Timed Model-Based Conformance Testing
A Case Study Using TRON

Testing Key States of Automated Trust Anchor Updating (RFC 5011) in AUTOTRUST

Bachelor Thesis

Carsten Rütz
Radboud Universiteit Nijmegen
The Netherlands

Supervisors

Julien Schmaltz
Open Universiteit
The Netherlands

Jan Tretmans
Radboud Universiteit Nijmegen
The Netherlands

June 20, 2010

Abstract

We investigate the usability of Timed Model-Based Testing *in a case study: Conformance of the implementation* AUTOTRUST *with Automatic Trust Anchor Updating, a protocol to help securing DNS. We present models for timing aspects of the protocol. Definition of checks on quiescence in the model are proven to be possible. Latency during input application can yield false test results. We present a model to validate a security-relevant missing feature of* AUTOTRUST. *Reuse of an existing model of the protocol, designed for model-checking, is shown to be infeasible. Finally, we show that the implementation* AUTOTRUST *behaves according to the tested part of its specification.*

Contents

1 Introduction

Testing is a way of quality assessment of programs to determine if a program 'does what it has to do'. Testing is time-consuming, sometimes taking up to 50% of the overall development time. Therefore, Model-Based Testing uses models describing the desired behaviour of a program to automate this task. Generation, execution, and checking the results of tests are combined to simplify and speed up testing. With *Timed Model-Based Testing* [7, 12, 14, 8, 9, 17], the timing of test stimuli and reactions can be considered as well, yielding more areas to apply this technique to. Embedded systems, for example, mostly involve strict timing constraints. These could be tested using Timed Model-Based Testing. Fly-by-wire systems in modern air planes are an example for such embedded systems: If a pilot hits the brakes, the breaking system should engage almost immediately to ensure secure travelling. Timed Model-Based Testing is a way of black-box conformance testing. Thus, no knowledge about the internal workings of the program to be tested is used and the tests are limited to the functional and timing properties.

Timed Model-Based Testing is a recent technique that has not been used much. Is it applicable to a practical problem at all? TRON, a tool that implements this testing technique, was used in this case study to conduct timed testing. Up to now, it has been used in few case studies only. A refrigeration controller [12], a train gate [13], a smart lamp [15], and some illustrative examples [11] were tested using Timed Model-Based Testing with TRON as yet.

We tested an implementation, AUTOTRUST, of an administrative helper protocol for securing DNS. We focussed on timing aspects that are defined in the protocol and quiescence, meaning that no output is produced within finite time. We included checks on quiescence in the model instead of in the adapter – responsible for applying the tests to the implementation – as in other research [6].

We created models for specific parts of the protocol and showed that it is possible to include checks on quiescence in the model. Furthermore, we discovered that latency during input application can lead to false test results. We also developed a model to validate a security-relevant missing feature of AUTOTRUST. Moreover, reuse of an existing model of the protocol, designed for model-checking, proved infeasible. Finally, we showed that the implementation behaves according to the tested part of the specification. All in all, we showed that Timed Model-Based Testing is indeed usable for this case study.

In the following, we will give an introduction to Automated Trust Anchor Updating (Section 2) and the theory of Timed Model-Based Testing (Section 3). Then, we will describe our results in detail (Section 4), discuss limitations and possible generalisations of those (Section 5), and draw conclusions (Section 6). In the appendices, we show the models of all executed tests (Appendix A) and give instructions on how to install all necessary tools for testing (Appendix B).

2 DNSSEC Trust Anchor Updating – An Introduction

2.1 DNS

The *Domain Name System (DNS)* [16] is a hierarchical system whose main purpose is to translate domain names into IP addresses. DNS data are represented in a tree hierarchy with a root, top level domains (TLD), and domains at lower levels in the tree. This tree is logically divided into *zones*. Each zone consists of a coherent – possibly multi-level – part of the tree. DNS is a client-server system. The data tree is spread over many *DNS servers* providing information to *resolvers* at the client side that can ask for translation of domain names into IP addresses and other information.

2.2 DNSSEC

Domain Name Security Extensions (DNSSEC) [3] add data origin authentication, data integrity, and a way of *public key distribution* to DNS, as the data provided by DNS servers can easily be

tampered with [4].

Asymmetric key cryptography is used to digitally sign DNS data and thus achieve data origin authentication and data integrity. Asymmetric key cryptography differentiates between *public-* and *private keys*. One or more public/private key pairs are bound to a zone. Within the scope of DNSSEC, the latter are kept private and used to create digital signatures. The former are published and used to check if signatures created with the corresponding private keys are valid. Hence, public keys – *keys* for short – have to be distributed to all resolvers. For an introduction to asymmetric key cryptography see for example the book Computer Networks by Tanenbaum [19].

Keys can be distributed manually or automatically; Manual distribution is not considered here. Automatic, trustworthy key distribution through DNS servers makes use of the hierarchical structure of DNS. Keys situated higher in the hierarchy can vouch for keys beneath by signing them. Therefore, a resolver can try to build a *chain of trust* from a – yet untrusted – key up to a trusted key higher in the hierarchy. One manually distributed 'top' key, which is trusted by the resolver, could vouch for all keys down the hierarchy. All other keys could be distributed automatically without endangering security. Thus, the root or the TLDs of DNS would be signed at best. But neither the root nor most TLDs are signed yet[1]. Therefore, so called *islands of security* arise. Consequently, a resolver "may need to know literally thousands of [manually distributed] trust anchors to perform its duties" [18].

2.3 Trust Anchor Updating

Automatic Trust Anchor Updating (RFC 5011 [18], called *the RFC* hereafter) automates the secure adding and updating of keys at specific places in the hierarchy. It is not about the manual distribution of the first key. Updating and adding keys is necessary as on the one hand the lifetime of a key can expire. On the other hand, with more than one key at a specific place in the hierarchy, a compromised key can be replaced by revoking it and adding a new one.

A *trust anchor*, as defined in the RFC, is a public key bound to a specific point of a signed part of the DNS hierarchy. Still, we will use the term *key* to denote a public key throughout this document – regardless of it being a trust anchor or not.

We will focus on *key states* and *key events* from the resolver's point of view (Figure 1). Key events cause a key to change to another state. There are six key states:

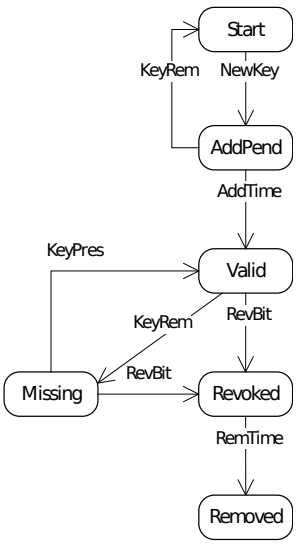


Figure 1: Key States of RFC 5011

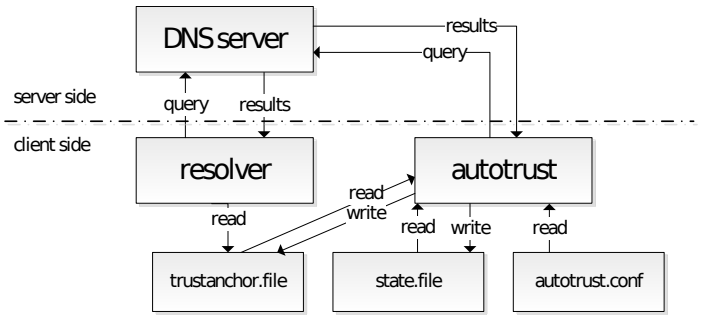


Figure 2: AUTOTRUST and its environment

[1]From 15th July 2010, this is not true any more. The root is now signed, see `www.root-dnssec.org`. But the protocol remains useful, according to Matthijs Mekking, because many TLDs are still not signed.

Start The key does not yet exist as a trust anchor at the client side. It has either not yet been seen at the client side or was seen but was absent in the result of the last query to the DNS server (KeyRem event).

AddPend The key has been seen and validated by (one or more) already existing trust anchors at the resolver. For security reasons, there is a hold-down time for the key before it can be used as a trust anchor.

Valid The key has been seen at the client side for at least the hold-down time and has been included in all validated results of queries to the DNS server. If the key has been used to sign other keys, the created signatures can now be validated with it.

Missing An abnormal state where the key remains a valid trust anchor but was not seen in the last validated result of a query to the DNS server. This happens for example if the key is removed from the DNS server without prior revoking.

Revoked The key has been revoked and must not be used as a valid trust anchor any more.

Removed After a hold-down time, all information about a revoked key may be removed. This state is more or less equivalent to the Start state, but in practice a removed key should not be re-introduced.

And there are six key events:

NewKey The client side sees a validated result of a query to the DNS server with a new key.

KeyRem At the client side a validated result of a query to the DNS server is seen, which does not contain this key.

KeyPres The key has reappeared in the DNS server query result.

AddTime The new key has been in every validated DNS query result for the add hold-down time.

RevBit The key has appeared in a DNS server query result marked as revoked.

RemTime A revoked key stayed in the revoked state for at least the remove hold-down time.

These descriptions are almost exact quotes from the RFC. Hold-down times when accepting or removing keys, solve security problems. See the RFC, section 2.

