Time dependent policy-based access control

Vasilikos, Panagiotis; Nielson, Flemming; Nielson, Hanne Riis

Published in:
Leibniz International Proceedings in Informatics

Link to article, DOI:
10.4230/LIPIcs.TIME.2017.21

Publication date:
2017

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Time Dependent Policy-Based Access Control

Panagiotis Vasilikos\textsuperscript{1}, Flemming Nielson\textsuperscript{2}, and Hanne Riis Nielson\textsuperscript{3}

\textsuperscript{1}Department of Applied Mathematics and Computer Science, Technical University of Denmark, Lyngby, Denmark
panva@dtu.dk
\textsuperscript{2}Department of Applied Mathematics and Computer Science, Technical University of Denmark, Lyngby, Denmark
fnie@dtu.dk
\textsuperscript{3}Department of Applied Mathematics and Computer Science, Technical University of Denmark, Lyngby, Denmark
hrni@dtu.dk

Abstract
Access control policies are essential to determine who is allowed to access data in a system without compromising the data’s security. However, applications inside a distributed environment may require those policies to be dependent on the actual content of the data, the flow of information, while also on other attributes of the environment such as the time.

In this paper, we use systems of Timed Automata to model distributed systems and we present a logic in which one can express time-dependent policies for access control. We show how a fragment of our logic can be reduced to a logic that current model checkers for Timed Automata such as UPPAAL can handle and we present a translator that performs this reduction. We then use our translator and UPPAAL to enforce time-dependent policy-based access control on an example application from the aerospace industry.

1998 ACM Subject Classification D.4.6 Security and Protection, D.2.4 Software/Program Verification, C.2.4 Distributed Systems, F.4.1 Mathematical Logic

Keywords and phrases Access Control, Timed Automata, Time-Dependent Policies, UPPAAL

Digital Object Identifier 10.4230/LIPIcs.TIME.2017.21

1 Introduction

Motivation. Cyberphysical systems play an increasingly important role in the technology development in many industries such as the aerospace, the automotive and the medical. Embedded systems are key components to cyberphysical systems and while verifying their safety goals has received a significant focus until now, security has been left for later. As more and more cyberphysical systems are integrated with real-time hardware, complex software, and internet connected devices through wireless connections, ensuring security goals of those systems is becoming essential. Particularly, assuring the confidentiality or the integrity of the information manipulated by the different components of a cyberphysical system is a crucial security goal.

Information security is usually achieved by access control policies, which formally specify desired flows of information inside a system. Access requests to the resources/data (objects)
of the system by users (subjects) are then either denied or allowed by a monitor that enforces the access control policies. The literature offers a vast number of access control models, where among all, the most used in practice are the discretionary access control (DAC) [29], mandatory access control (MAC) [28] and role-based access control (RBAC) [12], while lately a great attention has been given to the attribute-based access control (ABAC) model [17], wherein access control may depend on the attributes of the accessed data or the attributes of the environment such as the time.

Although access control policies is a well-established approach for information security at the subject-object level, distributed systems require precise policies that express also the desired information flows that occur at the application level, such as explicit flows. As an example, consider the explicit flow from the variable $y$ to the variable $x$ that arises from the assignment $x := y$ and the access control policy "$x$ can only be modified by $p$", where $p$ is a trusted process in the system. Although the assignment $x := y$ executed by $p$ does not violate the access control policy of the system, the fact that $p$ resides in a distributed environment together with potential bugs inside its source code gives no guarantees that the value of $y$ was written by $p$. For instance, the value of $y$ could have been influenced by an untrusted process $p'$ after a communication between $p$ and $p'$ and consequently $p'$ would have also influenced the value of $x$. To see more challenges that arise inside distributed systems consider the work of [25] where it illustrates that security policies may need to depend on the actual content of the data, while ABAC [17] models also address the need for time-dependent security policies.

**Contribution.** It is natural then to extend the enforcement of safety properties of embedded systems with enforcement of access control policies. The idea is that having an abstract model of an embedded system, one could eliminate possible security violations in the trusted part of the system before the actual run of the system happens. We use Timed Automata [3, 1] to model distributed systems and we specify an information flow instrumented semantics that allows us to record information about the accesses being performed; we call this information a behaviour of the system. To deal with formal definitions of security policies we present a behaviour logic (based on the behaviours of the system) that supports the specification of content, time and information dependent access control policies. Verification of Timed Automata has been successfully achieved by model checkers based on the timed computation tree logic (TCTL) [2], and consequently, we propose a reduction of a substantial fragment of our logic to a logic that can be handled by the well-established model checker UPPAAL [30]. Finally, we present a translator that performs this reduction and we illustrate our development using an example from the aerospace industry. Figure 1 sketches the example: A gateway with two processes, each of them produces data for different targets, uses a multiplexer and a demultiplexer to successfully deliver the data to the intended target. The multiplexer merges the data from the producers and sends it to the demultiplexer who is responsible for delivering it to the right target. The target of the data depends both on the time of the system while also on the content of it and thus it is challenging to express the appropriate security policy. The example is based on the secure gateway presented in [22], where a separation
kernel is used to allow the accesses to the resources of the system by the system’s processes to be temporal (based on time) and based on an information flow policy. The separation of the resources is used to ensure that untrusted processes such as passenger’s devices can have access to onboard communication systems, without alerting the safe operation of the aircraft.

**Related work.** There are many other papers dealing with access control [27, 19, 15, 14, 34, 32], however without considering time-dependent security policies; a survey of access control models is available at [11].

A rich logic that allows reasoning about time-dependent policies, together with a proof checker for the logic is considered in [10], however there are no information flow considerations at the application level such as explicit flows. SecPAL [5] is another logic that supports time-dependent policies, as well as the encoding of many well-known policy idioms such as DAC, MAC, RBAC, and ABAC, however the enforcement of time constraints is external to the language. A somewhat different approach has been taken in [4], where a monitor is used to enforce time-dependent access control, by checking a system’s logs that records the different actions of the users in a database system. Our contribution focuses on the challenges of time-dependent access control for embedded systems modeled as Timed Automata.

The work of [20] presents a formal specification and verification of the temporal role-based access control (TRBAC) model [6], a flexible model in which the roles of the system are enabled or disabled depending on the time of the system. They then use UPPAAL [30] to model a TRBAC system and verify the desired security policies. The same authors of this paper, present in [21], an extension of this model which is based on the generalized-TRBAC (GTRBAC) model [18]. The work of [13] considers spatial-TRBAC (STRBAC) models [7] in which the rights of the user may depend on the time as well as on the location of the user; again the different roles of the system are modeled as timed automata and verified in UPPAAL. Carlo Combi et.al in [9] merges temporal role-based access control with workflows, while in [8] he defines access-controlled temporal networks, an extension of the conditional simple temporal networks with uncertainty which allows you to model users and temporal authorization constraints. Although all of those models deal with an important number of access control policies at the subject-object level considering time dependencies, they are not able to express time-dependent policies with information flow considerations that occur at the application level of the system (e.g does a process running on behalf of a user respects the access control policy?).

The work of [26] formalizes the timed decentralised label model (TDLM) an extension of the traditional and well-established decentralised label model (DLM) [23], which deals with both information flow and time-dependent security policies; however, their work does not consider an enforcement mechanism for the policies. Our key contribution is to develop a logic that allows the specification of time, data’s content and information flow dependent policies for access control, and to make use of current model checkers such as UPPAAL [30] for the enforcement of the policies.

