Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 264 (2010) 53–68
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Present and Absent Sets: Abstraction for Testing of Reactive Systems with Databases

Petur Olsen, Kim G. Larsen, and Arne Skou[1]

Department of Computer Science
Aalborg University
Aalborg, Denmark

Abstract

We present a new abstraction of reactive systems interacting with databases. This abstraction is intended to be used for model-based testing. We abstract the database into two sets: present set and absent set, and present a proof of this abstraction. We present two extensions of FSM, the DBFSM and PAFSM. DBFSM are a form of FSM incorporating databases. PAFSM are an abstraction of DBFSM using present-absent sets. Depending on what type of testing is to be done, the translation is tailored to fit this purpose. We show how this translation is related to the present-absent abstraction. Finally, we illustrate the approach through a small example and show how this can be used for testing with the model-based testing tool Uppaal TRON.

Keywords: testing, model checking, abstraction, model-based testing

1 Introduction

Testing is generally considered the most widely used technique for error detection in software systems. Many systems today are heavily dependent on databases, there is however no efficient technique for testing systems using databases available. Several problems arise when testing systems dependent on databases. For instance, the test executed is dependent on the state of the database. Consider a test case requiring a user to be created. Running this test case after the user has been created is not possible without deleting the user first. Another problem is the huge amount of data stored in such databases.

Recently automated techniques and formal approaches have been developed for testing. One such being model-based testing (MBT) [1,6,7]. Doing MBT of a database systems is not trivial however. Consider modeling the entire database, this

[1] Email: petur@cs.aau.dk, kgl@cs.aau.dk, and ask@cs.aau.dk

doi:10.1016/j.entcs.2010.12.014

would require huge models and would not be practically possible. Some abstraction is needed in order to make MBT applicable to testing database systems.

This paper presents one such abstraction. We model the database as two sets, the present set and the absent set. The present set is an under-approximation of the data present in the database, and the absent set is an under-approximation of the data not present in the database. This way we can abstract over an infinite amount of databases with two small sets.

To enable model-based testing using this abstraction we present two new forms of FSM: DBFSM and PAFSM. We show that the present-absent abstraction is used to translate from DBFSM to PAFSM, and how specifications for testing can be developed using PAFSM. Additionally we show an example and how test cases can be generated from this example.

In this paper we consider reactive systems which interact with databases in a shallow manner, meaning no complex operations on the data are performed. Rather the system can insert or remove values to and from the database and the control flow of the systems can depend on the presence or absence of values. We refer to this simplistic view as databases althought databases are far more complex. The simplistic view in this paper is a starting point and is intended to be extended with a more complex view of databases.

The paper is structured as follows: Section 2 describes some related work. Section 3 describes model-based testing in its two forms, online and offline. Section 4 through 7 describe the theoretical parts of this paper. First the present-absent abstraction is explained and proved. Then extended finite-state machines are explained, and these are further extended to include databases and present absent sets. The abstraction and translation between DBFSM and PAFSM is described in Section 8. A short example is presented in Section 10, and Section 11 concludes the paper.

2 Related Work

Ran et al. [5,4] have proposed a similar approach, in a system they call AutoDBT. They model web-based systems using FSMs, and model the databases as two sets, the actual database, and a synthesized database. The synthesized database contains values not in the actual database, but available for testing. The synthesized database is used when the test is required to input some value into the database. These two databases are similar to our present-absent sets. They differ however, in that we only model a small subset of the data in the actual database. Additionally the testing algorithm differs in that AutoDBT generates guards to be executed before every test case, to ensure that the database is in a conforming state, whereas we populate the modelled databases according to the actual database to ensure that the model is always in a conforming state. Ran et al. do not specify what happens if the system never enters a conforming state for a specific test case. Additionally AutoDBT only supports offline testing whereas our approach supports both online and offline testing.

3 Model-Based Testing

Model-based testing originates in the formal approaches developed by Tretmans [6,7], and implemented in the tool TorX [8]. These approaches have been extended to include real-time by Hessel et al. [1], and implemented in Uppaal TRON [2]. Also, a number of commercial UML-based tools are emerging, such as Qtronic and ATG.