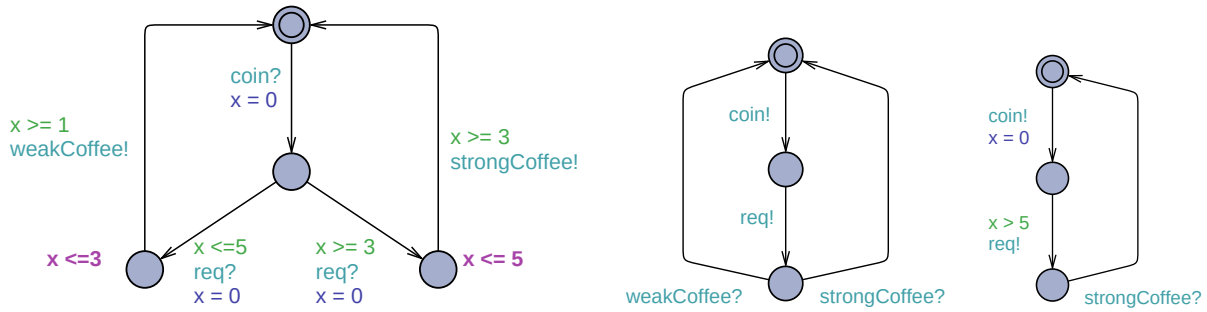
2.4 Autotrust

Autotrust is an implementation of Automated Trust Anchor Updating, developed by Matthijs Mekking at NLnet Labs[2]. We tested version 0.3.1 here.

Autotrust is used at the client side of a DNSSEC environment (Figure 2). The server side consists of a DNS server, whereas the client side comprises the resolver and Autotrust with its files. Autotrust.conf is the configuration file for settings concerning Autotrust. In the trust anchor file, all preconfigured trust anchors and keys that were received later and reside in the valid state are stored. The state file contains current state information on every key, including all trust anchors from the trust anchor file. The resolver can query the DNS server to translate domain names into IP addresses, among others. To validate the signatures of received results, it gets the appropriate trust anchors from the trust anchor file.

For each key, Autotrust queries the corresponding DNS servers for updated information and proceeds according to the transition system described above. For the scope of our case study, Autotrust has to be called explicitly to perform these updates.

[2] `www.nlnetlabs.nl`



(a) IUT model (b) ENV model 1 (c) ENV model 2

Figure 3: Coffee Machine

3 Timed Model-Based Testing

3.1 Overview

With *Timed Model-Based Testing*, automated test generation is based on a model that describes the specification of the *Implementation Under Test (IUT)*. Tests can either be generated *online (on-the-fly)* or *offline*. The first way combines generation, execution, and checking of the results at the same time. Thus, there are no intermediary test suites. With the second way, however, all tests are generated before they are executed. For a discussion of the (dis)advantages of both see [7].

Generated tests stimulate the IUT with inputs, which reacts with outputs that are then checked on their correctness with respect to the specification. Correctness includes the timing of inputs and outputs. If the output itself or its timing is unexpected, the test fails.

Timed automata [2] are the formalism used to describe the specification. They are finite state machines extended with clock variables – clocks for short – that increase automatically as time passes. First, clock variables may be reset when taking a transition. Furthermore, they may be used as guards on transitions, stating when a transition may be taken. Finally, clocks can be used with invariants for states, denoting how long a system may reside in a state. Semantics of I/O timed automata are defined in terms of a *timed input/output transition system (TIOTS)* [7]. This is a labelled transition system with inputs, outputs, and special delay labels as actions. From the IUT's perspective, inputs are suffixed with a '?' and outputs with a '!'.

A model of a simple coffee machine is shown as an illustrative example (Figure 3). The IUT model (Figure 3(a)) accepts a coin and a request for coffee (inputs). Depending on when the request for coffee is issued, weak or strong coffee (outputs) is produced. Clock x is reset when the input coin arrives. Clock guards determine which transitions can be taken based on the timing of the request ($x \leq 5,\ x \geq 3$). A clock invariant ($x \leq 3$) on the left side defines that this state must be left before a deadline. Note the non-determinism when the request is issued in the time interval [3,5].

A *conformance relation* between an implementation and its specification defines correctness with respect to a specification. Here, the *relativised timed input/output conformance relation (rtioco)* [7] is used. It is based on the *input/output conformance relation (ioco)* by Jan Tretmans [20]. rtioco extends ioco with explicit environment assumptions and takes timing into account. It is also used to derive test cases from a specification automatically.

$$i\ \mathbf{rtioco}_e\ s = \forall \sigma \in (s,e).\ \ out((i,e)\ after\ \sigma) \subseteq out((s,e)\ after\ \sigma) \tag{1}$$

Informally, equation 1 says: The implementation i is **rtioco** under environment e with respect to specification s means that "after any input/output trace possible in [...] the specification s and environment [...] e, the implementation i in environment e may only produce outputs and timed delays which are included in the specification s under environment e." [12].

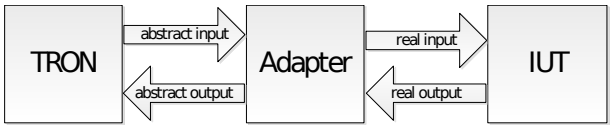


Figure 4: General TRON test setup

With rtioco, the testing model is partitioned into an IUT- and an environment part. The former describes the expected behaviour of the IUT in terms of stimuli (input) and reactions to those stimuli (output). The latter, however, replaces the real environment of the IUT and defines stimuli to apply to it. Due to this separation, testing can be limited to certain parts of the IUT (model).

Considering the coffee machine example (Figure 3), an environment model for testing the whole IUT model (Figure 3(b)) and one for testing only the strong coffee part of the model is given. The latter is achieved by issuing the request only after a minimum time of waiting ($x > 5$). The time unit for testing is not defined yet. Note that the notation for inputs and outputs in the environment models are interchanged compared to those in the IUT model.

Quiescence is used to denote silence and not internal stability as in the ioco theory [20]. It means that no output actions from the IUT can be observed [6] – the IUT remains quiet. Theoretically, this cannot be detected in finite time. Practically, however, the IUT is considered quiet if no output is observed after a finite time limit.

3.2 Uppaal-Tron

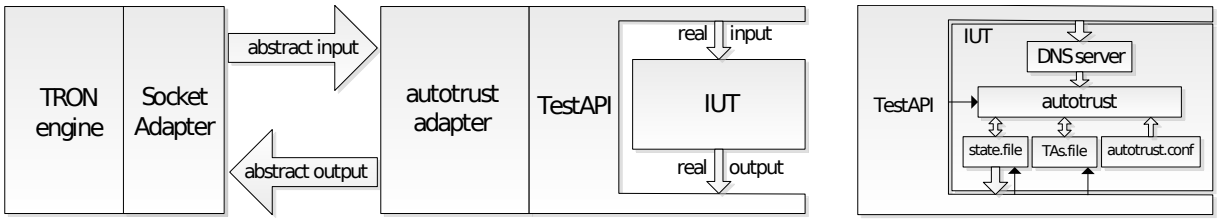
"UPPAAL is an integrated tool environment for modelling, validation and verification of real-time systems modelled as networks of timed automata"[3]. It provides an editor, a simulator, and a verification engine for timed automata. Defined timed automata can work in parallel and synchronise on matching input/output actions. The created models are templates that can be parametrised and instantiated multiple times. UPPAAL also comprises a powerful description language, among which committed states (when entered they must be left immediately), C-style functions, and data structures. A practical introduction is given in [5].

UPPAAL-TRON[4], TRON for short, is an extension to UPPAAL using timed automata models for Timed Model-Based Testing. It was formerly known as T-UPPAAL. We will use version 1.5-linux here. TRON is based on the rtioco theory described above. The specification model is partitioned into an IUT- and ENV part based on user-defined input-/output channels and testing is done on-the-fly. To generate tests, the testing algorithm first computes all possible input/output actions and allowed time delays in the current state. Then it chooses one of the inputs or delays or it observes an output and checks if it is correct according to the current state of the model. This is repeated until a user-defined time-out is reached or the test run fails. The result is one of 'pass' (time-out reached), 'fail' (incorrect – timing of an – output) or 'inconclusive' ("env model could not be updated with unexpected output from the IUT" [11]). For more details, see [7].

A test setup with TRON (Figure 4) consists of the TRON executable for test generation, execution and checking, an *adapter*, and the IUT. The adapter maps abstract inputs from the models to real inputs for the IUT and vice versa for outputs. It is always developed specifically for an IUT. Several example adapters come with TRON, including a *Trace Adapter* providing a text interface to test models manually. Others are developed open-source in JAVA and C++, demonstrating communication with TRON via sockets (TCP/IP) and employing thread programming for asynchronous input acceptance and processing.

A few parameters of TRON are important for the understanding of this document. Timing behaviour of TRON can be influenced to define how long to delay before an input is generated by the environment (parameter -P). Options are, among others, a random delay or the shortest

[3] www.uppaal.com
[4] www.cs.aau.dk/~marius/tron/



(a) Overview (b) IUT Details

Figure 5: Specific TRON Test Setup

possible delay (eager). Initialisation of the random number generator (-X) can also be configured manually. This results in almost exactly repeatable test runs, which is useful when running first tests.

TRON provides many more features, like lab conditions for timed testing (virtual time), which are explained in the user manual [11].

