      min(MaxInteger,  SatisfiedAssertions + ViolatedAssertions + ResolvedAnomalies +
           UnresolvedAnomalies);
3  formula DetectedAnomalies = min(MaxInteger,  ResolvedAnomalies + UnresolvedAnomalies);
4   timeToNextRuleResult : clock; // Clock used for the environment events.
5   // Functions.
6   ViolatedAssertions : [0 .. MaxInteger] init 0;  UnresolvedAnomalies: [0 .. MaxInteger
        ] init 0;
7   SatisfiedAssertions: [0 .. MaxInteger] init 0;  ResolvedAnomalies : [0 .. MaxInteger]
       init 0;
8   invariant (timeToNextRuleResult<=maxRuleTime) endinvariant
9    // The stochastic behaviour of the environment:
10  [event] (timeToNextRuleResult>=minRuleTime)
11     ->     AnomalyP *(1-UnresolvedP): (timeToNextRuleResult'=0)
12        & (ResolvedAnomalies'   = min(MaxInteger,ResolvedAnomalies+1))
13      +    AnomalyP *   UnresolvedP : (timeToNextRuleResult'=0)
14        & (UnresolvedAnomalies' = min(MaxInteger,UnresolvedAnomalies+1))
15      + (1-AnomalyP)*(1-ViolationP) : (timeToNextRuleResult'=0)
16        & (SatisfiedAssertions' = min(MaxInteger,SatisfiedAssertions+1))
17      + (1-AnomalyP)* (ViolationP)  : (timeToNextRuleResult'=0)
18        & (ViolatedAssertions'  = min(MaxInteger,ViolatedAssertions+1));
```

- *EnabConds* are clock conditions – $EnabConds : Locs \times Act \rightarrow CC(Clocks)$;
- *ProbTrans* is a partial probabilistic transition function, which given a location and an action name, gives a probability distribution over the next states (defined by a subset of clocks that are reset to zero by the transition named with action that leads to a new location, and that location) – $ProbTrans : Locs \times Act \rightarrow Dist(2^{Clocks} \times Locs)$; and
- *Lab* labels each location with a set of atomic propositions – $Lab : Locs \rightarrow 2^{AP}$

Clock constraints over a set of *Clocks*, $CC(Clocks)$, are defined by the syntax $\chi ::= \text{true} | x \leq d | c \leq x | x + c \leq y + d | \neg\chi | \chi \wedge \chi$, where $x, y \in Clocks$ and $c, d \in \mathbb{N}$ [25]. A PTA is *well-formed* when all enabled transitions take the automaton to states satisfying the clock invariant – see [25].

Environment Model. Listing 1.1 shows the environment part of the Prism model. It uses the clock variable `timeToNextRuleResult` (in line 4) to produce events between minRuleTime and maxRuleTime time units. The clock invariant (line 8) imposes the upper bound, while the clock guard (line 10) imposes the lower bound. Each time an event is produced we have a probabilistic choice between four possible alternative new states – two relating to anomalies (so conditioned on AnomalyP) and two to assertions (conditioned on (1−AnomalyP)). In each case the event is marked as negative (with probability UnresolvedP or ViolationP) or positive (with their complements), and we update the respective counter function.

Environment transitions are followed by transitions encoding the CM process. Process transitions in the Prism model correspond one-to-one to the actions in the process definition. Thus, for each action in the definition, there is a new transition with the same guard as the action and the same assignments. The transition name is the same as the name of the action. An action can be one of the standard ones: `issue`, `refuse`, `expire`, and `revoke`. For non-standard actions, the transition name is prefixed with "`u_`", to highlight it as non-standard.

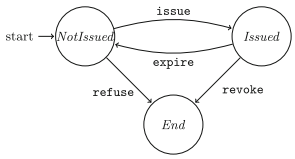


Fig. 4. High-level Certificate Life Cycle

Similarly, all user variables have the same prefix to avoid clashes with the constants, variables, and formulæ that we use for book-keeping, *e.g.*, `MaxInteger`, `SatisfiedAssertions`, `SeenEvents`. In this way, non-standard actions, *e.g.*, "`notify`", and user-specified variable assignments cannot alter the model semantics.

