Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 190 (2007) 79–94

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Keeping Secrets in Resource Aware Components

Tom Chothia

CWI, Amsterdam, The Netherlands

Jun Pang

Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany

Mohammad Torabi Dashti

CWI, Amsterdam, The Netherlands

Abstract

We present a powerful and flexible method for automatically checking the secrecy of values inside components. In our framework an attacker may monitor the external communication of a component, interact with it *and* monitor the components resource usage. We use an automata model of components in which each transition is tagged with resource usage information. We extend these automata to pass values and say that a value is kept secret if the observable behaviour of the automata is the same for all possible instantiations of that value. If a component leaks some, but not all of the information about its secret we use a notion of *secrecy degree* to quantify the worst-case leakage. We show how this secrecy degree can be automatically calculated, for values from a finite domain, using the μCRL process algebraic verification toolset.

Keywords: Secrecy, Q-Automata, automatic checking, component-based systems, quality of service, μCRL

1 Introduction

Component-based software development allows programs to be reused, interchanged and even downloaded onto a running system. Sometimes the maker or user of a component would like to keep some of the data inside the component secret from other components on the same system. This goal is complicated as the potential attackers may be running on the same computer as the target component and so can monitor its resource usage. We develop a framework for automatically checking how well values inside a component are kept secret from an attacker that can communicate with the component *and* monitor its resource usage.

doi:10.1016/j.entcs.2007.07.006

Resource usage or Quality of Service (QoS) aspects of components concern non-functional properties such as availability, response time, memory usage, etc. Following work on constraint semirings [4,13], we propose a general framework for a range of trust and quality values, which we call a Q-algebra. These algebras define a framework for quality values that could be combined with many kinds of automata or calculi. With the aim of making our system suitable for the kinds of applications we expect to model and of making our system more easily understandable, we have chosen to base our work on automata. These provide a concrete, intuitively clear, model of computation, and a structural approach to the analysis of the behaviour of components and their composition.

Components will be represented by Q-automata [10]. These automata have an additional cost label on each transition to indicate the impact of taking that transition on the quality attributes of the system. Most automata models do not distinguish between the interleaving of two actions and their possible concurrent occurrence, however in the model proposed here it is possible that concurrent components can perform their actions simultaneously without having to synchronise (e.g., in a communication). Therefore, the resource usage of an application can be quite different depending on whether the smallest units of abstraction happen at the same time or one after the other. For instance, given two transitions, both of which "cost" a certain amount of bandwidth, measured in Kbit/s, running both at the same time will require the sum of the two individual costs, whereas running them one after the other will only cost the maximum of the two individual costs. Time costs on the other hand will sum sequentially, but not concurrently and we may choose to model memory allocation costs by summing them both concurrently and sequentially.

We extend Q-automata to pass names from a finite domain of values in a similar way as non-value passing process calculi are often extended to value passing versions, c.f. [20]. We also define strong bisimulation as an equivalence relation for the (extended) Q-automata model that requires matching costs on transitions.

We define *Component Secrecy Degree* and *Variable Secrecy Degree* to measure how well a model of a component keeps its secrets from any attacker who can interact with the component and monitor its resource usages. We do not consider an attacker that can examine the source code of a component, we are interested in modelling attackers that are other components running on the same computer as the target and so they can only use the standard communication channels and monitor the resource usage of other components. A component has prefect secrecy if any two instances of that component, observed by the attacker, are bisimilar for any possible secret values. The degrees of secrecy give a measure of how much information about the components secrets are leaked. This is similar to the approach of [11] in measuring anonymity degrees. We use the μCRL toolset [5] for performing state space reduction modulo bisimulation [6], which we use along with some purpose-built scripts to generate all possible cases of the model and to calculate degrees of secrecy.

Main contributions.

The contributions of this work include

- a resource aware automata model of components with secrets,
- a corresponding definition of secrecy and a method of checking it via bisimulation,
- a tool to automatically check our measures of secrecy (based on μCRL).

Related work.

Weighted automata have a simple weight or cost on each transition and have been extensively studied since the early days of computer science [14,22]. Our automata model differs from weighted automata by using a Q-algebra to provide the costs; this allows us to define a truly compositional model of the resource usage of components. Timed automata models label transitions with costs representing the time they take [1]. Priced or weighted timed automata [2,3] model time using clocks and have costs on states and transitions. The cost of each transition is paid each time the transition is made whereas the costs of each state is paid once for each time unit the automata spends in that state. This provides an expressive model of costs and time that is, in many cases, undecidable [8]. Our model is also similar to some process calculi, such as CCS [19], one of the main differences, apart from the automata setting, is that we allow multisets of actions to happen at the same time.