Even though the aspects of this paper do not concern with real-time directly, it is intended to be used to extend Uppaal TRON to allow testing of data intensive systems. Uppaal TRON assumes timed automata as specification and supports conformance testing of real-time systems. Since the abstraction presented in this paper differs somewhat depending on whether the purpose is online or offline testing, a short description of these two types of testing is presented.

3.1 Online Testing

Online testing merges test-case generation and execution into one activity. The test cases are dynamically derived from a simulation of the model and sent to the implementation under test (IUT) directly. Output from the IUT is observed and the state of the model is updated accordingly. The advantages of online testing include easier handling of non-determinism and the reduction in state-space. Non-determinism is easier to handle since the IUT is dynamically observed, thereby revealing which non-deterministic choices have been taken, eliminating the need for the test tool to track unnecessary states. The state-space is reduced for the same reason. Disadvantages include the difficulty to reason about coverage and the arbitrarily long traces complicating the process of linking an erroneous test case to an error in the IUT.

3.2 Offline Testing

Offline testing involves generating a batch of test cases prior to executing them on the IUT. Test cases are generated by model-checking for a specific purpose and storing the trace from the model-checker. This trace serves as a test case to be executed to test the purpose. The advantages of offline testing include the ability to specify and reason about coverage in a very precise manner. Disadvantages include problems with handling non-determinism and the requirement of model-checking the model, requiring the entire state space to be explored, which can lead to state-space explosion. Handling non-determinism is a problem since the test case needs to take into account all possible outcomes of a test purpose. Consider a test case requiring a user to be present in the database. If the user is not present he needs to be created before the test can proceed. Some test-case execution tools do not support such non-determinism. QTP, an industrially used test-case execution tool, only supports static test cases of produced inputs and observed outputs. This problem of requiring static test cases is major when testing databases which inherently depend on an internal state and evolve dynamically during testing.

4 Present and Absent Sets

We now introduce the present and absent set abstraction originally proposed in [3]. The abstraction abstracts a database into two sets; the *present set* and the *absent set*. The present set is an under-approximation of the values which are present in the database and the absent set is an under-approximation of the values which are not in the database. This can be seen as a three-valued-logic, where if the value is in the present set it corresponds to *true*, if the value is in the absent set it corresponds to *false*, and if the value is in neither it corresponds to *unknown*. If the value is in both sets it corresponds to an erroneous state, this should be avoided. This abstraction allows us to abstract over an infinite number of databases and abstract away from the actual content of the database, using a relative small set of values.

We define the following sets [3]:

$\mathbb{D}$	is a set of *elements* (e.g. records, relations, tuples etc.) The complete set of values that can be entered into the database.
$D_n \subset \mathbb{D}$	is the concrete *state* of a database. The database used by the real system and can contain huge amounts of data.
$\mathbb{C} \subset \mathbb{D}$	is a set of representative elements. These can be chosen intuitively or by some heuristic, e.g. a few from each table.
$P_n \subseteq \mathbb{C}$	is the *present set*, containing the elements known to be present in database D_n.
$A_n \subseteq \mathbb{C}$	is the *absent set*, containing the elements known to be absent from database D_n.
$d \in \mathbb{D}$	is an element in the actual system.
$c \in \mathbb{C}$	is an element in the abstract system.

Figure 1 illustrates these sets. P_n can grow to fill the entire space $\mathbb{C} \cap D_n$ and A_n can grow to fill the entire space $\mathbb{C} \setminus D_n$. Some interesting observations follow from these sets:

$P_n \subseteq D_n$	every element in the present set must be in the database.
$A_n \cap D_n = \emptyset$	no element in the absent set can be in the database.
$P_n = A_n = \emptyset$	means no knowledge about the contents of database D_n.
$P_n \cup A_n = \mathbb{C}$	means everything is known about database D_n, given the current $\mathbb{C}$.
$P_n \cap A_n = \emptyset$	must always hold. The same element can never be present in and absent from the same database at the same time.

Three operations are allowed on the database: insert, remove, and query for presence. These operations are defined below on databases and present-absent sets. Queries are split into positive and negative since these are handled differently.