4 Results

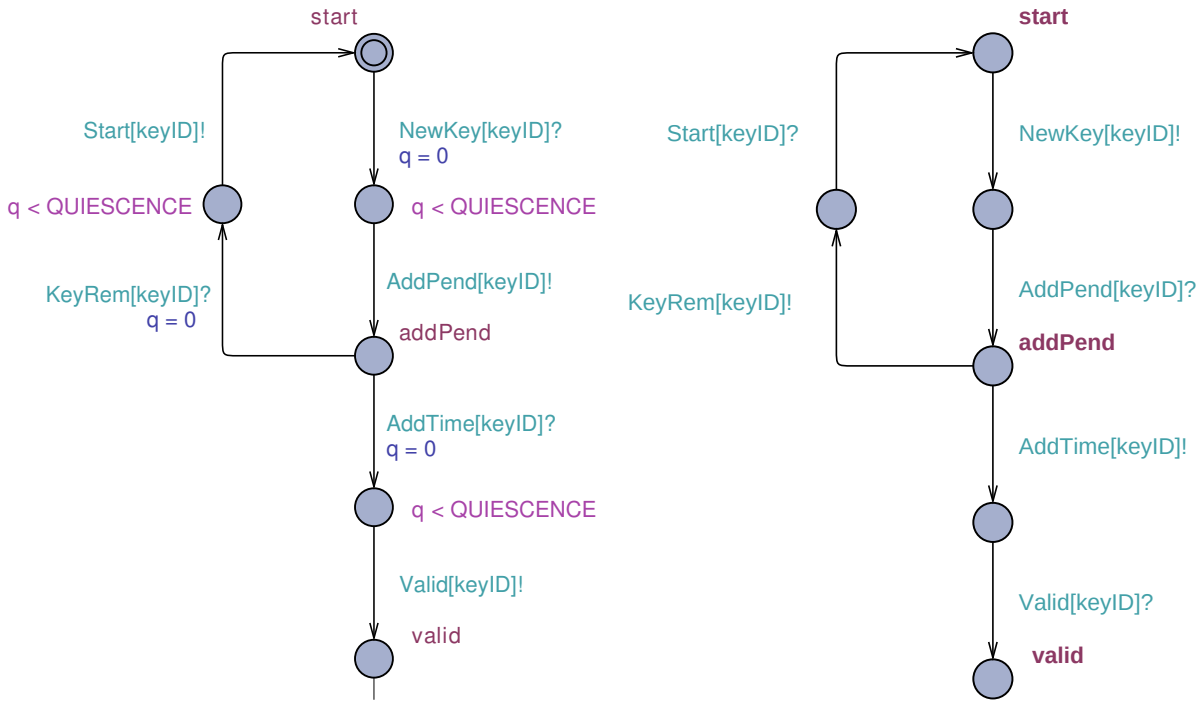
Having introduced the methods, we will present our results here. First, we will describe the developed adapter. Second, we will explain what problems occurred during the modelling process and solutions to them. Third, we will describe latency problems and their influence on testing. Fourth, we present a model validating a missing feature of AUTOTRUST. Fifth, we will discuss problems concerning the reuse of an existing model. Finally, we will comment on the used tools.

4.1 Adapter Development

The specific test setup for AUTOTRUST includes TRON and its internal Socket Adapter for communication, the AUTOTRUST *adapter* with a *Test API*, and the IUT (Figure 5(a)). Communication between TRON's internal Socket Adapter and the AUTOTRUST adapter is done via TCP/IP. The AUTOTRUST adapter and Test API communicate with function calls and the Test API communicates with the IUT via file reading and writing as well as via the command line.

The adapter and Test API translate abstract inputs into real ones and vice versa for outputs. The Test API abstracts from the technical details of input application and output analysis to or from the IUT, respectively. These inputs correspond to key events and the outputs to key states. Both of them are linked to keys, e.g. the input *NewKey[1]* denotes the key event NewKey for key 1 and the output *AddPend[1]* means that key 1 has reached the AddPend state. All key event inputs include a call to AUTOTRUST to make sure AUTOTRUST notices them. *ConfTrustAnchor[keyID]*, *ResetKey[keyID]*, and *CallAutotrust[keyID]* are additional inputs, not defined in the RFC. The first preconfigures a key as trust anchor by putting it into the trust anchor file; the second resets a key by removing it from the state file; and the third calls AUTOTRUST directly. Although the third one is linked to a key, within the adapter all calls to AUTOTRUST are the same regardless of the key they were issued for. Furthermore, input acceptance and processing in the adapter is done asynchronously to keep the adapter from blocking when processing a computationally complex input. Therefore, abstract inputs are first put into a buffer. Another thread then consumes the inputs and calls the appropriate Test API functions. The same thread also sends back abstract outputs.

The IUT comprises, besides AUTOTRUST, the DNS server, a state file, a trust anchor file (TAs.file), and a configuration file (Figure 5(b)). The resolver is not included in the IUT any more, because it is not concerned with updating keys. Big arrows in the figure depict the general information flow, small ones show additional interactions between the IUT and the Test API. These additional interactions are explicit calling of AUTOTRUST and direct manipulation of the



(a) IUT Key Model

(b) Environment Key Model

Figure 6: Intermediary States and Quiescence

state file and trust anchor file. Inputs are fed indirectly to AUTOTRUST through a DNS server. As we want to test AUTOTRUST only, we assume that the DNS server works correctly and neither the DNS server nor the above mentioned files are manipulated by any other process. As DNS server implementation we use *BIND*[5], which supports DNSSEC. Reloading DNS data – *zone reloading* – with BIND is only possible once a second, because BIND relies on file timestamps, which are exact to the second only. A workaround to reliably reload zone data involves repeated querying of the DNS server to check on a serial number which is increased with every update.

Testing is not yet fully automated. Depending on the type of test, different hold-down times have to be configured manually in the AUTOTRUST configuration file.

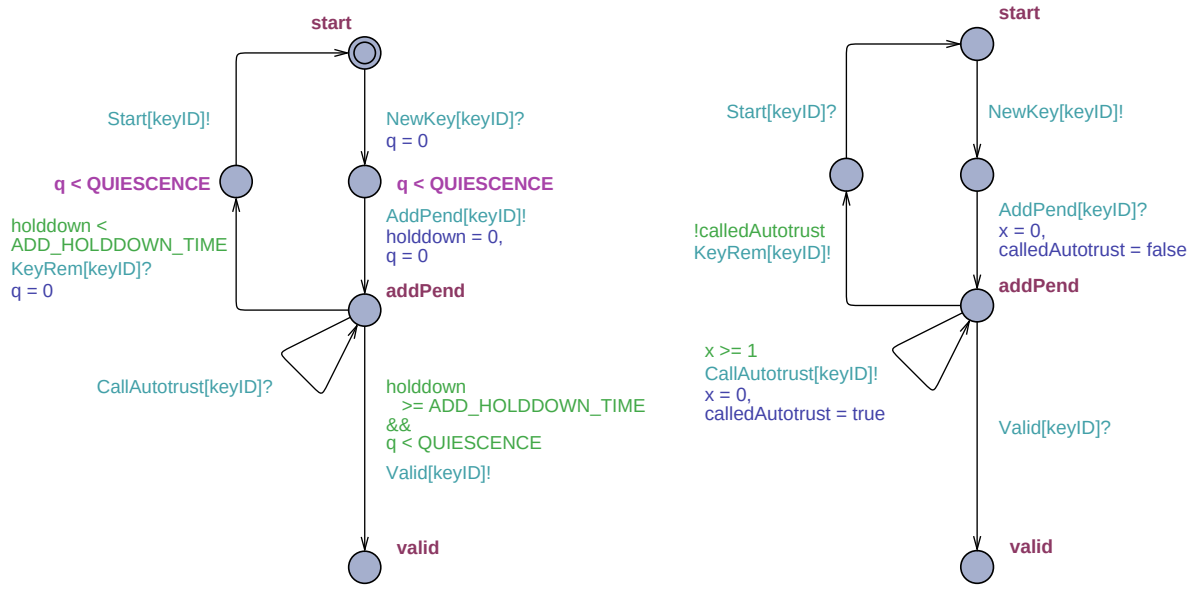
4.2 Intermediary States, Quiescence and Coverage

Models developed for testing the key state conformance of AUTOTRUST generally follow the state transition system of the RFC. As a convention for the models the state names start with a letter in lower-case, whereas the input and output names start with a letter in upper-case. Furthermore, we name a transition after the input or output action that is linked to it. A transition with output Valid[1] is thus called the *Valid transition*.

Anonymous intermediary states between each pair of key states are present in all models (Figure 6). To issue an input, we need to define a transition for it. Once AUTOTRUST reaches a certain state after it has consumed the input, the state change has to be communicated back to TRON by an output. But it is not possible to put two input (or output) actions on a single transition with I/O timed automata. Therefore, we need another transition to test for the occurrence of the output. Thus, we need an intermediary state between each pair of key states, which was unnecessary in the original state transition system of the RFC.

We encountered two situations in modelling quiescence: Either there is a time gap between the consumption of input and the generation of output or there is no time gap. No time gap means

[5] www.isc.org/software/bind



(a) IUT Key Model

(b) Environment Key Model

Figure 7: Quiescence With Time Gap

that zero time passes, not considering computation time.

Without a time gap, we reset the quiescence clock q in the IUT key model with every input arrival (Figure 6(a)). We expect the IUT to react within finite time (variable *QUIESCENCE*), so we introduced the state invariant $q < QUIESCENCE$ at the intermediary state. Now, the IUT has to leave the intermediary state before the time limit is reached by producing an appropriate output. Otherwise, the test fails. Clock q is local to the model, so every instantiation of the model has its own.