In the context of computer security, there are several papers on modelling resource consumptions of different execution scenarios, [9,17]. A conceptual difference between our work and these papers is that they mainly focus on measuring the resources consumed by the attacker in an attack, whereas we allow the attacker to measure the resources consumed by the target system, and possibly use it to launch an attack. In this respect, our work is close to the body of research on side channel and timing attacks [15,18,21].

Structure of the paper.

In the next section we introduce our automata model and cost algebra. In Section 3 we extend these automata to pass values and define secrecy degrees for these values. Section 4 introduces our automatic tool to check secrecy. Finally Section 5 concludes the paper.

2 Q-Automata: Modelling Resource Usage

Our model of resource aware components uses values from a *Q-algebra* to label the transitions of an automaton.

2.1 Q-algebra

To compute and analyse QoS values in a standard way, we develop a general framework as in the approach of De Nicola *et al.* [13]. First we recall the concept of a constraint semiring:

Definition 2.1 A *constraint semiring* is a structure $R = (C, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ where C is a set, $\mathbf{0}, \mathbf{1} \in C$, and $\oplus$ and $\otimes$ are binary operations on C such that:
— $\oplus$ is commutative, associative, idempotent and has identity $\mathbf{0}$
— $\otimes$ is associative and has identity $\mathbf{1}$
— $\otimes$ distributes over $\oplus$ and has $\mathbf{0}$ as an absorbing (zero) element

Note that for a constraint semiring (or *c-semiring*, for short) as above, the operation $\oplus$ induces a partial order $\leq$ on C defined by $a \leq b$ if and only if $a \oplus b = b$. Moreover, two elements are comparable with respect to $\leq$ if and only if application of $\oplus$ to these elements yields one (the larger w.r.t. $\leq$) of the two. Actually, $\oplus$ always yields the least upper bound of the elements to which it is applied.

Constraint semirings can be used to compose QoS values with "addition" $\oplus$ to select among values and "multiplication" $\otimes$ to combine them. Given an action of cost c_1 and another action with cost c_2 then the cost of both actions together is $c_1 \otimes c_2$, whereas $\oplus$ returns the least upper bound of c_1 and c_2. The $\mathbf{0}$ element, as the identity of $\oplus$, is the least possible cost value and the $\mathbf{1}$ element, as the identity of $\otimes$, is the neutral cost value.

A few examples:
— (shortest) time: $(\mathbb{R}_+ \cup \{\infty\}, \mathtt{min}, +, \infty, 0)$
— bandwidth: $(\mathbb{N} \cup \{\infty\}, \mathtt{min}, \mathtt{max}, \infty, 0)$
— data encrypted: $(\{\mathtt{true}, \mathtt{false}\}, \vee, \wedge, \mathtt{false}, \mathtt{true})$
— access control: $(2^U, \cup, \cap, \emptyset, U)$, where U is the set of all users and 2^U is the powerset of U.

Constraint semirings work well when there is just one way to combine quality values. We may use these values to represent the cost of a method call, a sequence of reduction steps or the cost to execute an entire program. When dealing with a number of concurrent processes these steps may take place sequentially or in parallel and these two ways of combining actions might have very different overall results on the resource usage of the system. For instance, two processes that both require a certain number of CPU cycles per second will require a higher number of cycles per second when run at the same time than when run one after the other. We can model these different ways of combining values by adding a new multiplicative operator:

Definition 2.2 A *Q-algebra* is a structure $R = (C, \oplus, \otimes, \varobar, \mathbf{0}, \mathbf{1})$ such that $R_\otimes = (C, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ and $R_\varobar = (C, \oplus, \varobar, \mathbf{0}, \mathbf{1})$ are c-semirings. C is called *the domain of* R.

The $\varobar$ operator is used to combine two values concurrently: $c_1 \varobar c_2$ is the cost of c_1 and c_2 at the same time. The $\otimes$ operator combines values sequentially: $c_1 \otimes c_2$ is the cost of c_1 followed by c_2. Combining costs concurrently or sequentially will not affect the least or neutral cost elements so the two operations share their identities. As before, $\oplus$ is used to select between values. For example:
— (shortest) time: $(\mathbb{R}_+ \cup \{\infty\}, \mathtt{min}, +, \mathtt{max}, \infty, 0)$
— bandwidth: $(\mathbb{N} \cup \{\infty\}, \mathtt{min}, \mathtt{max}, +, \infty, 0)$

2.2 Q-Automata

In this section we introduce Q-automata. These consist of an initialised labelled transition system together with a (labelled) Q-algebra to specify the cost of each transition. Note that each transition is labelled with a multiset of actions as a representation of simultaneous and multiple occurrences of actions. A multiset over a set X is a function $m : X \to \mathbb{N}$ and the set of all multisets over X is denoted by $\mathbb{M}(X)$. For two multisets m_1 and m_2 over X, their sum $m_1 + m_2$ is the multiset over X defined by $(m_1 + m_2)(x) = m_1(x) + m_2(x)$.