**Organisation.** The remainder of this paper is organized as follows. In Section 2, we give the definition of a Timed System (a system of Timed Automata) and in Section 3 we define an information flow instrumented operational semantics for Timed Systems. Section 4 presents the syntax and the semantics of our behaviour logic called BTCTL (behaviour TCTL) and we illustrate how we can successfully express security policies for our gateway example. In Section 5 we present the reduction of the BTCTL logic to a variation of the TCTL [2], called TCTL+ and in Section 6 we present our translator. Finally, in Section 7, we give our
conclusions and we outline our future work, while in appendix A we give the proof of our main theorem.

2 Systems of Timed Automata

2.1 Timed Systems

A Timed System $TS$

$$p_1 : TA_1 || ... || p_n : TA_n \ (n \geq 1)$$

which sometimes we will call system, is the parallel composition of $n$ timed automata called processes, written as $TS := (TA_i)_{i \leq n}$.

The processes are able to exchange information via synchronous message passing, using polyadic channels from the finite set $\text{Chan}$. Each of the processes in the timed system, is labelled with a unique identifier $p \in P = \{p_1, ..., p_n\}$ and we write $\text{Var}_p$ and $\text{Clock}_p$ for the data variables and clocks appearing in the process $p$. We also require that the sets of data variables and clocks for the processes are mutually disjoint ($\forall i \neq j : \text{Var}_{p_i} \cap \text{Var}_{p_j} = \emptyset \land \text{Clock}_{p_i} \cap \text{Clock}_{p_j} = \emptyset$) and we write $\text{Var} = \bigcup_i \text{Var}_{p_i}$ and $\text{Clock} = \bigcup_i \text{Clock}_{p_i}$ for the overall data variables and clocks appearing in the timed system.

2.2 Timed Automata

Formally, we model a Timed Automaton [3, 1] $TA$ as a 4-tuple $(q_0, E, I, Q)$ where $q_0$ is the initial location of the automaton, $E$ is a finite set of edges, $I$ is mapping from the automaton’s locations to conditions that impose invariants, and $Q$ is the set of the automaton’s locations.

The edges are labelled with actions $g \rightarrow \text{act}$ that take the form $(q_s, g \rightarrow \text{act} : \vec{r}, q_t)$ where the syntax of the $\text{act}$ is given by

$$\text{act} ::= \vec{x} := \vec{e} | \text{ch}\vec{e} | \text{ch}\vec{r}\vec{x}$$

and $q_s \in Q$ is the source location and $q_t \in Q$ is the target location.

Every action $g \rightarrow \text{act}$ consists of a guard $g$ which has to be true in order for the action to be performed and it ends with a reset on the clocks $\vec{r}$. The assignment action $g \rightarrow \vec{x} := \vec{e}$ performs multiple assignments $\vec{x} := \vec{e}$, while the action $g \rightarrow \text{ch}\vec{e}$. $\vec{r}$ is used to communicate the data of the expressions in $\vec{e}$ using the channel $\text{ch}$ and the action $g \rightarrow \text{ch}\vec{r}\vec{x}$ is used to receive data and store it in the variables of the vector $\vec{x}$. We shall assume that the sequences $\vec{x}$ and $\vec{e}$ of data variables and expressions, respectively, have the same length and that $\vec{x}$ does not contain any repetitions. Finally, we write $\vec{x}(i)$ (and also $\vec{e}(i)$) for the $i$-th element of the vector $\vec{x}$ (and $\vec{e}$ respectively). To cater for special cases of the assignment action, we shall allow to write $g \rightarrow \text{skip} : \vec{r}$ when $\vec{x}$ (and hence $\vec{e}$) is empty; also for any kind of action we shall allow to omit the guard $g$ when it equals to $t t$ and to omit the clock resets when $\vec{r}$ is empty. If it is the case that all of the above take place together we omit the whole action.

Example 1. The timed system of our gateway example is given in Figure 2. The timed system consists of six processes $P = \{p_1, p_2, m, d, c_1, c_2\}$. Two producers $p_1$ and $p_2$ send their data via the channels $in_1$ and $in_2$ respectively. The multiplexer $m$ collects the data from the producers using the channel $\text{ch}$ and then forwards it to the demultiplexer $d$, who then distributes it to the consumers $c_1$ and $c_2$ via the channels $out_1$ and $out_2$ respectively. The access policy that we want to impose here is that the consumers $c_1$ and $c_2$ read data only from the producers $p_1$ and $p_2$, respectively.
We use clocks to model the temporal accesses to the resources of the system as required in [22]. In particular we use two clocks $v$ (in the multiplexer) and $r$ (in the demultiplexer) to model instantaneous transitions (time does not pass) and we use a clock $t$ (in the multiplexer) to split the overall execution time of the timed system into periods of 10-time units. In each period the multiplexer $m$ reads data from the channel $in_1$ only whenever $t \in [0, 7]$, while it reads data from the channel $in_2$ only whenever $t \in [5, 10]$. Whenever $t \in [5, 7]$ the multiplexer chooses non-deterministically to read either from $in_1$ or $in_2$. The multiplexer then transports the data together with a constant, using the dyadic channel $ch$ to the demultiplexer $d$; the constant is used as a mark to indicate the source of the data. Finally, the demultiplexer delivers the data to the right consumer according to the constant.

The transitions of the timed systems are labeled with behaviours. A behaviour records information relevant to the action that has occurred and also information about the processes.

The expressions $e$, guards $g$ and conditions $c$ that label the locations are defined as follows using boolean tests $b$:

\begin{align*}
  e & ::= e_1 \text{ op}_a e_2 | x | n \\
  b & ::= \text{ tt } | \text{ ff } | e_1 \text{ op}_r e_2 | \neg b | b_1 \land b_2 \\
  g & ::= b | r \text{ op}_c n | (r_1 - r_2) \text{ op}_c n | g_1 \land g_2 \\
  c & ::= b | r \text{ op}_d n | (r_1 - r_2) \text{ op}_d n | c_1 \land c_2
\end{align*}

The arithmetic operators $\text{ op}_a$ and the relational operators $\text{ op}_c$ are as usual. For comparisons of clocks we use the operators $\text{ op}_c \in \{<, \leq, =, \geq, >\}$ in guards and the less permissive set of operators $\text{ op}_d \in \{<, \leq, =\}$ in conditions.

### 3 Information Flow Instrumented Semantics

#### 3.1 Behaviours

The transitions of the timed systems are labeled with behaviours. A behaviour records information relevant to the action that has occurred and also information about the processes.
that were involved in the action. Formally a behavior takes the form
\[ b \in B_{\text{local}} \cup B_{\text{com}} \]
where
\[ B_{\text{local}} = P \times \overrightarrow{\text{Var}} \times \overrightarrow{\text{Exp}} \]
are the behaviours that occur due to assignments and
\[ B_{\text{com}} = P \times \text{Chan} \times \overrightarrow{\text{Var}} \times \overrightarrow{\text{Exp}} \times P \]
are the behaviours that occur due to the communication between two processes. We write \( \overrightarrow{\text{Var}} \) and \( \overrightarrow{\text{Exp}} \) for the sets of vectors with elements over the data variables and arithmetic expressions respectively.