Listing 1.2. Lifecycle Observer Module

```
1  module HighLevelLifecycle
2    error : bool init false;
3    active: bool init true;
4    issued: bool init false;
5   [issue]    active & !issued & !error
6            -> (issued'=true);
7   [issue]  !(active & !issued & !error)
8            -> (error'=true);
9   [refuse]   active & !issued & !error
10           -> (issued'=false) & (active'=false);
11  [refuse] !(active & !issued & !error)
12           -> (error'=true);
13  [expire]   active & issued & !error
14           -> (issued'=false);
15  [expire] !(active & issued & !error)
16           -> (error'=true);
17  [revoke]   active & issued & !error
18           -> (issued'=false) & (active'=false);
19  [revoke] !(active & issued & !error)
20           -> (error'=true);
21 endmodule//
```

Semantics of the High-Level Certificate Life Cycle. The semantics also include the definition of the high-level certificate life cycle as shown in Fig. 4. A certificate starts at the state *NotIssued*, where it can be either refused or issued. If it is issued (*Issued*), then it can be either expired or revoked. Actions refuse and revoke end the certificate life cycle, while action expire changes the abstract certificate state back to the *NotIssued* state. The high-level certificate life cycle does not consider any user-defined actions. The `HighLevelLifecycle` module in Listing 1.2 observes whether one of the standard actions is taken in a state where it is not applicable. In this case it sets the variable `error` to true and refuses to take any more standard actions (all guarded by `!error`). The Prism property "`Pmax=? [F (error)]|`" verifies that this life cycle is respected, by asking for the maximum probability of eventually (F) reaching a state where `error` is true – this should be zero.

5 Code and Prism Model Generator

As shown in Fig. 1, there are two components that translate the Certification Model (CM) – the CM2Prism Translator and the CM2Monitor Translator.

Listing 1.3. Pseudo-code for Combining Types (fragment)

```
conv combineAdd(type tpA, type tpB) {
  bool swapd=false; type tp1=tpA, tp2=tpB;
  if (tpA > tpB) {tp1=tpB; tp2=tpA; swapd=true;}
  switch (tp1) {
        case INT:
            switch (tp2) {
            case INT:   return INT;
            case FLOAT: return FLOAT;
            }
        case SECONDS:
            switch (tp2) {
            case SECONDS: return SECONDS;
            case MINUTES:
            return conv(SECONDS,
             swapd ? 60 :  1, swapd ?  1 : 60);
            default: // ask MINUTES
            conv r=combineAdd(MINUTES, tp2);
            return conv(SECONDS,
             (swapd? 60*r.scale1 :r.scale0),
             (swapd? r.scale0 :60*r.scale1));
            }
        // ... other types
  }
  return NONE;
}
```

The former component is responsible for producing a formal Prism model for analysing the CM and deciding whether it is fit for purpose. The latter component produces a set of monitoring rules that are passed to the runtime monitor (work described in [17]) and at the same time produces an executable version of the CM for the Certification Manager. As the translation happens at runtime, we translate to Lisp, as it can execute code produced dynamically.

The translations to the Prism model and Lisp code are done by the same piece of code – a decision taken to make it easier to track both artefacts and increase our confidence that they follow the same logic. The translator traverses the XML structure of the CM using a reflective Java visitor and applies a method `visitX` to each element `X`. Each method `visitX` updates certain global information (*e.g.*, names and types of variables), calls the appropriate visitors for the sub-elements of the element `X` and produces one string for the Prism model and another one for the Lisp code. We keep a (hash) Map of IDs to type information (a name and a type pair). Thus, a variable definition "`bool foo false`" will insert into the symbol map the mapping "`foo`" → ("`u_foo`", BOOL). The abstract syntax tree node for each XML element X contains the type of X, its representation in Prism, and its representation in Lisp. So for a declaration like "`int bar min (3 * 6) max (100 - 7) init (40 + 2)`", which declares an integer variable with min/max values and an initial value, we create a node of type INT with the following two strings: (i) "`u_bar : [(3 + 6)..(100 - 7)] init (40 + 2);`" for Prism, and (ii) "`(defparameter u_bar (+ 40 2) )`" for the Lisp interpreter (which does not need any type information, nor min/max values for the variable).