Definition 2.3 A *Q-automaton* is a structure $P = \langle S, t, A, R, T\rangle$ where:
— S is a finite set of *states*,
— $t \in S$ is its *initial state*,
— A is a (finite) set of *action names*,
— $R = (C, \oplus, \otimes, \oplus\!\!\!\!|\,, \mathbf{0}, \mathbf{1})$ is a labelled QoS algebra with domain C of *costs*,
— and $T \subseteq S \times \mathbb{M}(\mathit{Act}) \times C \times S$ is the set of transitions.
The set of *actions* of P, written Act, is derived from the set of action names A in the following way: each name $a \in A$ can occur as an *input* action (denoted $a?$), an *output* action (denoted $a!$) or as an internal action (also denoted by a). We thus obtain $A^O = \{a! : a \in A\}$, the set of output actions of P, $A^I = \{a? : a \in A\}$, the set of input actions of P, and $A^\tau = A$ the set of internal actions of P. The sets A^O, A^I, and A^τ are assumed to be pairwise disjoint. Finally, we let $\mathit{Act} = A^O \cup A^I \cup A^\tau$.

The (finite) computations of Q-automata are defined in the standard way.

Definition 2.4 Let P be a Q-automaton specified as in Definition 2.3. A *computation (of length $n \geq 0$)* starting from a state $s_0 \in S$ is a sequence $(s_0, m_1, c_1, s_1), \ldots,$ (s_{n-1}, m_n, c_n, s_n) with $(s_i, m_i, c_i, s_{i+1}) \in T$ for all $0 \leq i \leq n-1$. If $n = 0$, then the computation is the empty sequence.

Based on the Q-algebra and the costs of the transitions, we can compute the cost for each computation.

Definition 2.5 Let $\gamma = (s_0, m_1, c_1, s_1), \ldots, (s_{n-1}, m_n, c_n, s_n)$ be a computation as specified in Definition 2.4. Then the *cost of* γ is $\mathbf{1}$, if $n = 0$ and $c_1 \otimes \ldots \otimes c_n$ if $n \geq 1$.

So, the cost of a computation (a sequence of transitions) is computed using the "sequential multiplication" operator $\otimes$. Note that to compare the costs of different computations, the additive (selection) operation $\oplus$ can be used since it yields the least upper bound of the given values. "Concurrent multiplication" $\oplus\!\!\!\!|\,$ is used when Q-automata collaborate in a composite automaton (their product). This product automaton has as its Q-algebra the product of the Q-algebras of its components. Its state space is the Cartesian product of the state spaces of its components and its transitions are combinations of the components' transitions, as defined below.

In the formal definition of product automata, we use an auxiliary function *sync*, which in turn uses a relation $\Rightarrow$ over pairs of multisets of actions such that the pair on the left equals the pair on right except one input on a name in one multiset and one output on the same name in the other multiset have been removed and a communication on that name has been added.

Definition 2.6 Let A be a set of action names, Act be its associated set of actions as defined in Definition 2.3 and m_1 and m_2 be two multisets over Act. Then $(m_1, m_2) \Rightarrow (m'_1, m'_2)$ if either there exists an $a \in A$ such that:
— $m_1(a?) \geq 1$ and $m_2(a!) \geq 1$
— $m'_1(a) = m_1(a) + 1$ and $m'_1(a?) = m_1(a?) - 1$,
— $m'_2(a!) = m_2(a!) - 1$,
— and $m'_1(b) = m_1(b)$ and $m'_2(b) = m_2(b)$ for all other actions $b \in Act$.
or if $(m_2, m_1) \Rightarrow (m'_2, m'_1)$ as above.

Let $\Rightarrow^*$ be the reflexive, transitive closure of $\Rightarrow$. Then
$sync(m_1, m_2) = \{m'_1 + m'_2 :$ for all m'_1, m'_2 such that $(m_1, m_2) \Rightarrow^* (m'_1, m'_2)\}$.

Thus $sync(m, m')$ is the set of all multisets that can be obtained by adding m and m' with any possible combination of communications between them. Note that $\Rightarrow^*$ in particular allows the multisets not to communicate at all.