For instance, the local behaviour \( p : (\vec{x}, \vec{e}) \) records that the process \( p \) has performed an assignment in which the vector \( \vec{e} \) is used to modify the variables of the vector \( \vec{x} \), while the behaviour \( p : ch(\vec{x}, \vec{e}) : p' \) records that a communication between the processes \( p \) (the sender) and \( p' \) (the receiver) has happened, using the channel \( ch \), and the vector \( \vec{e} \) is the vector of expressions whose values have been communicated and have been bound to the variables of the vector \( \vec{x} \).

In all of the behaviours, the vectors that are being used must have the same length while for the delay action we will write the empty behaviour \( \epsilon \).

### 3.2 Operational Semantics

To specify the semantics of timed systems, let \( \sigma \) be a state mapping data variables to values (which we take to be integers), let \( \delta \) be a clock assignment mapping clocks to non-negative reals and let \( \kappa \) be a mapping from data variables to sets of processes which we will call writers. The mapping \( \kappa \) is used to monitor the explicit flows that occur from the assignments and the communication between two processes in the system; we explain the use of \( \kappa \) in more detail in a while. We then have total semantic functions \([\cdot]\) for evaluating the expressions, boolean tests, guards, and conditions; we evaluate expressions either with a state \( \sigma \) or the mapping \( \kappa \), where for the first case the evaluation returns a value and in the second it returns the writers of the expression. The evaluation of boolean expressions only depends on the states, whereas that of guards and conditions also depend on the clock assignments.

The configurations of a timed system \( TS = (TA_i)_{i \leq n} \) are of the form \( \langle \vec{q}, \sigma, \delta, \kappa \rangle \), where \( \vec{q} \) is a vector of nodes and we write \( q(i) \) for the \( i \)-th element of the vector \( \vec{q} \), \( q'[q'/q] \) to substitute the node \( q \) with the node \( q' \) in \( \vec{q} \), we have that \( \forall i : q(i) \in Q_i \) and finally we shall assume that the sets of nodes of the processes are mutually disjoint (\( \forall i \neq j : Q_i \cap Q_j = \emptyset \)).

The transitions of a timed system take the form
\[
\langle \vec{q}_s, \sigma, \delta, \kappa \rangle \xrightarrow{b} \langle \vec{q}_t, \sigma', \delta', \kappa' \rangle
\]
and the initial configurations are of the form \( \langle \vec{q}_0, \sigma, \lambda r.0, \kappa_0 \rangle \) where \( \vec{q}_0 \) is the vector whose elements are the initial locations of the timed automata of the system and \( \kappa_0 \) maps each variable to the process that it belongs to (\( \kappa_0(x) = \{ p \} \) iff \( x \in \text{Var}_p \)). The transition relation is given in Table 1.

The rule for the assignment, ensures that the guard is satisfied in the starting configuration and updates the mappings \( \sigma, \delta, \kappa \) and the location of the process \( p_{ij} \) and finally ensures that the invariant is satisfied in the resulting configuration. The behaviour \( p_{ij} : (\vec{x}, \vec{e}) \) records
which corresponds to the communication between the producer and the consumer.

The access control policies can then be enforced statically before the execution of the system. The dependent security policies for access control, based on the behaviours of the system. In this section, we present our behaviour based logic that the process \( p_i \) is performing an assignment to the vector \( \vec{x} \) using the vector \( \vec{e} \), and \( \kappa' \) records the information flow that occurs due to this behaviour, by updating the writers of each variable \( \vec{x}(i) \) with the writers of the expression \( \vec{e}(i) \), where for a single expression \( e' \), \( [e']_k = \bigcup_{v \in f(v(e'))} \kappa(y) \) and \( f(v(e')) \) is the set of free variables occurring in \( e' \).

To understand the rule for the communication one could see it as an assignment of the form \( \vec{x} := \vec{e} \) where \( \vec{e} \) are the expressions which are used at the channel output action and \( \vec{x} \) the variables that are used in the channel input action.

Finally, the delay rule only modifies the clock assignment with a delay \( d \) ensuring that the invariant is satisfied in the resulting configuration. The mapping \( \kappa \) remains the same since the delay action produces the empty behaviour \( \epsilon \).

**Example 2.** To see how the semantics for the \( \kappa \) mapping works, return to Example 1 and consider the transition

\[
(q, \sigma, \delta, \kappa) \xrightarrow{p_i \cdot \text{in}(x,x_1): m} (q_1[5/3], \sigma[x \mapsto \sigma(x_1)], \delta[v \mapsto 0], \kappa[x \mapsto \{p_1\}])
\]

which corresponds to the communication between the producer \( p_1 \) and the multiplexer \( m \). We have that \( q = (1, 2, 5, 8, 3, 4) \), and let

\[
\kappa = \{x_1 \mapsto \{p_1\}, x_2 \mapsto \{p_2\}, x \mapsto \{m\}, y \mapsto \{m\}, z \mapsto \{d\}, z_1 \mapsto \{c_1\}, z_2 \mapsto \{c_2\}\}
\]

and the resulting mapping \( \kappa[x \mapsto \{p_1\}] \) records that \( p_1 \) has written its value into the variable \( x \), since there is an explicit flow from the variable \( x_1 \) to the variable \( x \) and \( x_1 \) has previously been written by \( p_1 \).

### 4 Time Dependent Policies in BTCTL

In this section, we present our behaviour based logic **BTCTL** which serves to specify time-dependent security policies for access control, based on the behaviours of the system. The access control policies can then be enforced statically before the execution of the system.
4.1 The Logic

The syntax of the BTCTL formulas $\phi$ is given by

$$\phi ::= g \mid \text{set}_1 \text{rel} \text{set}_2 \mid \forall b(\phi_1, \phi_2) \mid \phi_1 \land \phi_2 \mid \neg \phi$$

where

$$\text{set} ::= e \mid W.$$  

We have basic formulas which can be either a guard $g$, or relations between two sets of writers, $\text{set}_1 \text{rel} \text{set}_2$, where $\text{rel} = \{\subseteq, \not\subseteq\}$. The underlined set expression $\subseteq$ denotes the set of writers of the expression $e$ and $W \in \mathcal{P}(\text{P})$ is a set of writers. We use the box operator $\forall b(\phi_1, \phi_2)$ to speak about pre- and post-conditions whenever the non-empty behaviour $b \not= e$ happens. Informally speaking, a configuration $\gamma$ will satisfy the $\forall b(\phi_1, \phi_2)$ formula whenever for all of the system runs starting at $\gamma$, if a transition labelled with the behaviour $b$ occurs, then $\phi_1$ should hold at the configuration before the transition and $\phi_2$ at the configuration after it. As we will see shortly, the box operator will be the key formula to express access control policies. The $\neg \phi$ and $\phi_1 \land \phi_2$ cases are the usual ones. Finally, we sometimes write $\phi_1 \Rightarrow \phi_2$ for $\neg(\phi_1 \land \neg \phi_2)$.