A time gap did occur when we replaced the artificial key events AddTime and RemTime with actual waiting for the hold-down time to pass (Figure 7). Proceeding from the AddPend state to the Valid state by an AddTime/Valid input/output combination is now substituted with a single Valid transition. This transition may only be taken after the add hold-down time has passed. Therefore, it includes the local *holddown* clock in the guard $holddown \geq ADD\_HOLDDOWN\_TIME$. We abstracted from the real hold-down time defined in the RFC and the one recommended by AUTOTRUST, which are in the order of 30 days, to make testing feasible. We chose three seconds, because on the one hand this is long enough for AUTOTRUST to do its work and on the other hand it does not slow down testing too much. As quiescence limit we chose ten second, as this is greater than the hold-down time but does not slow down testing too much either. Now, the quiescence clock q is reset when reaching the AddPend state. Successive calls to AUTOTRUST every second (Figure 7(b)) should eventually report the state change to the Valid state. If the quiescence limit has passed without any output, the test fails due to the quiescence guard on the Valid transition.

To achieve fair coverage of the whole model, it was necessary to limit the input choices in the AddPend state. For the Valid transition to be taken, AUTOTRUST has to be called after the hold-down time. Therefore, the test algorithm has to delay until after the hold-down time or has to issue multiple successive calls to AUTOTRUST. After that, AUTOTRUST needs to be called (again). There are three choices in the AddPend state of the environment model: *a*) wait, *b*) call AUTOTRUST after one second of waiting, or *c*) generate a KeyRem event. *a)* is not chosen, as we configured TRON not to wait before generating an input (-P parameter with option *eager*). We did not change this, as it is still a desirable setting for the rest of the model. Choices *b)* and *c)* are left, which are chosen with equal probability. But the KeyRem- and the Valid transition are not chosen with equal probability, because for the Valid transition to be taken, the algorithm has to choose option

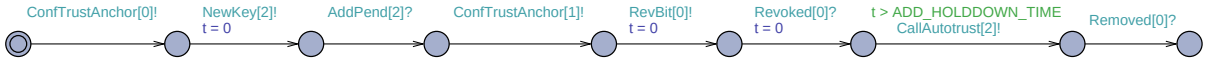


Figure 8: Missing Feature – Environment Model

b) at least n times in a row – for a hold-down time of n seconds. As there are n moments of choice between options *b)* and *c)*, the chance for option *b)* to be taken n times in a row is only 0.5^n if only one key is tested. To solve this, we limited the repeated moments of choice to just one by setting a boolean flag (*calledAutotrust*) if AUTOTRUST is called. Because of the guard *!calledAutotrust* on the KeyRem transition, the KeyRem input cannot be generated any more once AUTOTRUST has been called for the first time.

4.3 Adapter Caused Latency

Latency caused by computation time of the adapter is a problem we discovered while concurrently testing multiple keys. With latency, we mean the time between the sending of an input at TRON until its arrival at the IUT.

Say, testing has proceeded to the AddPend state for key x and the adapter has delivered all inputs to AUTOTRUST. TRON starts the add hold-down timer in the IUT model for key x. Then, other inputs are generated and put into the buffer at the adapter. Assume that for key x TRON then chooses the KeyRem event and it proceeds to the intermediary state in the IUT key model. There it awaits the output Start[x]. Processing of all buffered input actions at the adapter causes the hold-down time for key x to pass before the KeyRem input is delivered to AUTOTRUST. Now, there is a call to AUTOTRUST among the inputs before the KeyRem event for key x. The state change to the Valid state of key x is reported to TRON. But the IUT key model of key x cannot be updated with this – correct – output, as it already awaits the Start output for key x. This yields a false negative test result.

Suppressing the problem partially with the guard $holddown < ADD\_HOLDDOWN\_TIME$ on the KeyRem transition is possible (Figure 7(a)). Nevertheless, it does not completely solve the problem. An increased hold-down time would not completely solve the problem either. So the problem is still open.

4.4 A Missing Feature

Here, we will show how a strict environment can be used to validate a missing feature of AUTOTRUST. Furthermore, we will describe an extended IUT model that is able to detect the missing feature without a special environment.

The author of AUTOTRUST notes in the todo list for version 1.0 that it misses a security-relevant feature described in RFC 5011 [18]:

> If all of the keys that were originally used to validate this key are revoked prior to the timer expiring, the resolver stops the acceptance process and resets the timer.

We used a strict environment model (Figure 8) that drives the IUT key model(s) into a situation where it becomes obvious that AUTOTRUST misses this feature indeed. Furthermore, we used an IUT model that is incapable of detecting the missing feature directly (a part of it is shown in figure 7(a)).

For testing, three keys are used. Key 0 is preconfigured as a trust anchor; Key 2 is introduced as a new key and was thus signed with key 0. After that, key 1 is introduced as a second trust anchor and the first trust anchor, key 0, is revoked. At this moment, AUTOTRUST should have also reported, that the acceptance process for key 2 was stopped, by reporting its state change to the Start state. As we do not accept this output in the environment model, the test would have failed in this case. But the output is not generated. The next call to AUTOTRUST after the add

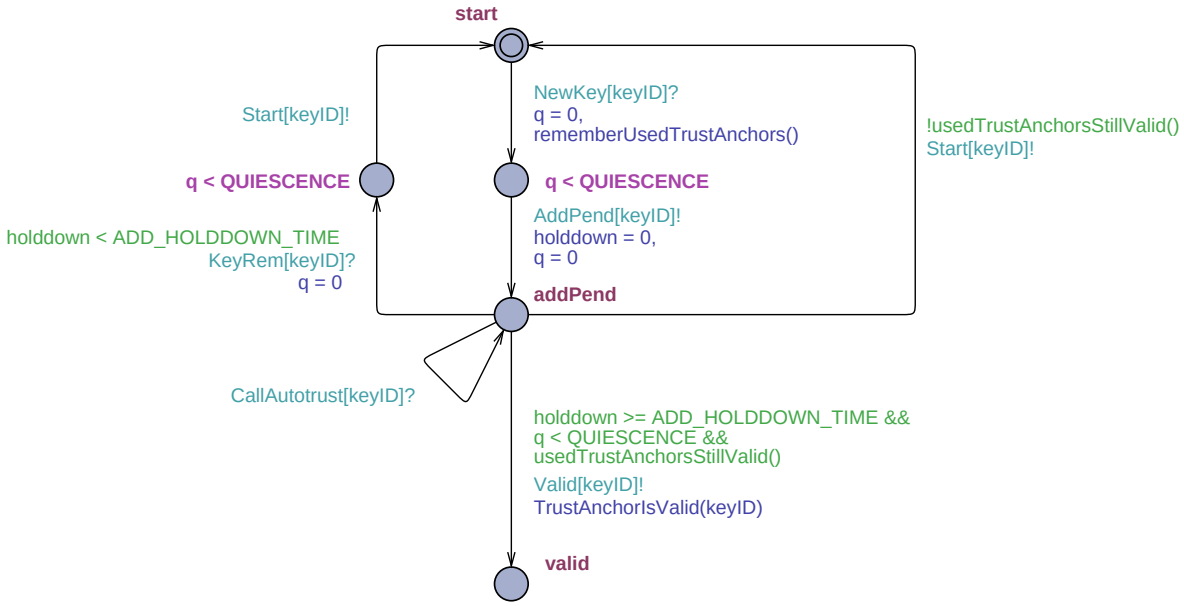


Figure 9: Missing Feature – IUT model

hold-down time should only report that key 0 reached the Removed state, because we configured the add and remove hold-down time the same. However, AUTOTRUST also reported that key 2 reached the Valid state and the test failed.

Automatic detection of the missing feature would have been possible with an IUT model using the extended modelling capabilities of UPPAAL (Figure 9). We present this model as an idea of how to model this specific part of the RFC; We did not fully implement nor tested AUTOTRUST with it. In the model, UPPAAL's C-style functions are used, which can be called when a transition is taken, as well as be used in guards. When a new key is introduced, all trust anchors used to sign it are remembered (`rememberUsedTrustAnchors()`). From the AddPend state, the key may only proceed to the Valid state if all trust anchors used to sign this key are still valid (`bool usedTrustAnchorsStillValid()`). On proceeding to the Valid state, the function `TrustAnchorIsValid(keyID)` is called to store the fact that the key is now a valid trust anchor. If not all trust anchors are valid any more (`!usedTrustAnchorsStillValid()`), the newly introduced key changes to the Start state again. A global data structure could be used by the described functions to communicate whether a key is currently a valid trust anchor.

4.5 Model Reuse Difficult

Aarts, Houben, and Uijen developed a model for checking the protocol of auto trust anchor updating [1]. This was done for a case study on model-checking with UPPAAL. The developed model includes the server side with a DNS server and the client side with a resolver and key states for several keys.

Reusing this model for Timed Model-Based Testing proved difficult. First of all, the model was designed for model- checking, not Model-Based Testing. Therefore, there are no intermediary states included which separate the sending of a key event from the reception of a state change from the IUT. Therefore, their model is closer to the original state transition system in the RFC (Figure 1). Furthermore, the model was too complex to start testing with. It involves exact descriptions of interactions between the DNS server and the client side. But we wanted to focus on the IUT and test only a small part of it.

Thus, we did not further investigate the reuse of this model.