Definition 2.7 Let $P_1 = \langle S_1, t_1, A_1, R, T_1\rangle$ and $P_2 = \langle S_2, t_2, A_2, R, T_2\rangle$ be two Q-automata. Then their product, denoted by $P_1 \boxtimes P_2$, is the Q-automaton defined as $P_1 \boxtimes P_2 = \langle S, t, A, R, T\rangle$ with
— $S = S_1 \times S_2$,
— $t = (t_1, t_2)$,
— $A = A_1 \cup A_2$,
— $T = T_1^{new} \cup T_2^{new} \cup T^{joint}$ where:

- $T_1^{new} = \{((s, t), m, c, (s', t)) : (s, m, c, s') \in T_1$ and $t \in S_2\}$,
- $T_2^{new} = \{((s, t), m, c, (s, t')) : s \in S_1$ and $(t, m, c, t') \in T_2\}$, and
- $T^{joint} = \{((s, t), m, c, (s', t')) : \exists (s, m_1, c_1, s') \in T_1, (t, m_2, c_2, t') \in T_2$ such that $m \in sync(m_1, m_2)$ and $c = c_1 \oplus c_2\}$.

In some cases we may want to impose a more restrictive model of communication on our automata, for instance we might want to require that only a single automaton can receive on a given channel or we might want to test our automata in the knowledge that no other automaton will ever be listening on some channel. We can do this by blocking all transitions that involve a given (internal, input, or output) action.

Definition 2.8 Let $P = \langle S, t, A, R, T\rangle$ be a Q-automaton and let $\alpha \in A^O \cup A^I \cup A^\tau$ be an action of P. Then $P\backslash\alpha$, *the restriction of P with respect to α*, is the Q-automaton $\langle S, t, A, R, T'_P\rangle$ with $T'_P = \{(s, m, c, s') : (s, m, c, s') \in T_P$ and $m(\alpha) = 0\}$. We define $P \setminus \emptyset = P$ and $P \setminus (\{\alpha\} \cup Y) = (P \setminus \alpha) \setminus Y$.

3 Secret Values

3.1 Value Passing Q-automata

This subsection provides the additional machinery that we need to define and test secret values. We extend Q-automata to pass names from an ordered, finite data domain of values. We add a finite set of variables V and a finite domain of values D to the definition of Q-automata. Our definitions can naturally be extended to cover cases where different variables have separate, yet finite, data domains. However, to keep the presentation simple, we confine to the single value domain case in this paper. Below, we use $x, y, \ldots$ to refer to variables, $d_1, d_2, \ldots$ are elements of the data domain and $n_1, n_2, \ldots$ refer to either a variables or an element of the domain.

Definition 3.1 An *extended Q-automata* is a structure $P_e = \langle S, t, A, R, D, T \rangle$ where:
— S is a finite set of *states*,
— $t \in S$ is its *initial state*,
— A is a (finite) set of *action names*,
— $R = (C, \oplus, \otimes, \oplus, \mathbf{0}, \mathbf{1})$ is a labelled QoS algebra with domain C of *costs*,
— D a finite domain of data values,
— and $T \subseteq S \times \mathbb{M}(\mathit{Act}) \times C \times S$ is the set of transitions.
The set of *actions* of P, written Act, contains four types of actions: Input on a name from a segment of the domain $a?(x)[d_1, \ldots, d_2]$, output on a name $a!(n_1)$, internal action on a name $a(n_1)$, and conditionals on data values *if* $(n_1 = n_2)$ as well as *if* $(n_1 \neq n_2)$. [1]

The input action $a?(x)$ "binds" the variable x. We say that a variable is "free" in a multiset of actions if it is used in an output or in a conditional but is not bound by an input. Furthermore, we say that a variable is free from a given state if there exists a trace, starting at that state, in which the variable appears free before it is bound.

Below we define well-formed extended Q-automata.

Definition 3.2 An extended Q-automata P is *well-formed* if

- None of the multisets of actions, on the transitions, bind the same variable more than once.
- There are no free names from the start state.

From now on, we only consider well-formed extended Q-automata. We define the semantics of value passing automata by mapping them back into the non-value passing automata. We do this by mapping each combination of name and data value to a single name. For instance, the input action a?(x)[1,. . . ,4] would be mapped to four different input actions $a\_1?, \cdots, a\_4?$, where each of these is a variable-free action in the basic system. To translate the "*if*" condition, we test the values and

[1] This can in principle be easily extended to any Boolean function as conditionals.

if the condition holds then we replace it with an empty transition otherwise we remove it.