> Example 3. Going back to Example 1, each of the variables has a time-dependent policy which specifies the maximum set of permitted writers of the variable. We are interested in the policies of the variables of the multiplexer, the demultiplexer and the two consumers:

$$P_x = (0 \leq t \land t < 5 \Rightarrow x \subseteq \{p_1\}) \land$$  
$$(5 \leq t \land t \leq 7 \Rightarrow x \subseteq \{p_1, p_2\}) \land$$  
$$(7 < t \land t \leq 10 \Rightarrow x \subseteq \{p_2\}),$$  

$$P_y = \not\subseteq \{m\},$$  

$$P_z = (0 \leq t \land t \leq 7 \land y = 1 \Rightarrow z \subseteq \{p_1\}) \land$$  
$$(5 \leq t \land t \leq 10 \land y = 2 \Rightarrow z \subseteq \{p_2\}),$$  

$$P_{z_1} = \not\subseteq \{p_1\},$$  

$$P_{z_2} = \not\subseteq \{p_2\}.$$  

The first line of the policy for the variable $x$, expresses that whenever $t \in [0, 5)$, only the process $p_1$ is allowed to write data to $x$, while both $p_1$ and $p_2$ may write to $x$ if $t \in [5, 7]$ and similarly to the first line, if $t \in (7, 10]$ then only $p_2$ can write to $x$. On the other hand, looking at the policy for the variable $y$, a write action to $y$ is allowed only by the multiplexer. The rest of the policies can be explained accordingly.

We then perform the enforcement of the access control policies by checking the following formulas:

$$\Phi_x = \forall_{p_1 \cdot \text{in}(x, x_1) \cdot m}(\text{tt}, P_x) \land \forall_{p_2 \cdot \text{in}(x, x_2) \cdot m}(\text{tt}, P_x),$$  
$$\Phi_{y,z} = \forall_{\text{m.ch}((y, z), (1, x)) \cdot d}(\text{tt}, P_y \land P_z) \land \forall_{\text{m.ch}((y, z), (2, x)) \cdot d}(\text{tt}, P_y \land P_z),$$  
$$\Phi_{z_1} = \forall_{\text{d.out}(z_1, z) < 1}(\text{tt}, P_{z_1}),$$  

$$\Phi_{z_2} = \forall_{\text{d.out}(z_2, z) < 2}(\text{tt}, P_{z_2}).$$

Each of the formulas express that whenever someone is writing to the variable (or variables) appearing as a subscript, then the policy of the variable (or variables) is imposed as a post
condition. The variable \( x \) is accessed (someone is writing data to \( x \)) whenever \( p_1 \) and \( p_2 \) communicates with the multiplexer and thus we have to impose the policy of the variable \( x \) for both of those actions, while the variables \( y \) and \( z \) are being accessed whenever the multiplexer communicates with the demultiplexer and that happens with two communication actions. Similarly, we define the formula for the variables \( z_1 \) and \( z_2 \).

### 4.2 Semantics of BTCTL

The formal rules that define whenever a configuration \( \gamma \) satisfies a BTCTL formula \( \phi \) are given below:

\[
\begin{align*}
gamma & \models g & \text{iff} & & g = (\bar{q}, \sigma, \delta, \kappa) \Rightarrow [g](\sigma, \delta) \\
gamma & \models \text{set}_1 \text{ rel } \text{set}_2 & \text{iff} & & \gamma = (\bar{q}, \sigma, \delta, \kappa) \Rightarrow [\text{set}_1] \kappa \text{ rel } [\text{set}_2] \kappa \\
gamma & \models \forall \square_b(\phi_1, \phi_2) & \text{iff} & & \forall \gamma_0 \exists \gamma \in \text{Trace} \gamma : \\
 & & & & \forall i \geq 1 : b_i = b \Rightarrow \gamma_{i-1} \models \phi_1 \text{ and } \gamma_i \models \phi_2 \\
gamma & \models \phi_1 \land \phi_2 & \text{iff} & & \gamma \models \phi_1 \text{ and } \gamma \models \phi_2 \\
gamma & \models \neg \phi & \text{iff} & & \gamma \not\models \phi
\end{align*}
\]

A guard \( g \) is then satisfied by a configuration \( \gamma \) whenever \( g \) holds in \( \gamma \). For the case of the set relation \( \text{rel} \), \( \gamma \) satisfies it whenever \( \text{set}_1 \text{ rel } \text{set}_2 \) evaluates to true and we do that check by lifting the definition of \( [.] \kappa \) to set expressions, by \( [e] \kappa = [e] \kappa \) and \( [W] \kappa = W \). A configuration \( \gamma \) satisfies the box formula \( \forall \square_b(\phi_1, \phi_2) \) whenever for all the execution paths that start from \( \gamma \), if a behaviour \( b' \) occurs and \( b' \) is syntactically equal to \( b \), then the pre-condition \( \phi_1 \) has to hold at the configuration before the behaviour and the post-condition \( \phi_2 \) after it. The rest of the cases are the usual ones.

► **Example 4.** Consider now the prefix of an execution trace of the timed system from Example 1

\[
\begin{align*}
pr = \gamma_0 & \xrightarrow{p_1 : \text{in}_1(x, x_1) : m} \gamma_1 & \xrightarrow{m \text{ ch}(y, z)(1, x)} \gamma_2 & \xrightarrow{d \text{ out}_1(z_1, z) : c} \gamma_3
\end{align*}
\]

where for the initial configuration \( \gamma_0 = (\bar{q}_0, \sigma_0, \delta_0, \kappa_0) \), we have that \( \bar{q}_0 = (1, 2, 5, 8, 3, 4) \), \( \sigma_0 \) is arbitrary, \( \delta_0 = \lambda c.0 \) and

\[
\kappa_0 = [x_1 \mapsto \{ p_1 \}, x_2 \mapsto \{ p_2 \}, x \mapsto \{ m \}, y \mapsto \{ d \}, z \mapsto \{ d \}, z_1 \mapsto \{ c_1 \}, z_2 \mapsto \{ c_2 \}]
\]

and for the rest of the configurations

\[
\begin{align*}
\gamma_1 & = (\bar{q}_1, \sigma_1, \delta_1, \kappa_1), \bar{q}_1 = \bar{q}_0[6/5], \sigma_1 = \sigma_0[x \mapsto \sigma_0(x_1)], \delta_1 = \delta_0[y \mapsto 0], \kappa_1 = \kappa_0[x \mapsto \{ p_1 \}] \\
\gamma_2 & = (\bar{q}_2, \sigma_2, \delta_2, \kappa_2), \bar{q}_2 = \bar{q}_1[5/8][9/8], \sigma_2 = \sigma_1[y \mapsto 1, z \mapsto \sigma_1(x)], \delta_2 = \delta_1[y \mapsto 0], \\
& \kappa_2 = \kappa_1[y \mapsto 0, z \mapsto \{ p_1 \}] \\
\gamma_3 & = (\bar{q}_3, \sigma_3, \delta_3, \kappa_3), \bar{q}_3 = \bar{q}_2[8/9], \sigma_3 = \sigma_2[z_1 \mapsto \sigma_2(z)], \delta_3 = \delta_2, \kappa_3 = \kappa_2[z_1 \mapsto \{ p_1 \}].
\end{align*}
\]

Now consider the formulas \( \Phi_x, \Phi_{y, z}, \Phi_{z_1} \) from Example 3 and to illustrate how the semantics work for the box operator, we will do the appropriate checks for those formulas on \( pr \).