Type Conversions. The translation is mostly straightforward – what makes it more interesting is the type-checking and type promotion that is performed, *e.g.*, when we add a float to an integer the result is a float, in particular for

Listing 1.4. Life-cycle Execution Loop Body

```
(defun lifecycle-loop ()
 (progn
  (update-time) ;; used by guard and actions
  (let ((tr (find-if (lambda (x)
              (funcall (transition-guard x)))
              ***transitions***)))
   (when tr
    (funcall (transition-action tr))))))
```

duration expressions. Expressions involving durations are being transformed into the lowest unit used, *e.g.*,adding seconds to minutes results in seconds. Listing 1.3 shows the pseudo-code for type conversion when we have an *additive* expression. We see that INT+INT produces an INT, INT+FLOAT a FLOAT, and SECONDS+MINUTES produces SECONDS with scaling factors of 1 for the first expression and 60 for the second one. When the first type is SECONDS and the other is not MINUTES, we call the same function recursively pretending that the first type was MINUTES, so as to see if we can convert to MINUTES first and then to SECONDS. The recursion terminates at type WEEKS, the last duration type, that knows only how to add itself to another type WEEKS – it is always the smaller type that knows how to convert the type that is one level up. Similar functions exist for multiplication and division, as one can multiply two INTs to get an INT, an INT and a DURATION to get a DURATION but cannot multiply two DURATIONs, and can divide two DURATIONs but not an INT and a DURATION. Type translation is needed for both the Prism formal model and the Lisp code we generate, as otherwise we would not be able to have duration expressions where units are mixed. For the Lisp code all durations are eventually converted to nano-seconds, as that is the smallest unit supported by its system clock.

Lisp Interpreter. The Lisp code that is called continuously at run-time is shown in Listing 1.4. It first updates the time of the clocks by storing the current time in global variable `***now***`. It then selects the first transition (based on the order defined in the CM), whose guard is true. If there is such a transition, it executes its actions. Listing 1.5 shows the code corresponding to the issue transition of the example in Fig. 2. We use ABCL, a Java-based Common Lisp, to execute the CM, as this allows smooth interfacing with the rest of the framework.

Language Constraints. As aforementioned in Sect. 4, our XML schema permits FLOAT parameters but not FLOAT variables, as the Prism modelling language does not support the latter – see the on-line Prism Manual [4]. Another inherited constraint has to do with the treatment of clock variables. While PTAs allow comparisons between clocks and both strict (*e.g.*, $<$) and non-strict (*e.g.*, $\leq$) comparisons, currently Prism only supports non-strict ones in all the analysis engines it has. For this reason we decided to also include this constraint, which may make it somewhat harder to express some guards, as now one needs to be careful to not introduce strict comparisons, *e.g.*, through negation.

Listing 1.5. Transition Definition in Lisp

```
1  (make-transition :name "issue"
2   :guard (lambda ()
3   (and (= u_state u_sInit)
4        (< (***ViolatedAssertions***) u_TooManyVio)
5        (< (***UnresolvedAnomalies***)
6           u_TooManyUnresolved)
7        (>= (- (***SeenEvents***) u_usedevents)
8            u_EnoughEvents)
9        (>= (- ***now*** u_localClock)
10           (* 1000000000 u_monitoringTime))))
11  :action (lambda ()
12  (progn
13   (assert (= *lstate* *slPreIssued*) ()
14     "*lstate* is ~S" *lstate*)
15   (setq *lstate* *slIssued*)
16   (setq u_state u_sAccept)
17   (setq u_localClock ***now***)
18   (setq u_usedevents (***SeenEvents***)))))
```

5.1 Differences Between Prism Model and Code

Variable types & limits. We have already seen that variable names in Lisp do not have types, as they do in the Prism model, and integers do not have min/max values either, as they are actually bignums, *i.e.*, arbitrary length integers. But these are not the only differences between the two artefacts we produce.