We record the mappings from variables to values using a partial function $\sigma : V \to D$. An action $a?(x)[d_1, \cdots, d_n]$ and a partial function σ are *compatible* if $\sigma(x) \in \{d_1, \cdots, d_n\}$, we write $a\_\sigma(x)?$ for $[a?(x)[d_1, \cdots, d_n]]_\sigma$, when σ is compatible with the action. For example if $\sigma = \{(x, 4)\}$, then $[a?(x)[1, \cdots, 5]]_\sigma$ is written as $a\_4?$. We note that given a partial function σ that does not map x to a value, the smallest extensions of σ that are compatible with the action $a?(x)[d_1, \cdots, d_n]$ are $\sigma \cup \{x \mapsto d_1\}, \ldots, \sigma \cup \{x \mapsto d_n\}$.

For output actions $[a]_\sigma$ is obtained by simultaneously substituting all variables for their values and forming the matching single output name. For instance, given $a!(x)$ and $\sigma = \{(x, 3)\}$, we have $[a!(x)]_\sigma = a\_3!$. We extend these definitions to multisets of actions in a natural way. Note that the well-formedness conditions mean that there is no conflict between variable names.

Definition 3.3 Given an extended Q-automaton $P_e = \langle S_e, t_e, A_e, R_e, D_e, T_e \rangle$ we define its mapping to a basic Q-automata $[\![P_e]\!] = \langle S, (t_e, \emptyset), A_e, R_e, T \rangle$ as follows: Each state in S is a pair (s_e, σ), where $s_e \in S_e$ and $\sigma : V \to D$ is a partial function that assigns concrete values to members of V. We define S and T to be the smallest sets such that:

- $(t_e, \emptyset) \in S$
- For all $(s, \sigma) \in S$ and for all $(s, m, c, s') \in T_e$:
  - · either there exists an “if” condition in m which is false using σ
  - · or we remove the names bound by m from σ and define S to be the set of the smallest extensions of that substitution that are compatible with m, then for all $\sigma' \in S$:

$$((s, \sigma), [m]_{\sigma'}, c, (s', \sigma'')) \in T \qquad \text{and} \qquad (s', \sigma'') \in S$$

where σ'' equals σ' with all names that are not free from state s' removed.

The removal of the names from σ' ensures that irrelevant extensions to the substitution mapping do not lead to more states than are necessary.

A parameterised Q-automata $P_e(x_1, \ldots, x_n)$ is an extended Q-automata that has free variables in its computations, hence it is not well-formed. However, an instantiated $P_e(x_1, \ldots, x_n)$ with a mapping $\sigma_0 = \{(x_1, d_1), \cdots, (x_n, d_n)\}$ is well-formed and can be translated to a basic Q-automata $[\![P_e(d_1, \ldots, d_n)]\!]$ with starting the procedure introduced in Definition 3.3 with the initial state (t, σ_0), instead of $(t, \emptyset)$.

3.2 Hiding and Bisimulation

We do not want the attacker to be able to observe the internal actions of any automaton, therefore we define a hiding operator that removes the names from the internal actions but leaves the transition and its costs.

Definition 3.4 Given a Q-automata $P(\vec{x}) = \langle S, t, A, R, T \rangle$ and a set of internal names $\mathcal{I}$, the automata that results from hiding these names is $P(\vec{x})\{\mathcal{I}\} = \langle S, t, (A \backslash \mathcal{I}), R, T' \rangle$ where:

$$T' = \{(s, m', c, s') \ : \ (s, m, c, s') \in T \text{ and for all } a \in \mathcal{I}.\ m'(a) = m'(a!) = m(a?) = 0 \\ \text{and } m(b) = m'(b) \text{ for all } b \neq a \ \}$$

We define *strong bisimulation* for automata with costs as follows:

Definition 3.5 We say two Q-automata $P_1 = \langle S_1, t_1, A_1, R_1, T_1 \rangle$ and $P_2 = \langle S_2, t_2, A_2, R_2, T_2 \rangle$ are *bisimilar*, denoted by $P_1 \sim P_2$, iff

- for all $(t_1, m, c, s_1) \in T_1$, then there exists $(t_2, m, c, s_2) \in T_2$ such that $\langle S_1, s_1, A_1, R_1, T_1 \rangle$ and $\langle S_2, s_2, A_2, R_2, T_2 \rangle$ are bisimilar.
- for all $(t_2, m, c, s_2) \in T_2$, then there exists $(t_1, m, c, s_1) \in T_1$ such that $\langle S_2, s_2, A_2, R_2, T_2 \rangle$ and $\langle S_1, s_1, A_1, R_1, T_1 \rangle$ are bisimilar.

Two extended automata are bisimilar if their mappings into the basic automata are also bisimilar. We note that this definition requires the names and the costs on an action to match, therefore if two automata perform the same actions at different costs they are not bisimilar.