The formula \( \Phi_x \) is the conjunction of two box operators, where for the first one because of the behaviour \( p_1 : \text{in}_1(x, x_1) : m \) of the transition \( \gamma_0 \xrightarrow{p_1 : \text{in}_1(x, x_1) : m} \gamma_1 \), we have to check that \( \gamma_1 \models P_x \) where

\[
\begin{align*}
P_x & = (0 < t \land t < 5 \Rightarrow x \subseteq \{ p_1 \}) \\
& \land (5 \leq t \land t \leq 7 \Rightarrow x \subseteq \{ p_1, p_2 \}) \\
& \land (7 < t \land t \leq 10 \Rightarrow x \subseteq \{ p_2 \}).
\end{align*}
\]
This check evaluates to true, since \( \gamma \) satisfies only the guard of the first line of the policy and \( \kappa_1(x) = \{ p_1 \} \). For the transition \( \gamma_1 : mch((y, z) : (1, x)) : d \) \( \gamma_2 \), because of the formula \( \Phi_{y, z} \), we have to check that \( \gamma_2 \models P_y \wedge P_z \) where

\[
\begin{align*}
P_y &= y \subseteq \{ m \}, \\
P_z &= (0 \leq t \land t \leq 7 \land y = 1 \Rightarrow z \subseteq \{ p_1 \}) \land \\
& \quad (5 \leq t \land t \leq 10 \land y = 2 \Rightarrow z \subseteq \{ p_2 \}).
\end{align*}
\]

This check evaluates to true, since \( \kappa_2(y) = \emptyset \) and \( \gamma_2 \) satisfies only the condition at the first line of the policy \( P_z \) and also \( \kappa_2(z) = \{ p_1 \} \). Finally for the last transition \( \gamma_3 : d_{out}(z_1, z) : c_1 \) \( \gamma_3 \), because of the \( \Phi_{z_1} \), formula, we have to check that \( \gamma_3 \models P_z \), and this check evaluates to true, since \( \kappa_3(z_1) = \{ p_1 \} \) and \( P_z = z_1 \subseteq \{ p_1 \} \).

5 Reduction of BTCTL to TCTL+

In this section, we perform a transformation of the original time system, and of the BTCTL formulas. The transformation is based on the work done in [16], where the action-based logic ATCTL (action-TCTL) is being reduced to TCTL [2]. A transformed formula produces a formula in TCTL+, a logic based on TCTL and in the next section we show how a fragment of TCTL+ can be handled by the model checker UPPAAL [30].

5.1 Behaviour Automata

A timed system \( TS = \{ TA_i \}_{i \leq n} \) yields a behaviour automaton \( BA = (v_0, E, I, Q, L) \), which is a kind of timed automaton in that it is the product automaton of the system, extended to contain auxiliary vertices that represent the actions of the system and a labelling function \( L \) that assigns to each vertex a property. A property is either a behaviour or a location invariant on each vertex and \( \Phi \) is a finite set of vertices.

The algorithm for constructing the edges \( E \), the labelling functions \( I \) and \( L \) and the set of vertices \( Q = Q_{gen} \cup Q_{aux} \) (\( Q_{gen} \cap Q_{aux} = \emptyset \)) where \( Q_{gen} \) and \( Q_{aux} \) contain the genuine and auxiliary vertices respectively, is given in Figure 3.

In the first step, we create the genuine vertices and we label them with the invariant of the location vector that they represent; each of those vertices is inserted in \( Q_{gen} \), which will be used in the next steps to create the auxiliary vertices.

In step 2 we create the auxiliary vertices and the edges that correspond to the assignment actions of the system. For each process \( p_i \), we start looking at all of its assignment edges \( (q_i, g \rightarrow \vec{x} := \vec{c}, \vec{r}, q'_i) \in E_i \). For each one of those edges and for all the vertices \( v_s \in Q_{gen} \) and \( v_t \in Q_{gen} \), where the label of \( v_s \), \( L(v_s) \) corresponds to a vector location where this assignment could have been performed and would have moved the system to the location \( L(v_t) \), we create the edges \( (v_s, g \rightarrow \text{skip: } t, v) \) and \( (v, \vec{x} := \vec{c}, \vec{r}, v_t) \), where \( v \) is a fresh auxiliary vertex; whereas in the construction of the product automaton one would have constructed only the edge \( (v_s, g \rightarrow \vec{x} := \vec{c}, r, v_t) \). The auxiliary vertex \( v \) is labelled with the assignment behaviour
are not labelled with behaviours. semantics is similar to the semantics of the timed automata, however now, the transitions

\( \text{action) in its behaviour automaton} \)

\( \text{to a single step transition (in the case of a delay) or a two-step transition (in the case of an} \)

\( \text{automaton} \)

\( \text{BA} \)

\( \text{ trace in the original system} \)

5.2 Trace Equivalence

First, we give the operational semantics of the behaviour automata in Table 2. The semantics is similar to the semantics of the timed automata, however now, the transitions are not labelled with behaviours.

Figure 3 The algorithm for constructing \( \mathcal{E}, \mathcal{I}, \mathcal{Q} \) and \( \mathcal{L} \).

Figure 4 Edge construction of BA.

\( p_i : (\vec{x}, \vec{c}) \) and its invariant is being set to \( (t = 0) \land \mathcal{I}(v_i)[\vec{c}/\vec{x}][\vec{0}/\vec{r}] \), to first ensure that the action of the edge leaving \( v \) will be performed instantaneously and secondly that we can not get stuck at an auxiliary vertex. Figure 4 illustrates the construction and note that each auxiliary vertex \( v \) has exactly one predecessor and exactly one successor.

Similarly to step 2, in step 3 we construct the auxiliary vertices for the communication actions of the system and finally in step 4 we define the set \( \mathcal{Q} \).

5.2 Trace Equivalence

From the construction of the behaviour automaton \( \mathcal{BA} \), it is essential that every execution trace in the original system \( \mathcal{T} \) can be interpreted as an execution trace in the behaviour automaton \( \mathcal{BA} \) and vice versa. Particularly, each transition in the system \( \mathcal{T} \) is equivalent to a single step transition (in the case of a delay) or a two-step transition (in the case of an action) in its behaviour automaton \( \mathcal{BA} \). To overcome the vagueness of this explanation we will later define an equivalence relation between execution traces of the system \( \mathcal{T} \) and the behaviour automaton \( \mathcal{BA} \).