4.6 Used Tools Did Well

First of all, AUTOTRUST passed all tests, except for the missing feature. Documentation on AUTOTRUST was good and AUTOTRUST was easy to use with a manipulatable DNS server. Second, TRON was suitable for all executed tests with respect to timing. Support for broadcast channel synchronisations would have been helpful, especially to model a single call to AUTOTRUST having an effect on all IUT key models. Although broadcast channels are supported by UPPAAL, TRON is not yet capable of deriving tests from models using these [11].

5 Discussion

5.1 Adapter Development

Key reintroduction, possible with the ResetKey input, is not recommended in the RFC in practice. For testing, however, it enabled us to test key state conformance of a single key multiple times in one test run.

Indirect feeding of input is also worse than feeding it directly to the implementation. But implementing a mechanism to recognise DNS queries from AUTOTRUST and answering them correctly, would have been more difficult and error prone. We chose the first solution, because it was more feasible.

BIND was given preference to NSD[6], another DNS server implementation, to ensure independence from AUTOTRUST: NSD is developed by the same organisation the author of AUTOTRUST works for.

To speed up zone reloading and thus the whole testing process, we could have manipulated BIND directly. But, to the best of our knowledge, there is no way to force BIND to reload zone data more than once a second with a parameter or other configuration options. Thus, we would have had to change the (open-)source code of BIND. We did not try this yet. Slow testing because of technical limitations can be considered as a more general problem with other projects as well.

Expenditure of time for adapter development was more than 50% of the overall testing process. The current adapter and TestAPI consist of more than 2,000 lines of code. Thus, much time had to be spent before actual testing could take place. However, with manual testing on this level of abstraction, a similar TestAPI would have to be developed. Thus, the necessary effort is not a disadvantage of Model-Based Testing itself.

5.2 Intermediary States, Quiescence and Coverage

The need for intermediary states between input and output combinations that we discovered will more generally occur with tests for state conformance of other implementations as well. This is because with I/O timed automata a transition can only represent one input or output.

Concerning quiescence, the described results are valid only if the time of input consumption is controllable. With that we mean that the actual time of input reception at the IUT can be controlled. Say, on the contrary, that AUTOTRUST would periodically call the DNS server, instead of being explicitly told to do so (this is actually possible by running AUTOTRUST as a daemon). We could still manipulate the DNS server, i.e. generate input, but could not control the consumption of this input by AUTOTRUST. We had to account for this by choosing a different quiescence time limit. Thus, with the presented solution the start of quiescence time measurement may be too early, yielding false test results.

Resetting the quiescence clock is possible at two alternative moments for the quiescence with time gap case (Figure 7(a)), yielding three possibilities altogether: *a*) *before* the first call to AUTOTRUST, as in the presented solution; *b*) *at* the first call to AUTOTRUST, after one second of waiting; or *c*) with *each* call to AUTOTRUST.

[6] `nlnetlabs.nl/projects/nsd`

With case *a)* – theoretically – there is a flaw. Say, testing has proceeded to the AddPend state and no call to AUTOTRUST is generated before the quiescence time limit. By then, $q \geq QUIESCENCE$ holds and the Valid transition cannot be taken any more. Now, AUTOTRUST is called, reporting the Valid state change, resulting in a false negative test result.

Resetting the clock *at* the first call of AUTOTRUST, case *b)*, seems better. But it has a similar problem as option *a)*. Assuming AUTOTRUST is called once before q has reached the hold-down time, and the next time when $q \geq QUIESCENCE$. The second call would report the correct state change, but due to the quiescence guard on the Valid transition it is rejected, resulting in a false negative test result as well.

Option *c)* presents a problem as well: Say, AUTOTRUST would never produce the Valid output at all. According to the model, AUTOTRUST can be called infinitely often from the AddPend state until testing times out. It would not be noticed that the Valid output has not been produced, yielding a false positive test result. Generally, this holds for both other options as well.

In the presented solution, we circumvented the theoretical flaw of *a)* by forcing the test algorithm to generate inputs without delay (-P parameter). The problem described for *c)* could be solved by forbidding calls of AUTOTRUST after the quiescence time limit. Forcing the AddPend state to be left before a finite time limit by adding an invariant could also solve the problems of *a)* and *b)*.

The described chance for the Valid transition to be taken, in order to achieve fair coverage, is strictly limited to testing only one key. More instantiated models would generate independent calls to AUTOTRUST and increase this chance. However, for only one key, this result can be generalised to k input options and the need to wait n time units. The chance for a single input option to be chosen n times is $(\frac{1}{k})^n$.

5.3 Adapter Caused Latency

A possible solution to the latency problem is to include the adapter and its latency in the test models [11]. Adding latency time intervals to time constraints in the original specification could also solve the problem partially, but it would make testing less exact.

Latency can also cause *false positive* test results [6]: If the model allows for an input i to be generated at time $t \in [a, b]$ but latency time d causes the input to arrive at the IUT at $t \geq a + d$, the test can never check input i for the time interval $[a, a + d]$. Even worse, input i might never arrive before b, if $d > b - a$. These are serious problems for test coverage.

6 Conclusions

We were able to show that Timed Model-Based Testing is indeed usable for a practical problem. But the technique also involves problems, which should be investigated more closely.

Nevertheless, Timed Model-Based Testing proved to be a suitable for testing timed requirements. Therefore, it can be considered as a promising technique that is applicable to embedded systems as well.

Compared to manual testing, the technique adds the ease of automated test generation, which increases test coverage in many cases.

Further research concerning this case study could include more concurrency in testing key states by separating key events from immediate calls to AUTOTRUST. This would increase interleaving of key events from different keys. Furthermore, testing could be extended to the whole RFC, including more detailed tests. Moreover, an experiment on latency caused by the developed adapter could reveal test coverage problems. Analysing timing coverage of tests a posteriori could show if varying ranges of input choices with respect to timing were covered with the executed tests.

More general, investigating timing coverage and latency of TRON could be part of further research. Furthermore, comparing the modelling capabilities usable with TRON to those usable with similar

tools could be interesting. Finally, discovering possible rules to develop models for both Model-Based Testing and model-checking could make (Timed) Model-Based Testing even more usable.

7 Acknowledgements

First, I want to thank my supervisors Julien Schmaltz and Jan Tretmans for fruitful discussions and all their help with this project. Special thanks to Julien for his patience and for supporting me a little longer than planned. Furthermore, I want to thank Matthijs Mekking, the developer of AUTOTRUST, for his advice and comments on this thesis. Finally, I want to thank all friends who helped me with discussions – of technical and non-technical nature – and hints for improvement.

A All Tests

We will describe all developed and executed tests and their models here, except for the missing feature test. For the latter, all relevant parts of the model were shown in the results section already. Focussing on key state conformance, we started with a single key, then we modified this test to include more timed testing, and finally, we concurrently tested multiple keys.

For all models, we did a 'no deadlock' check with UPPAAL to make sure that testing is not influenced by a technical modelling error. Furthermore, we used the UPPAAL simulator to manually verify that every state of the models can be reached and every transition can be taken; Again, to avoid influence on testing, as unreachable parts of the models could cause unnoticed skipping of tests. With the tests described in the following, we used a time unit of 1 second. Smaller time units would not have speeded up testing due to the DNS zone reload problem. The tests were generally executed with a fixed random seed to get repeatable results. Changing this could be part of future work.

A.1 Single Key Test

For the single key test, an IUT key model (Figure 10(a)) and a corresponding environment model (Figure 10(b)) are made, the latter allowing for testing every part of the IUT key model. All models follow the structure of the key state transition system of the RFC (Figure 1). Although we want to test only one key, two instantiations of the IUT key model are needed: The first one acts as a preconfigured trust anchor to sign the second one, whose key states are then tested. But the environment model is instantiated only once. To allow for multiple instantiation for different keys, the IUT key model is parametrised with a *key ID*.

In the environment model, key 0 is made a preconfigured trust anchor (ConfTrustAnchor[0]); the rest of the model is almost a mirrored version of the IUT key model, to test all possible key events. Contrary to the IUT key model, the environment model is not parametrised but refers to instantiations of the IUT key model. Furthermore, the environment model involves non-deterministic choices in the AddPend, Valid, and Missing states.