The equivalence used in the verification of secrecy models the observation power of the intruder. We use bisimulation to equate processes; this is in contrast to some previous work that used trace equivalence for secrecy, e.g. see [12]. While it is often possible, in an asynchronous setting, to implement processes in such a way that an intruder cannot tell the difference between two processes that are trace equivalent but not bisimilar, there also exist reasonable implementations in which the intruder can tell the difference. For instance, the two processes $a.(b + c)$ and $a.b + a.c$ are trace equivalent but not bisimilar. A reasonable implementation of these processes might use sockets for communication, in which case the first process would listen on port “a” for a message and then listen on ports “b” and “c” and accept only the first message that arrives. The second process could be implemented by either listening on port “a” and then port “b” or listening on port “a” and then port “c”. All an intruder has to do to tell these processes apart is to send on port “a” and then on port “b”. If the intruder can connect on port “b” they learn nothing, however if they find that port “b” is not open then they know that they are dealing with the second process. In this sense, using bisimulation rather than trace equivalence is a conservative decision; while it is possible for processes that are trace equivalent, but not bisimilar, to be safe, we cannot guarantee that they do not reveal information to the intruder.

A second advantage of using bisimulation is that it can be much more efficient to check. The added restrictions on bisimilar processes mean that we can reject certain paths as not bisimilar long before we could detect that they are not trace equivalent. In the most extreme cases checking a particular pair of processes for trace equivalence can take exponential time while checking the same processes for bisimulation being linear time, e.g. see [16].

3.3 Secrecy Degree

We say that a variable x in the automata $P(x)$ is kept secret if for any two possible values d_1 and d_2 for x it holds that $P(d_1)\{\mathcal{I}\}$ is bisimilar to $P(d_2)\{\mathcal{I}\}$ where $\mathcal{I}$ is the set of the internal names.

Different instantiations of an automaton are split into different equivalence classes whose members are all bisimilar. The Component Secrecy Degree measures the smallest of these classes. This represents the most that an attacker could deduce about the automata's secret (input) based on its communication and resource usage. Below, $\vec{z}$ denotes a vector of z values.

Definition 3.6 [Component Secrecy Degree]

The component secrecy degree (csd) of an automata $P(\vec{z})$ with respect to a set of internal names $\mathcal{I}$ is:

$$csd = \min_{\vec{d_1}} \mid \{\vec{d_2} \ : \ [\![P(\vec{d_1})]\!]\{\mathcal{I}\} \sim [\![P(\vec{d_2})]\!]\{\mathcal{I}\} \ \} \mid$$

The best possible Component Secrecy Degree would be $|D|^n$ where n is the size of $\vec{z}$ and $|D|$ is the size of the domain. So we will sometimes write the component secrecy degree as $csd : |D|^n$.

In the case that only one of our inputs needs to be kept secret we use Variable Secrecy Degree:

Definition 3.7 [Variable Secrecy Degree]

The variable secrecy degree (vsd) for the i^{th} variable of automata $P(\vec{z})$ with respect to a set of internal names $\mathcal{I}$ is:

$$vsd(i) = \min_{\vec{d}} \mid \{d_i \ : \exists d_1, \ldots, d_n \ [\![P(d_1, \ldots, \mathrm{d}_i, \ldots, d_n)]\!]\{\mathcal{I}\} \sim [\![P(\vec{d})]\!]\{\mathcal{I}\} \ \} \mid$$

The Variable Secrecy Degree quantifies the secrecy of just one input value. The best possible Variable Secrecy Degree would be $|D|$. So we for the value secrecy degree will sometimes write vsd : $|D|$. Note that csd is in fact an upper bound on vsd values, i.e. $\forall i.\ vsd(i) \leq csd$. This is because csd is the size of the smallest equivalence class whose members are indistinguishable for the attacker. Clearly, the number of different values that the i^{th} variable may take in this class cannot be larger than the size of the class.

We may use restriction to test conditions on automata. For instance if we have a booking agent component that may make purchases on our behalf, then we could check that it does not leak our credit card number without making a purchase by checking the variable secrecy degree for the credit card number in the automata restricted on the *purchase* action.