First, we give the operational semantics of the behaviour automata in Table 2. The semantics is similar to the semantics of the timed automata, however now, the transitions are not labelled with behaviours.
21:12 Time Dependent Policy-Based Access Control

Table 2 Semantics for Behaviour Automata.

\[
\langle v_s, \sigma, \delta, \kappa \rangle \rightarrow \langle v_i, \sigma', \delta', \kappa' \rangle \quad \text{if} \quad \begin{cases} 
(q, g \rightarrow x := \bar{e} ; r', v_i) \text{ is in } E \\
[s]{\sigma} = \text{tt} \\
\sigma' = \sigma[x \rightarrow [\bar{e}] \sigma] \\
\delta' = \delta[r \rightarrow \bar{0}] \\
\kappa' = \kappa[x \rightarrow [\bar{r}] \kappa] \\
[I(\langle v_i \rangle)](\sigma', \delta') = \text{tt}
\end{cases}
\]

\[
\langle v, \sigma, \delta, \kappa \rangle \rightarrow \langle v, \sigma', \delta', \kappa' \rangle \quad \text{if} \quad \begin{cases} 
\exists d > 0 : \delta' = \lambda r, \delta(r) + d, \\
[I(\langle v \rangle)](\sigma, \delta') = \text{tt}
\end{cases}
\]

Now let \( \gamma \) and \( \gamma' \) to be two configurations of a timed system \( TS \) and its behaviour automaton \( BA \) respectively. We define the relation \( \cong : \text{Config}_{TS} \times \text{Config}_{BA} \rightarrow \{ \text{tt, ff} \} \) to be

\[
\langle q', \sigma, \delta, \kappa \rangle \cong \langle v, \sigma', \delta', \kappa' \rangle \quad \text{iff} \quad \begin{cases} 
q' = \lambda v, \\
\sigma = \sigma', \\
\forall r \in \text{Clock} : \delta(r) = \delta'(r), \\
\kappa = \kappa',
\end{cases}
\]

where we recall that \( \text{Clock} \) is the set of the clocks appearing in the system \( TS \) and thus the clock \( t \) of the behaviour automaton \( BA \) is not included in \( \text{Clock} \). It is straightforward by the definition of \( \cong \) that configurations of the system \( TS \) can only be related with configurations that correspond to genuine vertices in the behaviour automaton \( BA \).

For the behaviour automata \( BA \), we define a macro transition \( t \) to be a single step delay transition \( \gamma_s \rightarrow \gamma_t \) or a two-step transition \( \gamma_s \rightarrow \gamma_{aux} \rightarrow \gamma_t \), where \( \gamma_{aux} \) is an auxiliary configuration (a configuration that corresponds to an auxiliary vertex) and \( \gamma_s \) and \( \gamma_t \) are genuine configurations (configurations that correspond to genuine vertices). Then lift the definition of \( \cong \) to single step transitions of the system \( TS \) and macro transitions of the behaviour automaton \( BA \) as

\[
\gamma_s \xrightarrow{t} \gamma_t \cong \gamma_s \rightarrow \gamma_t \quad \text{iff} \quad \begin{cases} 
\gamma_s \cong \gamma_s' \\
\gamma_t \cong \gamma_t'
\end{cases}
\]

\[
\gamma_s \xrightarrow{b} \gamma_t \cong \gamma_s' \rightarrow \gamma_{aux} \rightarrow \gamma_t' \quad \text{iff} \quad \begin{cases} 
\gamma_s \cong \gamma_s' \\
\gamma_t \cong \gamma_t' \\
\gamma_{aux} = \langle v, \sigma, \delta, \kappa \rangle \Rightarrow b = \lambda v
\end{cases}
\]

Now for each genuine configuration \( \gamma' \) of the BA, we have that every execution trace \( tr' = \gamma_0' \rightarrow \gamma_1' \rightarrow \gamma_2' \rightarrow \gamma_3' \rightarrow \gamma_4' \) of \( \gamma' \), with length greater than 0, can be parsed as a macro transition trace \( T_{tr'} = t'_1 t'_2 t'_3 \ldots \) where each \( t'_j \) is a macro transition. For example the finite execution trace \( \gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_{aux_1} \rightarrow \gamma_2 \rightarrow \gamma_{aux_2} \rightarrow \gamma_3 \rightarrow \gamma_4 \) which is a sequence of a delay, action(two-step), action(two-step), delay, will produce the macro transition trace \( t'_1 t'_2 t'_3 t'_4 \) where \( t'_1 = \gamma_0 \rightarrow \gamma_1, t'_2 = \gamma_1 \rightarrow \gamma_{aux_1} \rightarrow \gamma_2, t'_3 = \gamma_2 \rightarrow \gamma_{aux_2} \rightarrow \gamma_3 \) and \( t'_4 = \gamma_3 \rightarrow \gamma_4 \).

Similarly to the macro transition traces, for each configuration \( \gamma \) of the system \( TS \), we can write each nonzero-length execution trace, \( tr = \gamma_0 \xrightarrow{b} \gamma_1 \xrightarrow{b} \gamma_2 \xrightarrow{b} \gamma_3 \xrightarrow{b} \ldots \in \text{Trace}_r \) of \( \gamma \), as a transition trace \( T_{tr} = t_1 t_2 t_3 \ldots \) where \( t_i = \gamma_{i-1} \xrightarrow{b} \gamma_i \) (for all \( i \geq 1 \)).
The basic formula
\[ T_{\text{CTL}} + \] 
then results to true if and only if the transition trace \( T_{\text{tr}} \) of \( tr \) and the macro transition trace \( T_{\text{tr}'} \) of \( tr' \) have the same length and they are equivalent stepwise.

The following fact follows from the method of constructing a behaviour automaton and states that equivalent configurations in the timed system \( TS \) and its behaviour automaton \( BA \), produce equivalent execution traces.

**Fact 5.** For every timed system \( TA \), its behaviour automaton \( BA \) and two configurations \( \gamma \) and \( \gamma' \) such that \( \gamma \cong \gamma' \) we have that:

- \( \forall tr \in \text{Trace}_\gamma : \exists tr' \in \text{Trace}_{\gamma'} : tr \cong tr' \),
- \( \forall tr' \in \text{Trace}_{\gamma'} : \exists tr \in \text{Trace}_\gamma : tr \cong tr' \).

### 5.3 \text{TCTL}^+

For the behaviour automata, we define a new logic called \text{TCTL}^+ patterned after TCTL [2], and the syntax of a \text{TCTL}^+ formula \( \psi \) is given by

\[ \psi ::= \lor \psi | \land \psi | \neg \psi | \psi_1 \lor \psi_2 . \]

The basic formula \( \lor \psi \) is a propositional which is either a behaviour or a location vector and it holds in a configuration if its vertex is labelled with \( \lor \psi \); the rest of the basic formulas are the same as in \text{BTCTL}. The \( \forall \Box \psi \) formula holds in a configuration if for all of its execution traces, \( \psi \) holds in all the configurations of the trace, while for the \( \exists (\psi_1 \lor \psi_2) \) to hold, it is sufficient that there exists an execution trace where \( \psi_1 \) holds for a prefix of the trace and eventually \( \psi_2 \) also holds. The rest of the operators are the same as in \text{BTCTL}. The formal semantics of the \text{TCTL}^+ is given by:

\[
\begin{align*}
\gamma' & | \lor \psi & \iff & \gamma' = (v, \sigma, \delta, \kappa) \Rightarrow L(v) = \lor \psi, \\
\gamma' & | \land \psi & \iff & \gamma' = (v, \sigma, \delta, \kappa) \Rightarrow [\Box \psi], \\
\gamma' & | \Box \psi & \iff & \forall \gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \ldots \in \text{Trace}_{\gamma'} : \forall i \geq 0 : \gamma_i' \models \psi, \\
\gamma' & | \exists (\psi_1 \lor \psi_2) & \iff & \exists \gamma_0 \rightarrow \gamma_1' \rightarrow \gamma_2' \ldots \in \text{Trace}_{\gamma'} : \\
& & & \exists i : \gamma_i' \models \psi_1 \text{ and } \forall j < i : \gamma_j' \models \psi_1, \\
\gamma' & | \psi_1 \land \psi_2 & \iff & \gamma' \models \psi_1 \text{ and } \gamma' \models \psi_2, \\
\gamma' & | \neg \psi & \iff & \gamma' \not\models \psi.
\end{align*}
\]