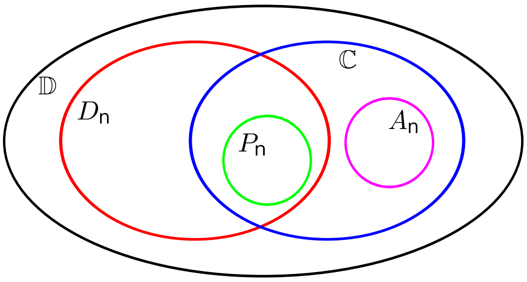

Fig. 1. Sets of the present-absent abstraction

Insert

Inserting into the database results in a new state with the inserted element added:

$$D'_n = D_n \cup \{c\}$$

This corresponds to adding the element to the present set and removing it from the absent set:

$$P'_n = P_n \cup \{c\}$$
$$A'_n = A_n \setminus \{c\}$$

Remove

Removing from the database results in a new state without the removed element:

$$D'_n = D_n \setminus \{c\}$$

In the present-absent sets this results in removing from the present set and adding to the absent set:

$$P'_n = P_n \setminus \{c\}$$
$$A'_n = A_n \cup \{c\}$$

Positive Query

This means that the element is in the database:

$$c \in D_n$$

If a query is positive we know the element is in the database, this means that we can add the element to the present set. We need to assert that the element is not in the absent, this is a consistency check.

$$P'_n = P_n \cup \{c\}$$
$$assert\ c \notin A_n$$

Negative Query

This means that the element is not in the database:

$$c \notin D_n$$

We can update the present-absent sets similarly to positive query:

$$A'_n = A_n \cup \{c\}$$
$$assert\ c \notin P_n$$

Theorem 4.1 *Operations on P_n and A_n are consistent and sound with respect to an actual D_n in the following sense:*

(i) *The P_n and A_n captured info does not contradict.*
  (a) $\forall c \in D_n \Rightarrow c \notin A_n$
  (b) $\forall c \notin D_n \Rightarrow c \notin P_n$

(ii) *P_n and A_n capture part of the D_n state.*
  (a) $\forall c \in P_n \Rightarrow c \in D_n$
  (b) $\forall c \in A_n \Rightarrow c \notin D_n$

Proof. We construct a proof by induction. We show that the properties hold for $P_n \cup A_n = \emptyset$ and show for each action that the properties hold after applying it on arbitrary sets.

(i) We assume no knowledge about the database; $P_n \cup A_n = \emptyset$. Theorem 1.i.a holds since the right side of the arrow is always true since A_n is empty, similarly for 1.i.b. Theorem 1.ii.a holds since P_n is empty so the right side of the arrow never needs to be evaluated, similarly for 1.ii.b.

(ii) We assume arbitrary sets P_n and A_n adhering to the requirements above. For each operation we now show that the properties hold after applying the operation.
  (a) *Insert:* After inserting c, 1.i.a holds since c is removed from A_n. 1.i.b holds since D_n contains c. 1.ii.a holds since both P_n and D_n contain c. 1.ii.b holds since A_n does not contain c.
  (b) *Remove:* After removing c, 1.i.a holds since D_n does not contain c. 1.i.b holds since c is removed from P_n. 1.ii.a holds since P_n does not contain c. 1.ii.b holds since D_n does not contain c.
  (c) *Positive Query:* After a positive query for c, 1.i.a holds since we assert that A_n does not contain c. 1.i.b holds since D_n contains c. 1.ii.a holds since we add c to P_n. 1.ii.b holds since A_n does not contain c.
  (d) *Negative Query:* After a negative query for c 1.i.a holds since D_n does not contain c. 1.i.b holds since we assert that P_n does not contain c. 1.ii.a holds since P_n does not contain c. 1.ii.b holds since D_n does not contain c.

□

Theorem 4.2 *For any operation performed on D_n, P_n, and A_n, yielding D'_n, P'_n,* *and A'_n, the captured info in P'_n and A'_n is more precise, i.e.*

$$P_n \cup A_n \subseteq P'_n \cup A'_n.$$

Proof. The proof is easy to see. When ever we remove an element from either P_n or A_n (during insert and remove operations) we always add the same element to the other set. This means the union of the sets can never shrink. During query operations we add an element to one of the sets and don't remove anything, meaning the union grows. □

Corollary 4.3 *Once P_n and A_n capture the entire knowledge of D_n, i.e. $P_n \cup A_n = \mathbb{C}$, performing operations will always keep the property*

$$P_n \cup A_n = \mathbb{C}$$

Since the knowledge can never shrink, once we have reached maximum knowledge we will stay at maximum knowledge.