Clock resets. By comparing the issue transition in Fig. 2 and in Listing 1.5, we can see that instead of resetting the clock variable `localClock` to zero, as it is done in the Prism model, we assign to it the current time of the global clock `***now***`. A similar change is also in the guard – instead of comparing the clock against the duration `monitoringTime` directly, we compare its distance from `***now***` (after converting the duration to nano-seconds).

Time granularity. In the implementation, all clocks and durations are expressed in nano-seconds. In the Prism model, clocks do not have a unit, so all durations are transformed to the same unit (that needs to be provided in the CM as `TimeUnit`). Dividing a duration expression by `TimeUnit` should produce a natural number, since clocks in PTAs can be compared against natural numbers only.

Tracking of the high-level life cycle state. The model and the implementation track the high-level life cycle state differently. In the Prism model we use an additional module called `HighLevelLifecycle` (see Listing 1.2), which synchronises with the main model module and checks if there are erroneous transitions. Instead, in the Lisp code each transition assigns an internal variable `*llstate*` to keep track of the current high-level life cycle state of the certificate – see line 15 in Listing 1.5. It uses this variable to assert the correct state of the certificate, before performing any of the transition assignments – see line 13 in Listing 1.5.

Continuous vs discrete execution points. In the Prism model, a transition like the issue one (Listing 1.2) can fire at any time point that satisfies its guard. In the Lisp code, the respective transition (Listing 1.5) will only be considered

every d seconds, where d depends on implementation issues, *e.g.*, the delay we have introduced in the main evaluation loop to avoid constant re-evaluation of transitions.

Non-deterministic vs deterministic behaviour and eager execution. The most important difference is that the Prism model has a non-deterministic behaviour – whenever multiple transitions are enabled it can execute any of them. It can actually elect to not execute any transition at all and instead simply let the time pass – Prism does not support *urgent* transitions [25] that must be taken immediately when they are enabled without allowing time to advance. The code we produce on the other hand, will always choose the first enabled transition and will execute it in an eager manner, without allowing time to pass.

Due to the last difference (and the one before it), the Lisp code that we produce *simulates* the behaviour of the Prism model, *i.e.*, exhibits only one possible behaviour among the behaviours that it can have. This is the usual case with all implementations of some formal model – the model is by definition more general, both because it has abstracted a number of implementation details away (*e.g.*, the execution speed of the system), and because it needs to describe a *family* of implementations and not a single one. For example, another reasonable implementation may choose to execute the last enabled transition. Yet another may choose a transition "non-deterministically", by evaluating the guards of the transitions in parallel and choosing the one whose guard evaluates to true first. This is something that will depend heavily not only on the expression each guard has to evaluate, but also on the current system state when evaluating these expressions, such as the current memory usage, the CPU load, *etc.*

6 Experimental Results

We have performed a number of experiments with the CM example of Sect. 3.

Experiment 1 – Respecting the high-level certificate life-cycle. Prism establishes that there is no error in the defined life-cycle of the CM, by calculating that the maximum probability of "`Pmax=? [F (error)]`" is zero (in 197.063 s). In a previous version of the CM, the result of this probability was non-zero, as we had mistakenly guarded the refuse action, with the certificate being at the `sAccept` instead of the `sInit` state.

Experiment 2 – Establish the maximum probability of revoking an issued certificate. Given the "`Pmax=? [F (revokeGuard)]`", Prism reports that `Pmax` is 0.262144 (in 209.7 s), which is too high – revoking a certificate is undesirable, since we have certified something as trustworthy, when in fact it is not.