A.2 Single Key Test with Timing

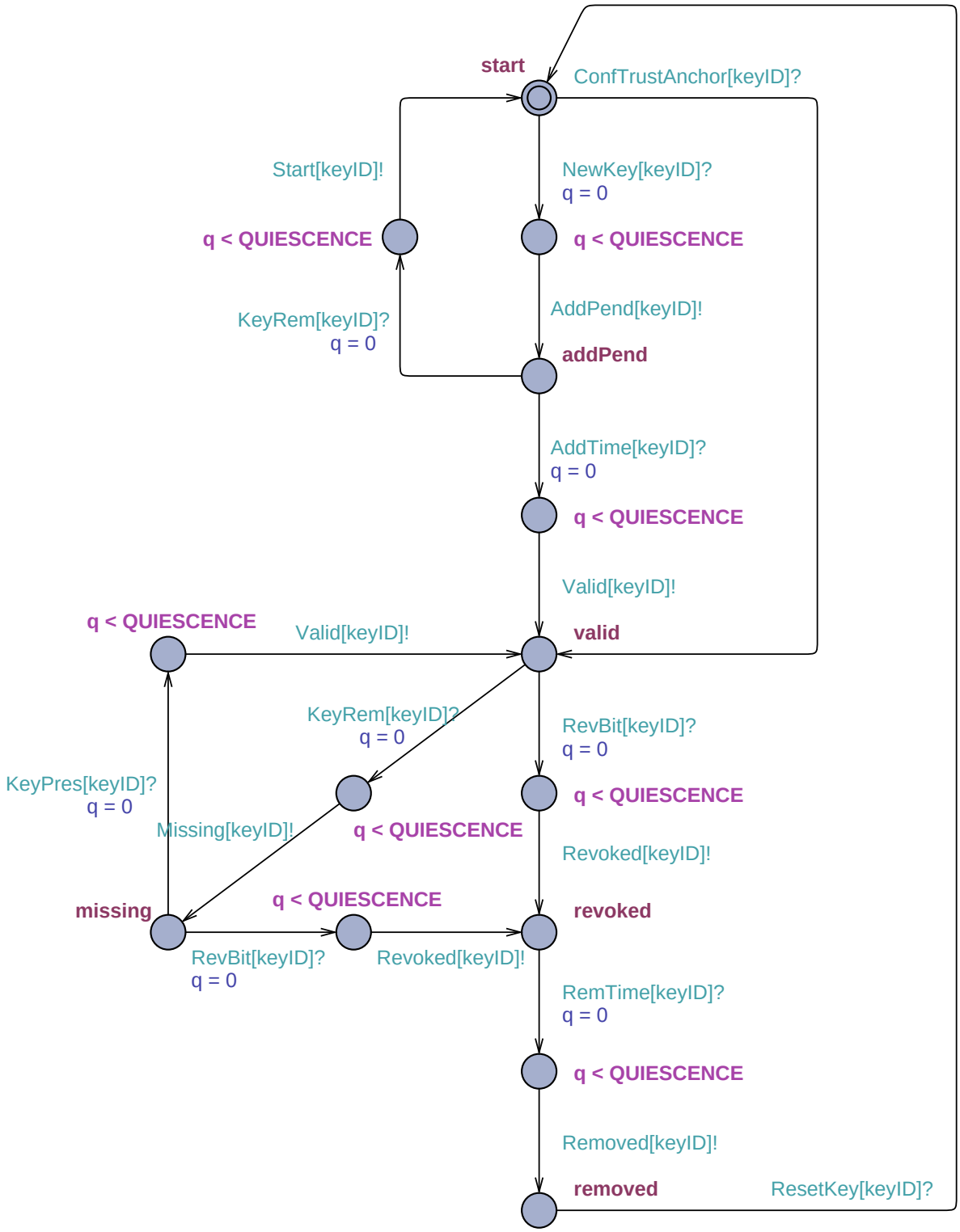
Compared with the single key test, we replaced the 'artificial' key events AddTime and RemTime with actual waiting for the hold-down time to pass (Figure 11). For the rest, the test is the same.

Proceeding from the AddPend state to the Valid state by an AddTime/Valid input/output combination is now substituted with the Valid transition only. A guard denotes that the transition may only be taken after the add hold-down time has passed ($holddown \geq ADD\_HOLDDOWN\_TIME$). The same holds for the transition from the Revoked state to the Removed state. To make testing feasible, we abstracted from the real hold-down time defined in RFC 5011 and the one recommended by AUTOTRUST, which are in the order of 30 days. We chose three seconds, as this is smaller than the quiescence time limit but did not slow down testing too much.

In the environment model, the part between the AddPend and Valid state was changed accordingly. As AUTOTRUST has to be called explicitly to report key state changes, it is called repeatedly in the loop transition at the AddPend state. This transition is provided with a guard ($x \geq 1$) to limit possibly huge number of successive AUTOTRUST calls to a single one per second. Again, an equivalent explanation can be given for the transition between the Revoked and Removed state.

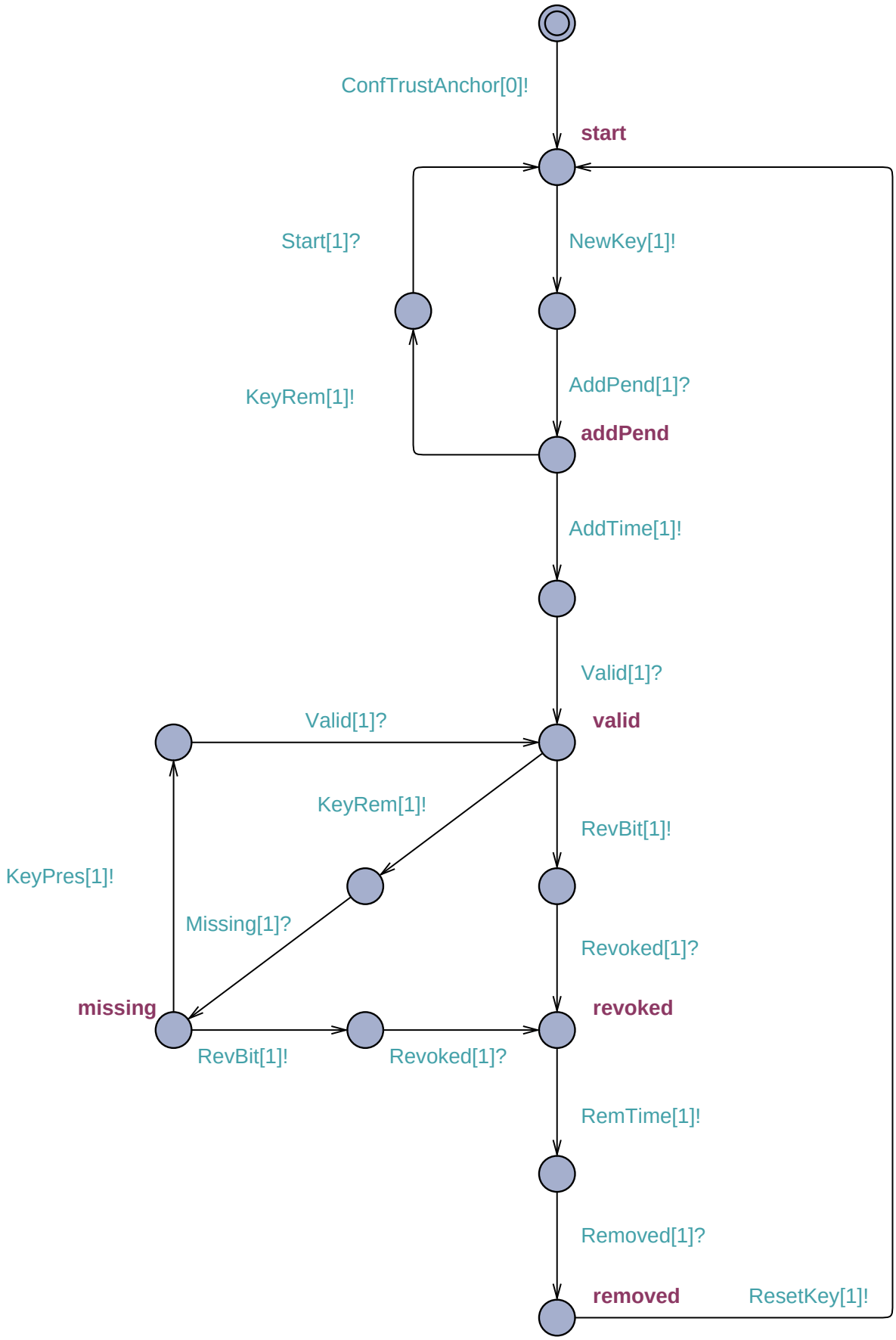
A.3 Multiple Keys

For concurrent testing of multiple keys, we reused the IUT key model and the environment key model of the single key test with timing (Figure 11). The IUT key model is left unchanged, but the former environment key model is now also parametrised with a key id (Figure 12). This is because five IUT key- and environment key model pairs are instantiated. A choice between the