5 Extended Finite-State Machines

Before introducing the novel FSMs we present EFSMs [9] on which DBFSM and PAFSM are based. An EFSM is an FSM extended with internal variables.

Definition 5.1 An *extended finite-state machine* is a 7-tuple $(Q, q_0, \Sigma, \Gamma, V, \psi, \delta)$, where:

- Q is a finite, non-empty *set of states.*
- $q_0 \in Q$ is the *initial state.*
- Σ is the *input alphabet*, a non-empty finite set of labels.
- Γ is the *output alphabet*, a non-empty finite set of labels.
- V is a finite *set of variable names.*
- $\psi \subset V \times Int$ assigns integer values to the variables.
- δ is a *state transition relation.*

δ relates a source state q, and an action a, to a target state q', given the current state of variables, ψ. This is written: $q \xrightarrow{a} q'$, and corresponds to a *transition* in the system. There are five types of actions:

- inputs, $\sigma \in \Sigma$
- outputs, $\gamma \in \Gamma$
- the null action, τ
- boolean conditions
- variable updates

A boolean condition action is only enabled if the condition evaluates to *true*. Boolean conditions and variable updates may use regular arithmetic and relational operators.

6 Database FSM

A *database finite-state machine* (DBFSM) is an EFSM where variables can have a type we call *database*. The database type has three operations: insert, remove, and query for membership, corresponding to the same operations on a real database. In the context of the present-absent abstraction, DBFSM should be seen as a system containing a real database.

Definition 6.1 A *database finite-state machine* is a 8-tuple $(Q, q_0, \Sigma, \Gamma, V, \psi, D, \delta)$, where:

- Q, q_0, Σ, Γ, V, ψ, and δ are defined as for EFSM.
- D is a *set of databases.*

A database can hold an infinite amount of values from variables. Three functions are defined for operating on databases: $Insert(d, v)$, $Remove(d, v)$, $Query(d, v)$, where $d \in D, v \in V$. $Insert(d, v)$ inserts the value of v into database d, $Remove(d, v)$ removes the value of v from database d, and $Query(d, v)$ returns a boolean, being *true* if the values of v is present in the database and $false$ otherwise.

This formalism gives us a convenient way to model systems using databases. However DBFSMs are infinite state systems and therefore not suited for modeling and testing.

7 Present-Absent FSM

We now introduce PAFSM which are an abstraction of DBFSM. The abstraction is done according to the present-absent abstraction.

Definition 7.1 A *present-absent finite-state machine* is a 9-tuple $(Q, q_0, \Sigma, \Gamma, V, \psi, DP, DA, \delta)$, where:

- Q, q_0, Σ, Γ, V, and δ are defined as for DBFSM.
- $\psi \subset V \times Values$ assigns integer values to the variables. $Values$ is a finite set of integer values.
- DP is a *set of sets*, each of size $|Values|$, representing the present sets. One for each database in the DBFSM.
- DA is a *set of sets*, each of size $|Values|$, representing the absent set. One for each database in the DBFSM.

$DP(d)$ represents the present set for database d, $DA(d)$ represents the absent set for database d.

This abstraction allows us to abstract over an infinite set of databases with a small set of sets. Additionally the PAFSM is finite-state, which enables straight forward state-space exploration. The requirement for integer values can easily be lifted to any value. Additionally the restriction is not a problem in practice, since the integer values can be translated into real database properties in the adapter prior to sending them to the IUT.

8 Translation

We now present the translation from DBFSM to PAFSM. Two translations are presented, they differ in the way unknown values are handled. The first translation assumes full knowledge of the database, and enters an error state if at any time an unknown value is observed. The second assumes no knowledge and is allowed to nondeterministically choose whether an unknown value should be treated as present or absent.

The DBFSM and PAFSM are the same in every aspect except transitions using one of the three operations on databases; insert, remove, and query. For the two translation it is explained who these are handled.