Experiment 3 – Explore the system behaviour. We need to explore the system behaviour to understand why revocations can occur with such a high probability. One can analyse the probability of having an assertion or an anomaly

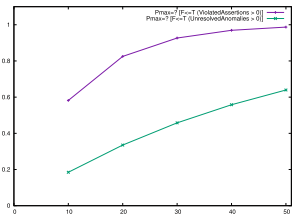


Fig. 5. Violations of assertions vs anomalies

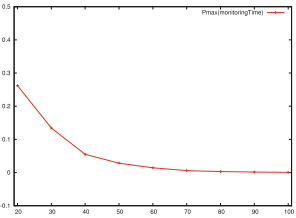


Fig. 6. `[F (revokeGuard)]` vs `monitoringTime`

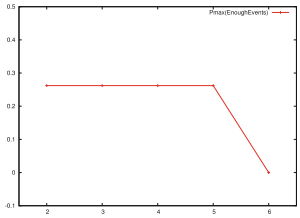


Fig. 7. `[F (revokeGuard)]` vs `EnoughEvents`

rule violation within T time units, as in Fig. 5. Anomaly rule violations start with a probability of 0.19 at time point 10 and reach a probability of 0.64 at time point 50. Assertion violations are more probable – they start with 0.58 and reach 0.99, so it is almost certain to have observed an assertion violation by time point 50.

Experiment 4 – Identify parameters that should be modified. We need to identify parameters that are too lax and discover better values for them to exclude this undesirable behaviour. A primary target is `monitoringTime` – maybe increasing it will render revocations improbable. Figure 6 shows the results (maximum probability for revocation) when `monitoringTime` ranges in $[20, 100]$ with a step of 10 (each point calculated in between 241.859 s and 694.947 s). The maximum probability drops constantly as the minimum monitoring time is increased. For a duration of 90 it drops to 0.00154 and for 100 to practically zero ($6.33 * 10^{-4}$).

Another interesting parameter is `EnoughEvents` – the minimum number of monitoring results we wish to observe before issuing a certificate. Exploring the behaviour of the system for values of this variable in the range $[2, 6]$ with a step of 1, produces the results in Fig. 7 (calculated in between 6.928 s and 237.929 s). The maximum probability for revoking the certificate stays constant at 0.262144 until we ask to observe at least 6 monitoring results, in which case it drops to exactly zero. So the parameter `EnoughEvents` offers better control. It also leads to models that can be analysed much faster than those that depend on the `monitoringTime` – this is because temporal constraints are far more expensive to analyse in PTAs than constraints involving discrete variables.

Experiment 5 – Re-validating chosen parameter values. The maximum probability of revoking a certificate when the probability of violation ranges in $[0.01, 0.35]$ (with a step of 0.02) validates setting `EnoughEvents` to 6 as a good choice. All cases report a zero probability, in between 6.372 and 8.201 s for each case.

7 Related Work

There is substantial work in validating and verifying cloud service providers or cloud services. Extensive work concerns the way evidence is collected to verify

security properties of cloud services. Evidence collection can be based on *(i)* assessments regarding specific standards or regulations, performed by either the cloud providers or third party authorities, known as self-assessment; *(ii)* trusted platform modules (TPM); *(iii)* performing tests; or *(iv)* continuous monitoring.

In self-assessment one either completes a specific questionnaire, as in the case of CSA STAR Level 1 and Level 2 [12], or completes reports regarding specific national or international standards, such as the CIF Guidance [10], COBIT [2], the compliance framework FISMA [16], or TRUSTe [6].

Trusted computing targets the integrity of software, processes, or data by collecting evidence through TPMs and related hardware. Muñoz and Maña [24] combine software and hardware-based cloud certification, aiming to bridge the gap between cloud certification and trusted computing. Another approach is MyCloud by Li *et al.* [23]. MyCloud is an architecture used for privacy protection based on traditional encryption mechanisms. It aims to allow clients to configure their own privacy protection, by decreasing as much as possible the trusted computing base and the cloud providers' ability to modify privacy settings.