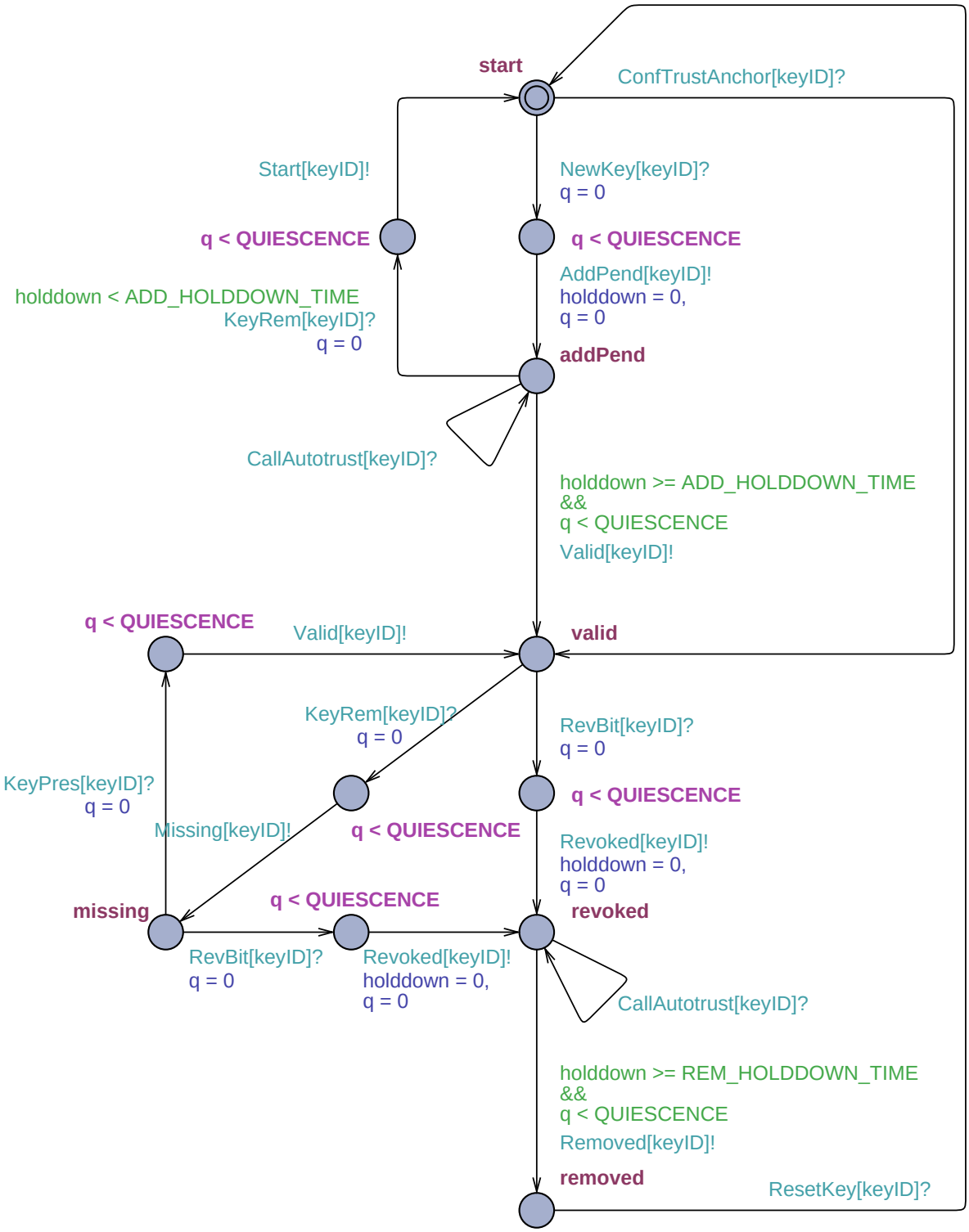
(a) IUT Key Model

Figure 10: Single Key Models



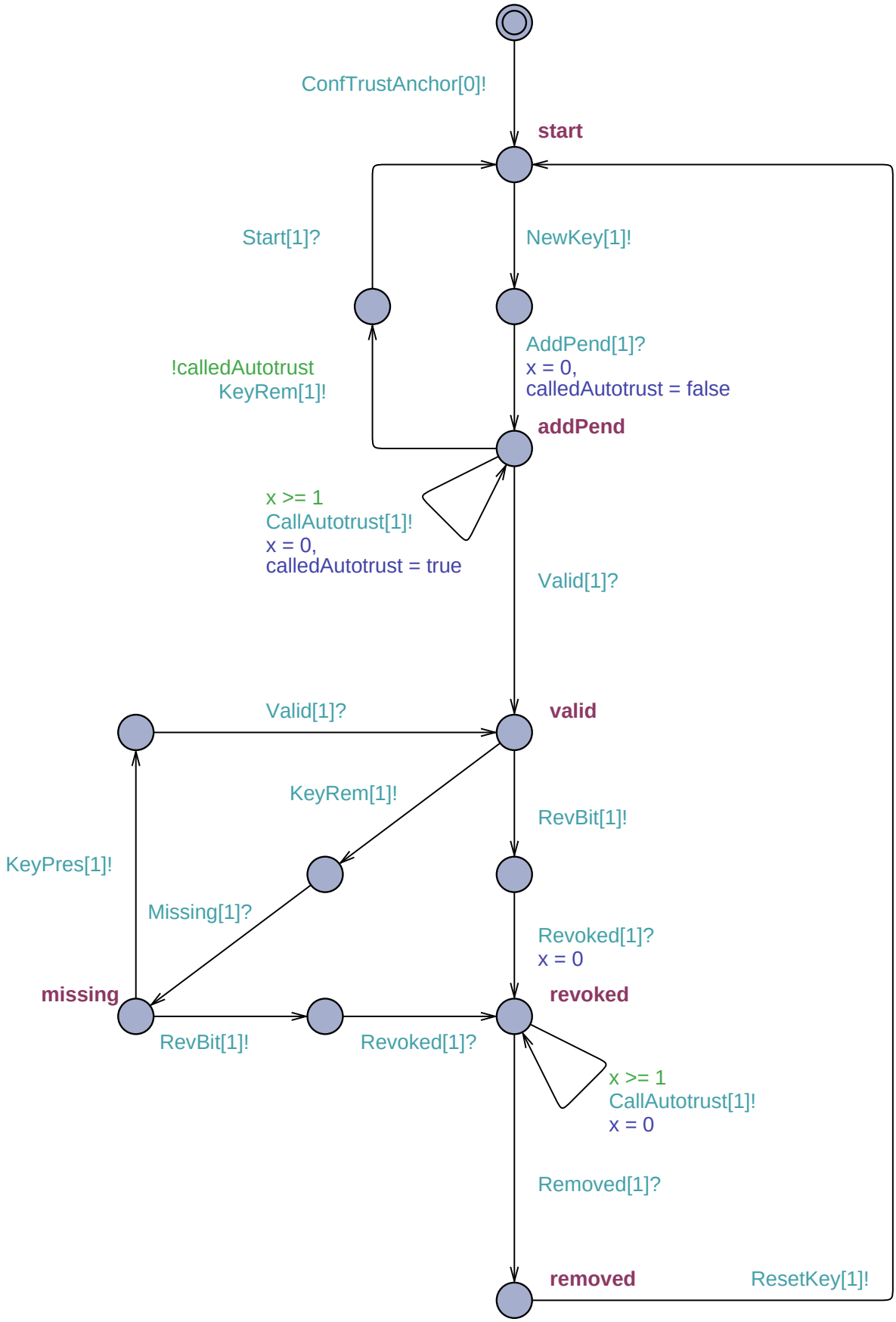
(b) Environment Key Model

Figure 10: Single Key Models (con't)



(a) IUT Key Model

Figure 11: Single Key Models with Timing



(b) Environment Key Model

Figure 11: Single Key Models with Timing (con't)

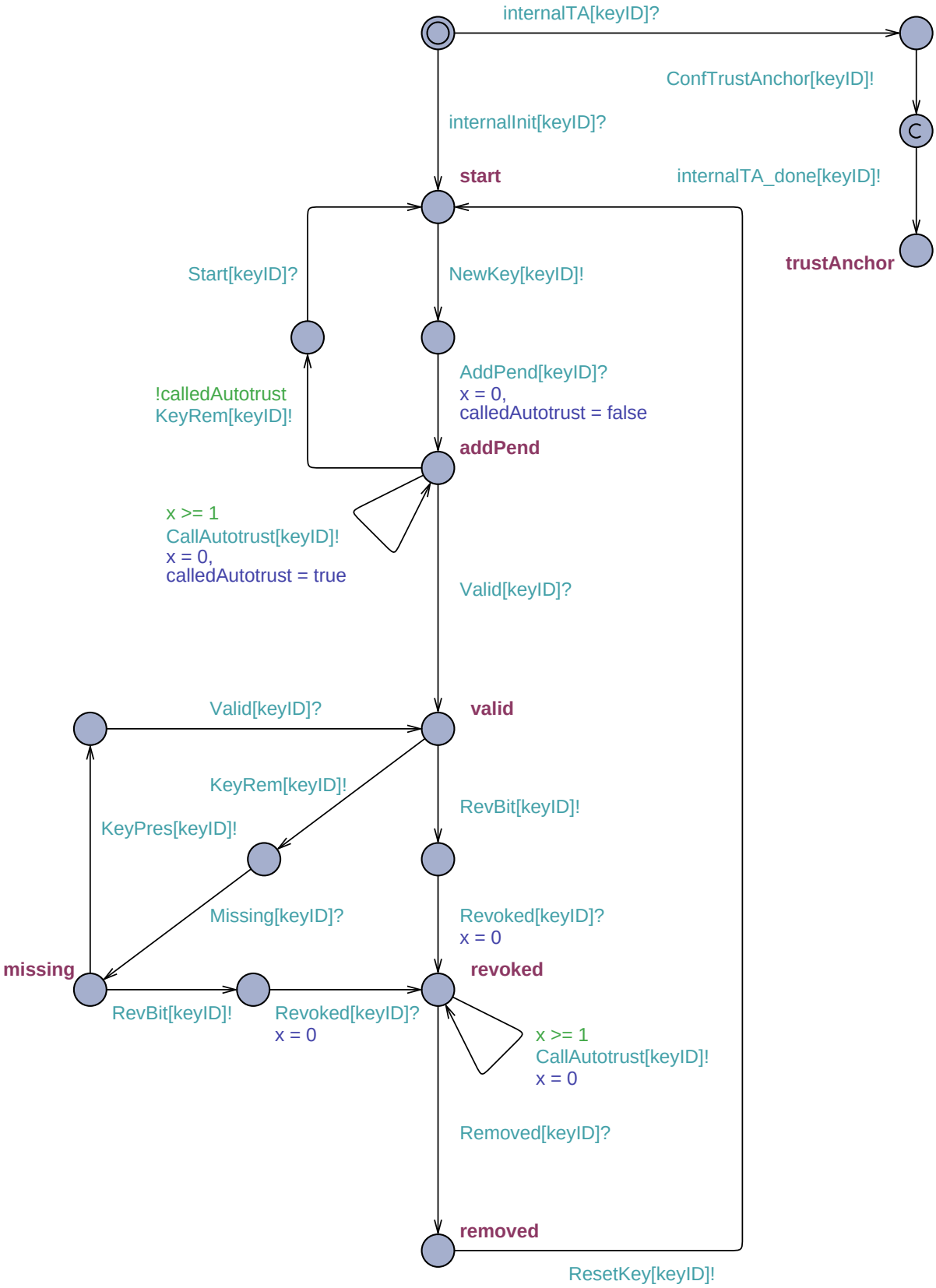


Figure 12: Multiple Keys – Environment Key Model

configuration as a trust anchor or as a usual key is added. The first means that the key remains a trust anchor for the rest of the test run, while the second leads to the same full key state test as before.