We conjecture that the secrecy degrees are fully compositional properties, i.e. for any automata $P(\vec{x})\{\mathcal{I}\}$ and P_A, the secrecy degrees of $P(\vec{x})\{\mathcal{I}\} \times P_A$ are not less than the corresponding secrecy degrees of $P(\vec{x})$. Similarly, the secrecy degrees of $P(\vec{x})\{\mathcal{I}\} \setminus A_\alpha$, where A_α denotes a set of restricted actions, is not less than those of $P(\vec{x})\{\mathcal{I}\}$. This means that we can quickly build up quite complex models and check combinations of systems. We leave a formal treatment of this conjecture as

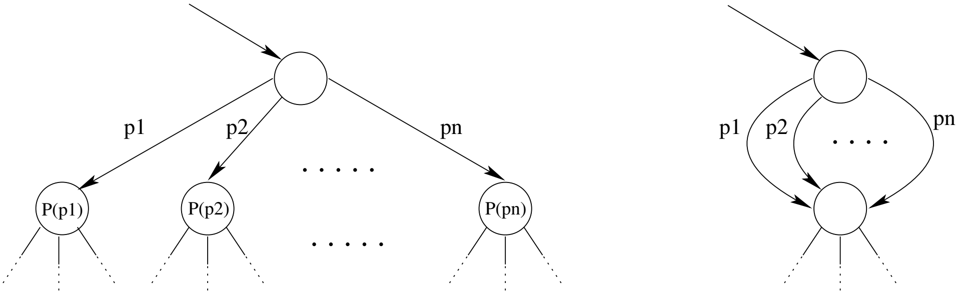

Fig. 1. Checking Secrecy Using State Space Reduction

future work.

4 Automatically Checking Secrecy in μCRL

Our definitions of secrecy require us to check bisimulations between every possible pair of automata inputs. Doing this by hand would not be easy, therefore we develop a tool to do this for us.

The user specifies an extended Q-automata, the finite domain of its variables, and the names that need to be restricted and the internal names. Our scripts generate all possible basic Q-automata for all possible variable values, and also hide and restrict the names. We then test all the possible bisimulations in one go via "state space reduction modulo bisimulation", see [7].

We illustrate our method in Figure 1. We make use of this to test an automaton $P(\vec{x})$ by generating one large automaton that has one transition from the start state for each possible input value, and these transitions go to a sub automata that behave as P would on that input. More formally:

Given an extended automata $P(\vec{x})$, a domain of variables $\{d_1, \ldots, d_n\}$ and a set of internal names $\mathcal{I}$, we generate all corresponding basic Q-automata for all possible inputs: $[\![P(d_1)]\!]\{\mathcal{I}\} = \langle S_1, t_1, A_1, R, T_1\rangle, \cdots, [\![P(d_n)]\!]\{\mathcal{I}\} = \langle S_n, t_n, A_n, R, T_n\rangle$. We then generate an automaton that branches to each of these automata from a start state: $P = \langle S, t', A, R, T\rangle$ where:

- $S = S_1 \cup \ldots \cup S_n \cup \{t'\}$
- $A = A_1 \cup \ldots \cup A_n \cup \{i : 1 \leq i \leq n\}$
- $T = T_1 \cup \ldots \cup T_n \cup \{(t', i, \mathbf{1}, t_i) : 1 \leq i \leq n\}$

This automaton is illustrated on the left of Figure 1. We perform state space reduction modulo bisimulation on this large automaton. [2] If a number of values the $P(x)$ automata are bisimilar then the state space reduction will produce an automaton that takes the initial transition with the value labels to the same state. If the result is a system in which all of the initial transitions go to the same state, as illustrated on the right of Figure 1, we know that $P(x)$ is bisimilar for all values of x, i.e. we have perfect secrecy. If the transitions go to a number of distinct states

[2] State space reduction modulo bisimulation reduces an automaton to a smaller automaton that is bisimilar.

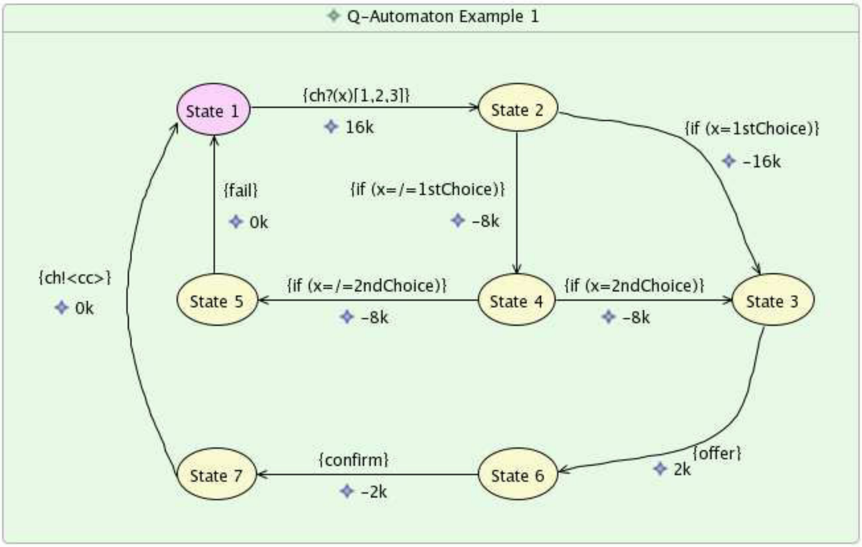

Fig. 2. An Example Extended Q-Automaton

then the attacker can learn something about the input.