With test-based evidence collection, research has mostly focused on the problem of testing web services. Damiani *et al.* [13] use security certificates based on signed test cases for assessing and certifying web services. A first step in the area of web service certification was done by SEI in 2008, which defined a web service certification and accreditation process for the US Army CIO/G-6 [5]. Anisetti *et al.* [7] provided a test-based security certification solution for services and a first approach to its integration within the SOA environment. The ASSERT-4SOA EU project also focused on formal and test-based service certification [8].

Finally, monitoring and dynamic collection of evidence for cloud services is a more recent development, due to its additional complexity. It requires uninterrupted monitoring services, even though the monitoring capabilities available in a service-based system change due to the dynamic nature of cloud services. To address these needs the SLA@SOI EU project has developed a dynamically configurable monitoring infrastructure for dynamically checking SLA monitorability, which runs on cloud systems and adapts automatically to changes in the available monitoring capabilities in service based systems [17,18]. Monitoring has also been used at the hypervisor layer to provide incident detection even when the guest OS experiences critical conditions and monitoring agents are unable to communicate with monitoring systems. Amazon's CloudWatch is a system of this category [1]. Moreover, Cloud Security Alliance's Cloud Trust Protocol [11] provides interfaces for extracting monitoring data from cloud systems.

Thus, most of the work in cloud certification focuses so far mainly on verifying and validating security properties of cloud providers and cloud services. To the best of our knowledge, our approach is the first one focusing on the exploration and validation of the certification process itself, prior to employing it.

8 Conclusion and Future Work

In this paper, we have presented an approach for analysing and validating cloud certification processes based on formal method techniques. This approach

translates a certification model (CM) into a model for the Prism model checker and into an executable version of it, in Lisp code.

For the Prism model, the environment of the certification process is modelled with probabilistic timed automata. The actions of the environment are abstracted by the probability of their occurrence. This probability is either estimated, obtained from historical data, or obtained by observing the system at run-time. This formal model enables analysing the process for different properties, from adherence to the expected high-level certificate life cycle, to min/max probabilities of revoking an issued certificate, *etc.* This allows one to explore whether the CM behaves as desired and can be used to certify cloud services.

At the same time, we translate the CM to code to be executed at run-time for certifying the cloud service in question. This code is produced alongside the Prism model and follows its behaviour, so that the analysis results remain valid.

In the future we plan to extend our framework so that it can also consider additional constraints on the expected behaviour of a cloud service and sufficiency conditions on this behaviour that must be met in order to issue a certificate. Currently our approach considers only the monitoring results for the properties we check on the cloud service, while the extended version would also observe the primitive execution events of the service. This capability would allow to observe a particular set of primitive event patterns for issuing a certificate.

Acknowledgments. This work was partly supported by the EU-funded project CyberSure [3] (grant no 734815).