A new general environment model (Figure 13) starts up the test for each key. It configures key 0 as a preconfigured trust anchor and key 1-4 as usual keys. For that, internal transitions – they are neither configured as inputs nor outputs – are used. We chose to test five keys at the same time, because RFC 5011 requires support for managing at least 5 keys at the same time. Nevertheless, our test setup is still very limited concerning this requirement, because we used only one preconfigured trust anchor to sign other keys. We did not revoke this trust anchor nor use other keys residing in the Valid state for signing.

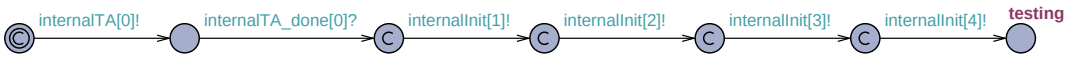


Figure 13: Multiple Keys – Environment Model

B Test Setup

We will describe the installation of all necessary programs and files to reproduce the tests on AUTOTRUST. Installation is limited to Linux systems, because AUTOTRUST itself, the AUTOTRUST adapter, and the TestAPI are developed specifically for it. In detailed steps, we will explain the installation on a Ubuntu 10.04 (Lucid Lynx) Linux distribution, referred to as Ubuntu hereafter. For other Linux systems, check if similar packages are available. Otherwise, a manual installation is necessary. Except for some hints, we refer to the installation instructions of each piece of software, in the latter case.

First, we will show which specific test files are needed. Then we will describe the installation of the DNS server BIND and its configuration. Furthermore, we will install the PStreams library for calling programs from C++ code, followed by the installation of AUTOTRUST itself. Finally, we will install the test tool TRON.

B.1 Test Files

A test file package with all models, the adapter, and the TestAPI is available at `www.cs.ru.nl/~julien/Julien_at_Nijmegen/rutz_bt.html`; Download and extract it. You should find a directory called `autotrustTesting`. All models used for testing are in the subdirectory `models`; AUTOTRUST configuration- and working files are stored in the subdirectory `autotrust`. The source code of the adapter is in the `autotrustTesting` directory itself and the TestAPI source code in the `autotrustTestAPI` subdirectory. There is documentation in the `doc/` subdirectories of both.

B.2 BIND and the Zone Files

We will describe the installation of BIND and the tool rndc that is used to administrate BIND. Unbuntu provides version 9.7.0, at least version 9.x is required. For a manual installation, download BIND from `www.isc.org/software/bind` and compile it with crypto support (- -with-openssl).

Installing BIND on Ubuntu is done with the following command:

```
sudo apt-get install bind9 bind9utils
```

For the configuration, we assume the configuration files are in `/etc/bind/`. If this differs on your system, change it in the following instructions accordingly. Unfortunately, you will have to change it manually in the code of the TestAPI as well (`autotrustTesting/autotrustTestAPI.cpp`).

To configure a sample zone, add this to `/etc/bind/named.conf.local`:

```
zone "example.local" {
  type master;
  file "/etc/bind/db.example.local.signed";
};
```

Furthermore, copy the file `autotrustTesting/zoneData/example.local.withKeys.signed` to `/etc/bind/db.example.local.signed`. Then restart bind to see if the configuration is accepted:

```
sudo /etc/init.d/bind9 restart
```

To make the administration tool rndc communicate with BIND, we create a key that is made known to BIND and rndc.

```
sudo rndc-confgen -a
      -c /etc/bind/rndc.conf
      -k dnsadmin -b 128
```

creates a configuration file `rndc.conf` in `/etc/bind` with a key for a user *dnsadmin*. Add an options section to this file, where *YourSecret* is the key generated for you, to make it look like:

```
key "dnsadmin" {
   algorithm hmac-md5;
   secret "YourSecret";
};
options {
   default-key "dnsadmin";
   default-server 127.0.0.1;
};
```

To make this key known to BIND and restrict connections with BIND to that key and the local host, we add the following to `/etc/bind/named.conf.options`, where *YourSecret* is the one from above.

```
key "dnsadmin" {
   algorithm hmac-md5;
   secret "YourSecret";
};

controls {
   inet 127.0.0.1 port 953
   allow { 127.0.0.1; } keys {"dnsadmin";};
};
```

Finally, restart BIND to finish the configuration:

```
sudo /etc/init.d/bind9 restart
```

To test the configuration, we make sure that rndc is working, by executing

```
sudo rndc status
```

which should give some status output of the server, with on the last line *server is up and running*. To see if we can reload the example.local zone, execute

```
sudo rndc reload example.local
```

which should give the output *zone reload up-to-date* as we did not change the zone file since the last restart of BIND.

B.3 PStreams Library

For Ubuntu, the installation is completed with one command:

```
sudo apt-get install libpstreams-dev
```

For a manual installation, visit the website[7] of the library.

B.4 Autotrust

AUTOTRUST depends on some DNS libraries which we will install first:

```
sudo apt-get install libldns1 libldns-dev ldnsutils libunbound2 libunbound-dev
```

For a manual installation, get ldns[8] and Unbound[9] and see the included documentation for an installation procedure. As Unbound is a DNS server as well, make sure to permanently stop the server to avoid interference with BIND.

As we want to test AUTOTRUST, we will install it from the sources; get it from the AUTOTRUST website[10]. In this case study we used version 0.3.1. Extract the downloaded archive and change to the extracted directory. Then run:

```
./configure
make
sudo make install
```

to compile and install AUTOTRUST. If configure does not find libldns and/or libunbound, see the file `doc/HOWTO` section 2c for details on how to make the libraries known to the configure script.

Testing the installation by calling

```
autotrust
```

on the command line, should result in AUTOTRUST finishing with the line *[autotrust] exit status: error loading configfile* which is all right for now.

B.5 Tron

For this case study, we used version 1.5 of TRON[11]. Get the archive and extract it. Then copy the `autotrustTesting` directory from above into this extracted directory.

Finally, copy the files `Executable.cpp, SocketReporter.cpp, SocketReporter.h`, and `sockets.cpp` from the `button-remote` directory to the empty directory `autotrustTesting/tron-src`.

To test the installation, run the following command in the TRON directory:

```
cd autotrustTesting
sudo make
```

This should start the first test. We need to use super user privileges, because testing involves manipulating the DNS server. For other tests, see the `Makefile` for make targets.

[7] pstreams.sourceforge.net
[8] nlnetlabs.nl/projects/ldns
[9] unbound.nlnetlabs.nl
[10] nlnetlabs.nl/projects/autotrust
[11] www.cs.aau.dk/~marius/tron

References

[1] Fides Aarts, Fred Houben, and Johan Uijen. Case study: Automated updates of DNS security (DNSSEC) trust anchors. Technical report, Radboud University Nijmegen (NL), January 2009.

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

[3] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033 (Proposed Standard), March 2005.

[4] D. Atkins and R. Austein. Threat Analysis of the Domain Name System (DNS). RFC 3833 (Informational), August 2004.

[5] G. Behrmann, A. David, and K.G. Larsen. A tutorial on uppaal. *Formal methods for the design of real-time systems*, pages 200–236, 2004.

[6] HC Bohnenkamp and AFE Belinfante. Timed model-based testing. *Tangram: Model-based integration and testing of complex high-tech systems*, pages 115–128, 2007.

[7] A. Hessel, K. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Testing real-time systems using uppaal. *Formal Methods and Testing*, pages 77–117, 2008.

[8] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. *Model Checking Software*, pages 109–126, 2004.

[9] M. Krichen and S. Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.

[10] K.G. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using uppaal. *Formal Approaches to Software Testing*, pages 79–94, 2005.

[11] K.G. Larsen, M. Mikucionis, and B. Nielsen. Uppaal Tron User Manual. 2009.

[12] K.G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou. Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In *Proceedings of the 5th ACM international conference on Embedded software*, page 306. ACM, 2005.

[13] M. Mikucionis, K.G. Larsen, and B. Nielsen. *Online on-the-fly testing of real-time systems*. Citeseer, 2003.

[14] M. Mikucionis, KG Larsen, and B. Nielsen. T-uppaal: Online model-based testing of real-time systems. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 396–397, 2004.

[15] M. Mikucionis and E. Sasnauskaite. On-the-fly Testing Using UPPAAL. 2003.

[16] J. Postel and J.K. Reynolds. Domain requirements. RFC 920, October 1984.

[17] J. Schmaltz and J. Tretmans. On conformance testing for timed systems. *Formal Modeling and Analysis of Timed Systems*, pages 250–264.

[18] M. StJohns. Automated Updates of DNS Security (DNSSEC) Trust Anchors. RFC 5011 (Proposed Standard), September 2007.

[19] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 2002.

[20] GJ Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 3(TR-CTIT-96-26), 1996.