8.1 No Knowledge

This translation assumes no knowledge about the database, i.e. $P_n \cup A_n = \emptyset$. This is suited for online testing where the knowledge of the database can be derived during test execution. This translation can also be used to generate abstract traces, or trees, where branches in the tree correspond to choices in the model. This way offline test cases can be generated.

8.1.1 Insert

If the value is in the present set this transition has no effect. If the value is in the absent set, it is added to the present set and removed from the absent set. If the value is in neither present nor absent the value is added to the present set.

8.1.2 Remove

If the value is in the present set, it is removed from the present set and added to the absent set. If the value is in absent set this transition has no effect. If the value is in neither present nor absent the value is added to the absent set.

8.1.3 Query

If the value is in the present set, return true. If the value is in the absent set return false. If the value is in neither, non-deterministically choose true or false and add the value to the corresponding set.

This translation is conforming to the present-absent abstraction, in that each action updates the sets according to the abstraction. When using this translation for online testing, the non-deterministic choices allow the model-checker to be in both states at the same time, and reduce the state space when observations from the IUT reveal which choice was correct. When trees are generated for offline testing the tester can traverse the tree and follow branches according to the output observed. How the non-determinism is handled in practice is shown in more detail in the concrete example in Section 10.

8.2 *Full Knowledge*

This translation requires full knowledge about the database, i.e. $P_n \cup A_n = \mathbb{C}$. This translation is basically the same as above, except we remove the unknown aspect of the three-valued-logic. This translation is suited for offline testing, where complete and static traces need to be generated to simplify the test execution.

8.2.1 *Insert*

Since we have full knowledge about the database we know that the value is either in present or absent and never in both. Taking the transition adds the value to the present set and removes it from the absent set.

8.2.2 *Remove*

Taking the transition adds it to the absent set and removes it from the present set.

8.2.3 *Query*

If the value is in the present set we return true, otherwise return false. We do not need to consult the absent set since we have eliminated the unknown factor.

This translation also conforms to the present-absent abstraction. Since we start with maximum knowledge about the database we know that all values are in either the present or the absent set. From Corollary 1 we know that we never lose knowledge. This enables us to simplify the query operations. This translation is specifically well suited for offline testing where static traces are required.

There are two issues using this approach: It requires the state of the database to be known a priory and it requires the test-case generation to be re-executed prior to each execution of the test suite (not for each test case or test purpose, only for the entire test suite.) The state of the database only needs to be checked before executing the test suite the first time, since the state after executing the test suite can be stored and used as input for the next execution. The requirement to re-execute the test-case generation can be a major problem. The model checking and test-case generation might take a long time to complete, and requiring this for each test-suite execution might significantly increase the execution time required to execute the test.

To alleviate this problem it might be possible to generate a strategy to bring the databases into a specific known state. This way test cases can be generated with this state as starting point, and at the end of test execution the database is returned to the desired state.

9 Advantages

There are several advantages using the present-absent abstraction over using databases. Initially the state-space is reduced considerably compared to modeling the entire database.

Traditional testing of databases require the database to be in a specific state when beginning the test, and require the tests to be executed in a specific order, to ensure the database is always in a known state. Using the present-absent sets and MBT, we can enter a subset of the state of the database into the present and absent sets, then rerun the test case generation, based on the current state of the database. This way we can abstract away from the initial state of the database, and still get automatic testing.

Traditionally testing is not performed on the system in actual use, since the test cases can interact arbitrarily with the actual database. By proving correctness on the present-absent sets, and proving that the test cases will only interact with the specific test data, the tests can be executed on the actual running system. By observing the state of the database the state of the system can be entered into the sets, and the tests can be executed, only affecting the test data in the database.

During online testing it is possible to start the testing process without any knowledge about the database, i.e. $P_n \cup A_n = \emptyset$. This way the state of the database can be dynamically learned by observing the system. As more knowledge is gained, the state space is reduced, and the testing can be guided in the desired direction.

10 Example

To illustrate the abstraction and test-case generation, an example is presented. This example is manufactured by hand since no tool support has been developed yet. The specification of the IUT is a network of timed automata in Uppaal syntax. Three network of timed automata are presented: one modeling the system using databases, one translation assuming no knowledge and one assuming full knowledge.