References

1. Amazon CloudWatch, http://aws.amazon.com/cloudwatch/
2. COBIT, http://www.isaca.org
3. CyberSure (CYBER Security inSURancE), http://cybersure.eu/
4. Prism Model Checker, http://www.prismmodelchecker.org/
5. Securing Web services for army SOA, www.sei.cmu.edu/solutions/softwaredev/securing-web-services.cfm
6. TRUSTe, http://www.truste.com/
7. Anisetti, M., Ardagna, C.A., Damiani, E.: Defining and matching test-based certificates in open SOA. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 520–522. IEEE (2011)
8. Anisetti, M., Ardagna, C.A., Guida, F., Gürgens, S., Lotz, V., Maña, A., Pandolfo, C., Pazzaglia, J.-C., Pujol, G., Spanoudakis, G.: ASSERT4SOA: toward security certification of service-oriented applications. In: Meersman, R., Dillon, T., Herrero, P. (eds.) OTM 2010. LNCS, vol. 6428, pp. 38–40. Springer, Heidelberg (2010). doi:10.1007/978-3-642-16961-8_11
9. Ccdb, USB Working Group: Common Criteria (CC) for Information Technology Security Evaluation (2012), http://www.commoncriteriaportal.org
10. Cloud Industry Forum: CIF Guidance, www.cloudindustryforum.org/about-us
11. CSA: Cloud Trusted Protocol, https://cloudsecurityalliance.org/research/ctp/
12. CSA: CSA Security, Trust and Assurance Resigtry (STAR), https://cloudsecurityalliance.org/star/

13. Damiani, E., Ardagna, C.A., El Ioini, N.: Open Source Systems Security Certification. Springer, US (2008)
14. ENISA: Security Certification Practice in the EU: Information Security Management Systems - A Case Study (2013), https://www.enisa.europa.eu/
15. FedRAMP Office: Guide to Understanding FedRAMP (2013), www.gsa.gov/portal/mediaId/170599/fileName/Guide_to_Understanding_FedRAMP_042213
16. FISMA: Federal Information Security Management, https://www.dhs.gov/federal-information-security-management-act-fisma
17. Foster, H., Spanoudakis, G.: Advanced service monitoring configurations with SLA decomposition and selection. In: Proceedings of ACM Symposium on Applied Computing, pp. 1582–1589. ACM (2011)
18. Foster, H., Spanoudakis, G.: Smart: A workbench for reporting the monitorability of services from SLAs. In: Proceedings of 3rd International Workshop on Principles of Engineering Service-Oriented Systems, pp. 36–42. ACM (2011)
19. Katopodis, S., Spanoudakis, G., Mahbub, K.: Towards hybrid cloud service certification models. In: IEEE International Conference on Services Computing, SCC, pp. 394–399. IEEE Computer Society (2014), http://dx.doi.org/10.1109/SCC.2014.59
20. Krotsiani, M., Spanoudakis, G., Mahbub, K.: Incremental certification of cloud services. In: SECURWARE 2013–7th International Conference on Emerging Security Information, Systems and Technologies, pp. 72–80 (2013)
21. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22110-1_47
22. Lagazio, M., Barnard-Wills, D., Rodrigues, R., Wright, D.: Certification schemes for cloud computing. EU Commission Report, http://dx.doi.org/10.2759/64404
23. Li, M., Zang, W., Bai, K., Yu, M., Liu, P.: MyCloud: Supporting user-configured privacy protection in cloud computing. In: Proceedings of 29th Annual Computer Security Applications Conference, pp. 59–68. ACM (2013)
24. Muñoz, A., Maña, A.: Bridging the gap between software certification and trusted computing for securing cloud computing. In: Ninth World Congress on Services, pp. 103–110. IEEE (2013)
25. Norman, G., Parker, D., Sproston, J.: Model checking for probabilistic timed automata. Formal Methods Syst. Des. **43**(2), 164–190 (2013), http://dx.doi.org/10.1007/s10703-012-0177-x
26. Rosu, G., Serbanuta, T.: An overview of the K semantic framework. J. Log. Algebr. Program. **79**(6), 397–434 (2010), http://dx.doi.org/10.1016/j.jlap.2010.03.012
27. Shanahan, M.: The event calculus explained. In: Wooldridge, M.J., Veloso, M. (eds.) Artificial Intelligence Today. LNCS, vol. 1600, pp. 409–430. Springer, Heidelberg (1999). doi:10.1007/3-540-48317-9_17
28. Spanoudakis, G., Kloukinas, C., Mahbub, K.: The SERENITY runtime monitoring framework. In: Kokolakis, S., Gómez, A.M., Spanoudakis, G. (eds.) Security and Dependability for Ambient Intelligence. AIDS, vol. 45, pp. 213–237. Springer, Boston (2009), http://dx.doi.org/10.1007/978-0-387-88775-3_13