# Present and Absent Sets: Abstraction for Data Intensive Systems Suited for Testing

Petur Olsen, Kim G. Larsen, Marius Mikučionis, and Arne Skou

Department of Computer Science
Aalborg University, Denmark
{petur,kgl,marius,ask}@cs.aau.dk

**Abstract.** This paper presents an abstraction of reactive systems interacting with relational databases. This abstraction is used as a modeling technique, enabling model-based testing of systems utilizing relational databases. The abstraction is developed to work on timed-automata models, in conjunction with Uppaal TRON. The abstraction substitutes each table in the database with two sets: *present* and *absent*. The present set provides an under-approximation of the set of relations which are present in the database, and the absent set is an under-approximation of the set of relations being absent from the database. It is expected that the approach makes data-intensive models more manageable in model checking and testing.

## 1 Introduction

Safety critical systems are becoming more and more complex. Often using databases to store huge amounts of data influencing the systems. It is therefore important to be able to model these databases in an efficient way. In this paper we propose a novel approach to modeling reactive systems interacting with databases with the purpose of doing model checking and model-based testing.

We consider systems interacting with a database in a shallow manner, meaning the system does not perform complex manipulation of the data. Rather, the system can only insert or remove relations from the database, and the control flow of the system can depend on the presence or absence of relations.

We propose an abstraction of the database using two sets: *present* and *absent*. These sets provide under-approximations of present and absent data respectively. This way we can abstract away from the actual content of the database and only be concerned with a relatively small subset of data.

## 2 Model-Based Testing

Model-based testing has it's roots in the formal approaches developed by Tretmans [4,5], and implemented in the tool TorX [6]. These approaches have been extended to include real-time by Hessel et al. [2], and implemented in Uppaal TRON [3].

Although the method in this paper does not concern real-time systems directly, we assume the same black-box conformance testing setup, harvest the inspiration from our previous experience in this area and aim at extending this framework to handle real-time systems that are also data intensive.

Uppaal TRON is an online conformance testing tool for real-time systems which assumes timed automata model as a test specification. The specifications unavoidably tend to be non-deterministic in timing and behavior. Non-deterministic specifications lead to lack of control over IUT during testing, where IUT is free to choose more than one option of how to respond and proceed. This poses a challenge to tester, which has to check the behavior

against multiple response options, foresee and choose from multiple options for input actions. The problem of multiple choices is even more extreme in real-time cases where the number of timing choices is uncountably large.

The solution to non-deterministic requirements is adaptive testing, where the test is shaped like a tree with condition checks in nodes and input stimuli actions on edges. Such trees can be stored and executed offline or executed at the same time while being derived online (focusing only on relevant part of specification at a time.) In both online and offline model-based testing cases, the model state space is being explored and tests are generated based on current estimate of possible states that IUT can be in.

UPPAAL TRON uses difference bound matrix (DBM [1]) structures and UPPAAL algorithms to encode and operate on timed automata state estimates. In a similar spirit, we propose simple bit-vectors to abstract and encode the state estimate of relational database-like data in UPPAAL timed automata models.

## 3 Abstracting Data Intensive Systems

We define the following sets:

$\mathbb{D}$ 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.
$\mathbb{C} \subseteq \mathbb{D}$ 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 know 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.

Given these sets some interesting observations follow:

$P_n = \emptyset \wedge A_n = \emptyset$ 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.

The system can perform three operations: *Insert*, *remove*, and *query* for presence. Below we explain how operations are performed.

| | | | | | |
|---|---|---|---|---|---|
| Insert: | $D_n = D_n \cup \{c\}$ | $\Rightarrow$ | $P_n = P_n \cup \{c\}$ | $\wedge$ | $A_n = A_n \setminus \{c\}$ |
| Remove: | $D_n = D_n \setminus \{c\}$ | $\Rightarrow$ | $P_n = P_n \setminus \{c\}$ | $\wedge$ | $A_n = A_n \cup \{c\}$ |
| Query with positive outcome: | $c \in D_n$ | $\Rightarrow$ | $P_n = P_n \cup \{c\}$ | $\wedge$ | *assert* $c \notin A_n$ |
| Query with negative outcome: | $c \notin D_n$ | $\Rightarrow$ | $A_n = A_n \cup \{c\}$ | $\wedge$ | *assert* $c \notin P_n$ |

**Theorem 1** *Operations on $P_n$ and $A_n$ are consistent and sound with respect to an actual $D_n$ in the following sense:*

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

*(b)* $\forall c \notin D_n \Rightarrow c \notin P_n$

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

Sketch of a proof by induction: 1) take $P_n = A_n = \emptyset$ as the initial state – the properties hold trivially, 2) start with arbitrary sets $P_n \cap A_n = \emptyset$, 3) apply the operations with arbitrary element and show that the properties are preserved.

*Example:* Figure 1 illustrates a simple example of the usage of present/absent sets. The user performs a `login` action. The IUT has to respond with `OK` or `Error` depending on user status. The environment has cases for all combinations of present/absent. When the IUT sends an `OK` or `Error` the environment checks it's guards and performs updates. The `Add` and `Remove` methods in the model are used in case no prior knowledge is available. These will be used in a similar way if the system has to insert values into the database. A trace from the example can be used to populate the database to ensure a certain part of the model is reached, e.g. by setting a user to absent and not present will ensure the left-most edge will be taken.

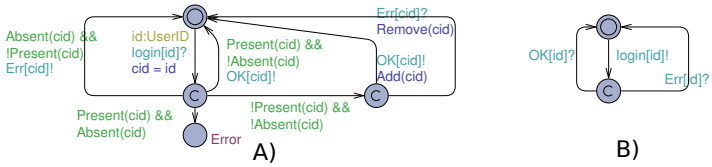


**Fig. 1.** Example. IUT is the database with present/absent lists (A), and the tester plays the role of environment/user (B).

## 4 Expected Results

It is expected that using this approach the data-intensive timed-automata models become more manageable during model checking as well as testing for several reasons. 1) The state-space of the model can be significantly reduced using this abstraction. This is accomplished by reducing the, potentially infinitely large, database into a set of relatively small lists. 2) in offline test generation, the proposed sets directly reveal what data items have to be present/absent in the database in order to fulfill the purpose of the test. 3) During online test, the state estimate size does not grow due to data uncertainty anymore and the precision is increasing as the test progresses.

## References


1. Alexandre David. Uppaal dbm library. http://www.cs.auc.dk/∼adavid/UDBM/, December 2006.
2. 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.
3. 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.
4. Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR*, pages 46–65, 1999.
5. Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, pages 1–38, 2008.
6. 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.