The example is a simple system where users can login and perform some work. In the system we have a single database, consisting of the users which are currently logged in. The users have three actions: *login*, *logout*, and *work*. The user can only login if he has not already logged in. He can only logout if he is logged in. The work can only be requested if he is logged in. When the user performs an action the system will return either *OK* or *Error*.

Figure 2 illustrates the system using databases. This is used as the specification of the IUT. The specification is a timed automaton implemented in Uppaal. The `Q` method queries the login database, called `dbLogin`, and returns true if `cid` is in the database.

The system has three input channels: `work?`, `login?`, and `logout?`, and two output channels: `OK!` and `Error!`. The input channels are use by the user to query

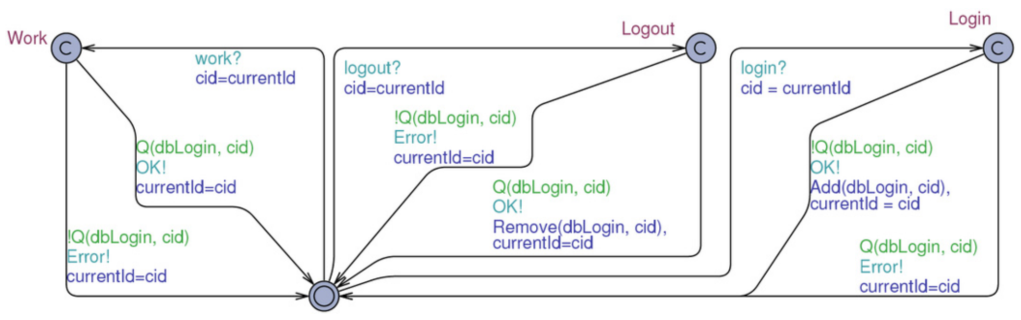

Fig. 2. Example using databases

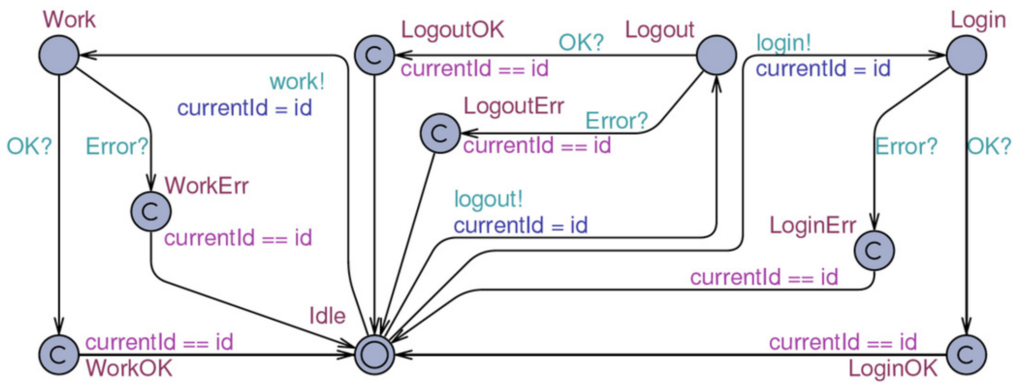

Fig. 3. The user

the system, the output channels are used to return to the user. The shared variable `currentId` is used to pass the id of the calling user to the system, and used by the user to ensure the result is returned to the correct user. The `Add` and `Remove` methods are used to add and remove `cid` to and from the database respectively.

Figure 3 illustrates the user of the system. This is an unintelligent user which presses all buttons in random order. Another type of user is one which follows the specification of the system. In this example such a user would for instance only try to request work if he knew he was logged in. The locations `WorkOK` and `WorkErr` are dummy locations used to specify test purposes (similar for login and logout). For instance to test whether a user is able to request work and get a positive response, Uppaal is asked for a trace where the user template enters the `WorkOK` location. The same user is used in all the examples.

We now explain how the model is translated using the two approaches explained above. We also explain how these model can be used for testing.