Our goal is to transform a \text{BTCTL} formula \( \phi \) into a \text{TCTL}^+ formula \( \psi \) and then show that for two equivalent configurations \( \gamma \) and \( \gamma' \) of a timed system \( TS \) and its behaviour automaton \( BA \) respectively, checking the formula \( \phi \) in \( \gamma \) it is sufficient to check the transformed formula \( \psi \) in \( \gamma' \) and vice versa. We perform the transformation of the formulas using a function \( T[\cdot] \).
21:14  Time Dependent Policy-Based Access Control

(a) The timed automaton of the process $p$

(b) The behaviour automaton of $p$

as follows

$$T[s] = s,$$

$$T[set_1 \ rel \ set_2] = set_1 \ rel \ set_2,$$

$$T[\forall b (\phi_1, \phi_2)] = \forall b (b \Rightarrow (T[\phi_1] \land \exists (b \ U (\neg b \land T[\phi_2]))),$$

$$T[\phi_1 \land \phi_2] = T[\phi_1] \land T[\phi_2],$$

$$T[\neg \phi] = \neg T[\phi].$$

For the special cases $\forall b (tt, \phi_2)$ ($\phi_2$ is not $tt$) and $\forall b (\phi_1, tt)$ ($\phi_1$ is not $tt$) we shall omit the transformed formula that corresponds to the trivial formula $tt$, by writting $T[\forall b (tt, \phi_2)] = \forall b (b \Rightarrow b \ U (\neg b \land T[\phi_2]))$ for the first case and $T[\forall b (\phi_1, tt)] = \forall b (b \Rightarrow T[\phi_1])$ for the second case. Finally, we shall assume that formulas in the pre-condition of the $\forall b (\phi_1, \phi_2)$ are not nested. To justify this assumption consider the following example

**Example 6.** Consider the timed automaton of a process $p$ (Figure 5a) with a variable $x$ and a clock $r$, and its behaviour automaton $BA$ (Figure 5b), where $b_1 = p : (x, 1)$ and $b_2 = p : (x, 2)$ are the behaviours of the actions $x:=1$ and $x:=2$ respectively, and all the location invariants in the timed automaton of $p$ are $tt$.

Now let $\phi = \forall b_1 (\forall b_2 (tt, x = 1), tt)$ and observe that every initial configuration of the process $p$ does not satisfy $\phi$, whereas every initial configuration of the behaviour automaton does satisfy the transformed formula $T[\phi] = \forall b_1 (b_1 \Rightarrow (\forall b_2 (b_2 \Rightarrow \exists (b_2 \ U (\neg b_2 \land x = 1))))).$

Since the proposed formula transformation is sufficient to express and enforce access control policies of our interest we leave the development of transformations that support the entire BTCTL as future work.

Finally, we state the correctness of the function $T[\_]$ with the following theorem

**Theorem 7.** For a timed system $TS$, its behaviour automaton $BA$, a $BTCTL$ formula $\phi$ and for every configuration $\gamma$ and $\gamma'$ of $TS$ and $BA$ respectively, we have that if $\gamma \models \gamma'$ then

$$\gamma \models \phi \iff \gamma' \models T[\phi]$$

The proof of Theorem 7 can be found in Appendix A.

5.4 Reduction Complexity

We give a computation bound for the algorithm of Figure 3, that given a timed system $TS = (TA_i)_{i \leq n}$ constructs the behaviour automaton $BA = (v, E, I, Q, L)$. Assuming that the computation time of all the simple operations (creation of fresh vertices, setting of invariants e.t.c) is constant, we have that : let $K = |Q_1| + ... + |Q_n|$ and $E = |E_1| + ... + |E_n|$ then the first part of the algorithm is bounded by $K^n$. The second part iterates over the assignement edges and all the pairs of the auxiliary vertices and that is bounded by $E \times K^{2n} \times n$, where
$n$ corresponds to the computation bound of checking the third condition of the branch of
the for-loop. Similarly to the second part of the algorithm the third part is bounded by
$E^2 \times K^{2n} \times n$ and therefore for the total sum of those bounds we obtain a complexity
of $O(E^2 \times K^{2n} \times n)$. Finally, for a BTCTL formula $\phi$ the complexity of the transformation
$T[\phi]$ is linear to the size of $\phi$.

6 The Translator

We have implemented a translator in Java that works together with the model checker
UPPAAL version 4.0 [30]. Figure 6 depicts the architecture of the translator.

UPPAAL is using a graphical interface in which one can model (draw) a system of timed
automata. We first do that and next UPPAAL saves it as a file in the eXtensible Markup
Language (XML) [33]; the xml file together with a text file that contains the desired property
$\phi$ that we want to check, are being passed to the translator. The translator parses the two
files and produces an xml file which contains the behaviour automaton of the system together
with a UPPAAL query file that includes the property $T[\phi]$. The two files are imported to
UPPAAL and then one can check if the desired property holds.

Since UPPALL does not allow nested formulas nor supports the operator $\exists \phi_1 U \phi_2$,
we had to find a workaround for some of the transformed formulas. The guards $g$ are
translated directly; for the $set_1$ rel $set_2$, we model a set as a bit array since UPPALL supports
multidimensional integer arrays and then we check the bit version of the relation rel. In case
of the $T[\forall b(\phi_1, \phi_2)]= \forall b(\phi_1 \land \exists (b \lor T[\phi_2]))$, UPPAAL allows labelling
a vertex with a string (the name of the vertex) and thus auxiliary vertices with label $b$
have as a name a string that corresponds to the behaviour $b$. For the part $b \lor T[\phi_2]$ we
annotate the outgoing edges of the auxiliary vertices with an assignment to a fresh variable
$a$ that works as a switch. We switch on by $a := 1$, only when we leave the auxiliary vertex,
and we switch off by $a := 0$, whenever we leave the successor of the auxiliary vertex. Thus
the formula $b \lor T[\phi_2]$ is transformed into the formula $a = 1 \Rightarrow T[\phi_2]$.

Finally, since the mapping $\kappa$ is not part of the timed automata of UPPAAL, we first
enumerate each variable and each process of the system and we then model $\kappa$ as a two-
dimensional array, whose first index corresponds to a variable and whose second to a process.
For instance, if a variable $x$ is enumerated with 1 and $\kappa(x) = \{ p \}$, where $p$ is a process of
the system and $p$ is enumerated with 2, then $\kappa[1][2] = 1$, while for any other index $j \neq 2$,
$\kappa[1][j] = 0$, modelling in that way that only $p$ has written data in $x$. The edges of the
automaton are also annotated with assignments to $\kappa$ to capture the updates to it whenever
the system performs an action.

7 Conclusions

We have successfully shown how to enforce access control policies on Systems of Timed
Automata using a behaviour-based logic. The logic allows specification of time, data’s content
and information flow dependent security policies, an essential need in the modern world of
cyberphysical systems. We have developed a sound reduction of a substantial fragment of our logic to a logic based on TCTL [2], so that the model checking of the formulas can be performed by existing model checkers such as UPPAAL [30]. We implemented a translator which performs the reduction and together with UPPAAL it enforces access control policies. Finally, we illustrated our development using an example from the aerospace industry, where ensuring data’s integrity is a life critical goal.