10.1 No Knowledge

Figure 4 illustrates the system where no knowledge is assumed. Each query to the database is translated into a call to the method `IsLogin`, which has three possible outcomes: `TRUE`, `FALSE`, and `UNKNOWN`, corresponding to the three valued logic in the abstraction. `IsLogin` consults the present and absent sets and returns based on the values. If the value is unknown a non-deterministic choice is available, either return `OK` and add the user to the present set, or return `Error` and remove the user from the set. The methods `Login` and `Logout` handle this. This has the effect of updating the database when the correct choice is observed from the IUT. Notice that when the result of a login action is unknown both the OK and the Error choice add the user. This is because, returning Error means the user is in the database, therefore we add him. If we return OK the user is not in the database, and we should remove him, however, since the login was successful the user is now added to the database, therefore we add him.

It can be seen that we do not make any consistency check on the present-absent sets, i.e. check for $P_n \cap A_n \neq \emptyset$. This is because we can verify that this can never be the case using the model checker with the following query:

```
A[] forall (id:UserID) !(Present(id) && Absent(id))
```

This query states: It is always the case that no user is in present and absent at the same time. If this query verifies there can occur no inconsistencies in the model.

This system starts with both sets empty. Whenever a choice is taken the sets are updated accordingly. If an online test is executed with this system as the specification, Uppaal would take both choices and keep track of two states in the system. When the actual action is observed from the IUT all nonconforming states are discarded. By running this system in the simulator in Uppaal we can simulate an online test where Uppaal makes the choice for the IUT. It can be seen that after executing for a while we reach a situation where we have full knowledge about the database, i.e. $P_n \cup A_n = \mathbb{C}$.

This system has been tested against an implementation using Uppaal TRON. The systems was instantiated with ten users. A mutant is made, in which the login action has a 1/500 chance to fail to update the database. The system has been implemented such that the database is filled with random values at initialization. This way the tester has no way of knowing the state of the database when starting the test. The test was run ten times on the correct implementation and ten times on the mutant. Each successful test executed about 22.000 action (input and output combined). One of the mutant runs failed to detect the mutant, this is due to the randomness of the mutant. The tests have shown us that the present-absent set approach has the capabilities to automatically test a system which interacts with a database without knowledge about the state of this database prior to testing.

We can also use this translation to generate abstract traces. A tree will be generated for each test purpose. A branch in the tree corresponds to a choice in the model. Uppaal CoVer can be used to generate these trees.

10.2 Full Knowledge

Figure 5 illustrates the system with full knowledge assumed. This system is similar to the system with no knowledge. Since we have full knowledge we can remove all transitions where the database state is unknown. This simplifies the model. This model is useful for generating static traces to be used for offline testing, by a

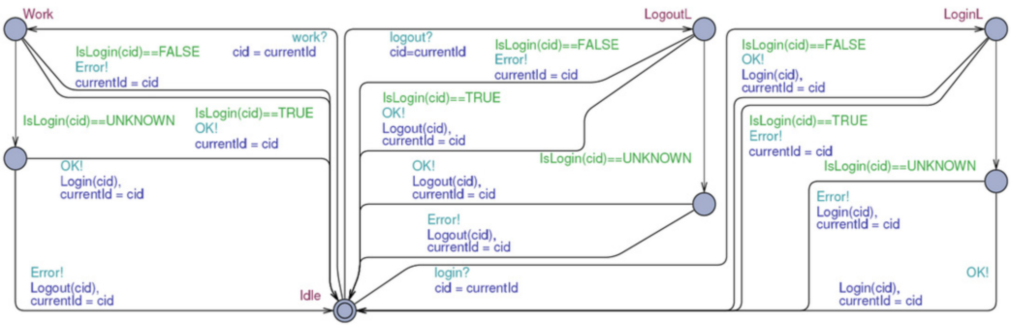

Fig. 4. Example with no knowledge

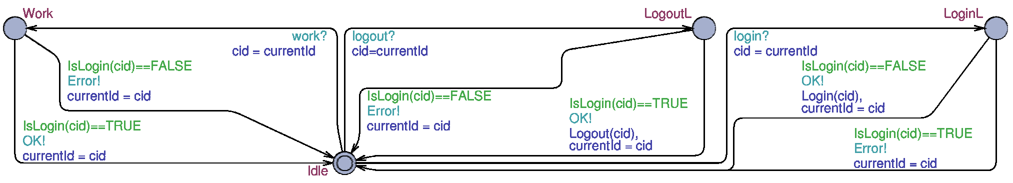

Fig. 5. Example with full knowledge

static testing tool. To generate the traces the UPPAAL model checker can be asked whether a template can reach a specific location and get a trace of how to reach this location. This trace can be used as a test case.