There are several ways in which we can extend our work. We are currently exploring how our development can be extended to capture more complex information flows such as implicit flows [31]. We have shown in [24] that the time aspect, as well as the non-deterministic semantics of Timed Automata, poses a challenge for that.

We are considering extensions to our logic that allow expressing richer access control policies and also how to develop a reduction which supports the entire syntax of the BTCTL logic. Another possibility is to explore new algorithms for determining if a formula of our logic holds in a timed system rather than reducing the formula to current TCTL-based logics.

References

14 Yong-Zhong He, Zhen Han, and Ye Du. Context active rbac and its applications. ISECS, pages 1041–1044, 2008.
15 Xuezhen Huang, Jiqiang Liu, and Zhen Han. A privacy-aware access model on anonymized data. INTRUST, pages 201–212, 2014.
A Proof of Theorem 7

Proof. The proof proceeds by structural induction on \( \phi \). The base cases are trivial since \( T[\phi] = \phi \), the formula \( \phi \) does not include any constraint about the clock \( t \), and \( \gamma \equiv \gamma' \).

The case \( \forall \Box b (\phi_1, \phi_2) \).

Assume that \( \gamma \models \forall \Box b (\phi_1, \phi_2) \) and thus by definition:

\[
\forall \gamma_0 \models b \models \gamma_1 b \models \ldots \in \text{Trace}, \forall i \geq 1 : b_i = b \Rightarrow \gamma_{i-1} \models \phi_1 \text{ and } \gamma_i \models \phi_2.
\]
Now take arbitrary trace \( tr' = \gamma'_0 \rightarrow \gamma'_1 \rightarrow ... \in \text{Trace}_{\gamma'} \), where \( \gamma' = \gamma'_0 \) and prove that
\[
\forall j \geq 0 : \gamma'_j \models b \Rightarrow (T[\phi_1] \land \exists(b U (\neg b \land T[\phi_2]))) .
\]

Now if \( tr' \) has length 0 then \( tr' = \gamma' \) and the proof is trivial since \( \gamma' \) is a genuine configuration and thus \( \gamma' \not\models b \). Similarly, if \( tr' \) has length greater than 0 and \( \gamma'_j \) is a genuine configuration then the proof holds. Now if \( \gamma'_j \) is an auxiliary configuration, consider the macro transition trace \( T_{tr'} = t'_1 t'_2 ... \) of the trace \( tr' \) and let \( t'_i \) be the macro transition which corresponds to the transition in which \( \gamma'_j \) is being involved and thus we have that \( T_{tr'}(i) = \gamma'_{j-1} \rightarrow \gamma'_j \rightarrow \gamma'_{j+1} \). Next, let \( \gamma'_j = (v, \sigma, \delta, \kappa) \) and using Fact 5 we have that \( \exists tr \in \text{Trace}_v : tr \equiv tr' \) and thus
\[
\forall h \geq 1 : T_{tr}(h) \equiv T_{tr'}(h)
\]
\[
\Rightarrow T_{tr}(i) \equiv T_{tr'}(i)
\]
\[
\Leftrightarrow \gamma_{i-1} \models b \Rightarrow \gamma_i \models \gamma'_{j-1} \rightarrow \gamma'_j \rightarrow \gamma'_{j+1}
\]
\[
\Leftrightarrow \gamma_{i-1} \models \gamma'_{j-1} \text{ and } \gamma_i \models \gamma'_{j+1} \text{ and } L(v) = b_i
\]

(2)

Now if \( \gamma'_j \not\models b \) then the proof is trivial. Otherwise, because of (2) \( L(v) = b_i \) we have that also \( b_1 = b \) and using (1) we have that \( \gamma_{i-1} \models \phi_1 \) and \( \gamma_i \models \phi_2 \). Next, using (2) \( \gamma_{i-1} \models \gamma'_{j-1} \) and our induction hypothesis we have also that \( \gamma'_{j-1} \models T[\phi_1] \) and since \( \phi_1 \) does not contain any nested formulas we also have that \( \gamma'_{j-1} \models T[\phi_1] \) as required. Finally, using (2) \( \gamma_i \models \gamma'_{j+1} \) and our induction hypothesis we have also that \( \gamma'_{j+1} \models T[\phi_2] \) and thus \( \gamma'_j \models \exists(b U (\neg b \land T[\phi_2])) \) as required.

For the other direction now assume that \( \gamma' \models \forall b \Rightarrow (T[\phi_1] \land \exists(b U (\neg b \land T[\phi_2]))) \) and thus
\[
\forall \gamma_{0}' \rightarrow \gamma'_{1} \rightarrow \gamma'_{2} \rightarrow ... \in \text{Trace}_{\gamma'} : \forall i \geq 0 : \gamma_i' \models b \Rightarrow (T[\phi_1] \land \exists(b U (\neg b \land T[\phi_2])))
\]

(3)

and take arbitrary trace \( tr = \gamma_0 \xrightarrow{b_1} \gamma_1 \xrightarrow{b_2} ... \in \text{Trace}_v \), where \( \gamma_0 = \gamma \) and prove that
\[
\forall j \geq 1 : b_j = b \Rightarrow \gamma_{j-1} \models \phi_1 \text{ and } \gamma_j \models \phi_2 .
\]

For the cases where the length of \( tr \) is 0 or \( b_j \not\models b \) then the proof is trivial. Therefore take \( j \) such that \( b_j = b \) and consider the transition trace \( T_{tr} = t_1 t_2 ... \) of the trace \( tr \) and thus, using Fact 5 we have that \( \exists tr' \in \text{Trace}_{\gamma'} : tr \equiv tr' \) and consequently
\[
\forall h \geq 1 : T_{tr}(h) \equiv T_{tr'}(h)
\]
\[
\Rightarrow T_{tr}(j) \equiv T_{tr'}(j)
\]
\[
\Leftrightarrow \gamma_{j-1} \models b \Rightarrow \gamma_j \models \gamma'_{j-1} \rightarrow \gamma'_j \rightarrow \gamma'_{j+1}
\]
\[
\Leftrightarrow \gamma_{j-1} \models \gamma'_{j} \text{ and } \gamma_j \models \gamma'_{j} \text{ and } \gamma_{j} \models \gamma'_{j} \text{ and } L(v) = b_j .
\]

(4)

Therefore because of (4) \( L(v) = b_j \) and (3) we have that \( \gamma_{aux} \models T[\phi_1] \land \exists(b U (\neg b \land T[\phi_2])) \) and thus since \( \phi_1 \) does not contain any nested formulas, \( \gamma'_j \models T[\phi_1] \) and \( \gamma'_j \models T[\phi_2] \); but then using (4) \( \gamma_{j-1} \models \gamma'_{j} \) and \( \gamma_{j} \models \gamma'_{j} \) and our induction hypothesis we get the required result.

The cases \( \phi_1 \land \phi_2 \) and \( \neg \phi \) can be proved straightforwardly using structural induction on \( \phi_1, \phi_2 \) and \( \phi \). 

\[\square\]