We use the dummy locations in the user template, Figure 3. The following query is used to test if the user with ID 0 can successfully login:

```
E<> Users(0).LoginOK
```

Verifying this with all users absent generates the following trace:

```
Users.Login[0]!
IUT.LoginOK[0]!
```

Meaning: First user with ID 0 sends a login request to the IUT, then the IUT sends `loginOK` to user 0. To show that we can generate different traces depending on the state of the database, we run the verifier again with all users present in the database. This generates the following trace:

```
Users.Logout[0]!
IUT.LogoutOK[0]!
Users.Login[0]!
IUT.LoginOK[0]!
```

Here we can see that the user first has to log out, before he can log in successfully. This shows that by updating the state of the database we can generate static traces which conform to the state of the database.

This simplistic example serves to illustrate the abstraction, but it is not representative of a realistic database system. Extending the user table in the database with properties can easily be achieved by creating present and absent sets for each property. This approach can also be extended to include relations between tables. A one-to-many relation can be modeled using present and absent sets for each entry in the *one* relation, these sets can hold the values which the corresponding entry relates to. How this preforms in practice needs to be analyzed in future work.

11 Conclusion

We have introduced the abstraction from a database into present and absent sets and a proof of this abstraction. We have introduced two new forms of FSMs, the DBFSM and PAFSM and explained how to translate from DBFSM to PAFSM. Furthermore, we have explained two different translations and how these relate to the present-absent abstraction.

We have illustrated an example of a simple system using a database, and how this system can be translated into a system using present and absent sets. We have explained how test cases can be generated from this system, as well as the benefits of using our approach when performing online and offline testing.

We are able to perform online testing of systems without taking any assumptions about the state of the database into account. As the test progresses, we gradually gain more knowledge about the state of the database. This increase of knowledge will reduce the state space of the simulation model, as well as enable us to potentially guide the testing in a desired direction.

We enable two forms of offline testing. One without assuming any knowledge about the state of the database. We are able to generate abstract traces which automatically learn the state of the database and make choices accordingly to reach the desired state. By examining the state of the database prior to generating the test cases, we are able to generate static traces which can be executed without any branching. This removes the problem of state dependency when performing offline testing on database systems. There are some potential performance issues with this approach, but we are hopeful as to finding a solution to these problems.

As future work we plan to extend the simplistic view of database presented in this paper. We plan to measure the effectiveness of this approach on larger examples, preferably industrial. We are currently working on extending the Uppaal model checker to improve the effectiveness of model checking systems using present and absent sets.

References

[1] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using uppaal. In *Formal Methods and Testing*, pages 77–117, 2008.

[2] K.G. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using uppaal. In *Formal Approaches to Testing of Software*, Linz, Austria, September 21 2004. Lecture Notes in Computer Science.

[3] Petur Olsen, Kim G. Larsen, Marius Mikucionis, and Arne Skou. Present and absent sets: Abstraction for data intensive systems suited for testing. In R. Huuck, G. Klein, and B. Schlich, editors, *Doctoral Symposium on Systems Software Verification (DS SSV'09)*, volume AIB-2009-14 of *Aachener Informatik Berichte*, pages 26–28, Aachen, Germany, 2009. RWTH Aachen University.

[4] Lihua Ran, Curtis Dyreson, and Anneliese Andrews. Autodbt: A framework for automatic testing of web database applications. *Lecture Notes in Computer Science*, 3306/2004:181–192, 2004.

[5] Lihua Ran, Curtis Dyreson, Anneliese Andrews, Renée Bryce, and Christopher Mallery. Building test cases and oracles to automate the testing of web database applications. *Inf. Softw. Technol.*, 51(2):460–477, 2009.

[6] Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR*, pages 46–65, 1999.

[7] Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, pages 1–38, 2008.

[8] Jan Tretmans and Ed Brinksma. Torx: Automated model based testing. In *Proceedings of the First European Conference on Model-Driven Software Engineering*, page 12 pp., 2003.

[9] Gregor von Bochmann and Jan Gecsei. A unified method for the specification and verification of protocols. In *IFIP Congress*, pages 229–234, 1977.