We can measure how much the attacker learns using our secrecy degree, which we can read off from the simplified automaton. If we look at the set of labels on the transitions that go to the same state after simplification as equivalence classes then the component secrecy degree is the size of the smallest class. To find the variable secrecy degree it is necessary to look at each of the equivalence classes in turn and find the one that has the smallest number of possible values for the given variable. This smallest number of possible different values is the variable secrecy degree.

4.1 Example

We illustrate the details of our method with a simple example. Figure 2 shows an extended Q-automaton that two "choices" parameters (numbers between 1 and 3, which are different) and a credit card number parameter (either cc1 or cc2). It then inputs a number on the channel *ch* that represents what the environment is offering. If this number matches one of the two choices, the component reports the offer back to its owner on a private channel (this is an internal action) and if the owner confirms, it releases the credit card number. In order to automatically check our extended Q-automata we write them in Aldebaran like format:

```
(0,"ch?(x)[1,2,3]",16,1)            (1,"if (x=1stchoice)",-16,2)
(1,"if (x=/=1stchoice)",-16,3)      (3,"if (x=2ndchoice)",-8,2)
(3,"if (x=/=2ndchoice)",-8,4)       (4,"fail",0,0)
(2,"offer",2,5)                     (5,"comfirm",-2,6)
(6,"ch!<cc>",0,0)
```

The first thing we can check is that the credit card number is kept secret unless the owner confirms its release. So we test the automata restricted on the *confirm* action hoping to find that the credit card number does not then affect the behaviour

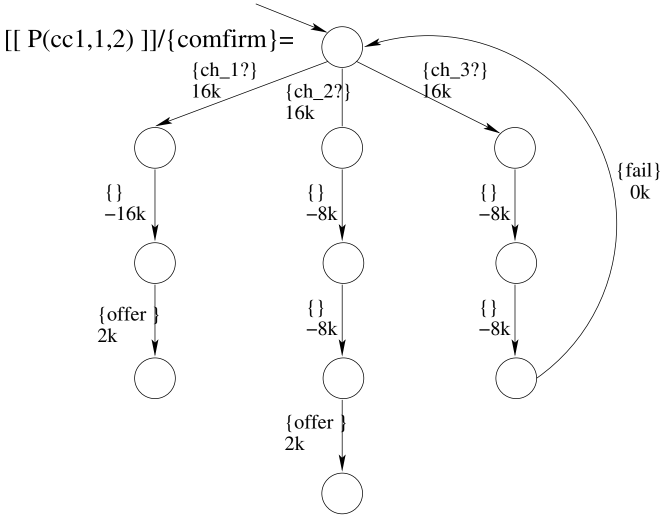

Fig. 3. An Basic Q-Automaton Example

of the automaton. To test this secrecy we need to translate the restricted automata into a basic Q-automaton, this is done for the inputs cc1,1,2 in Figure 3.

By replacing the multiset of actions with an ordered list of actions and appending the costs to this list we can write down these basic automata in the Aldebaran like format. For instance, the automata in Figure 3 would be written as:

```
(0,"ch_1_16k?",1)                    (0,"ch_2_16k?",2)
(0,"ch_3_16k?",3)                    (1,"-16k",4)
(2,"-8k",5)                          (3,"-8k",6)
(4,"offer_2k",7)                     (5,"-8k",8)
(6,"-8k",9)                          (8,"offer_2k",10)
(9,"fail_0k",0)
```

We have written a Perl script that, given a value passing Q-automaton and information on the domains of its variables, generates the large automaton that includes all possible inputs, as described in Section 4, we then use μCRL toolset to perform state space reduction modulo strong bisimulation. Finally another script reads off our secrecy values by looking at all the transitions from the start state in the reduced version of the automaton.

When analysing the example in Figure 2 we find that, as expected, the credit card value is kept secret ($vsd(1) = 2 : 2$) however we also find that the choices are not secret at all ($vsd(2) = vsd(3) = 1 : 3$). On closer inspection we can see that the first and second choices can be distinguished by the way in which memory is deallocated. If the environment provides the component's first choice then the component will stop its search and deallocate 16k in one go, whereas if the environment offers the 2nd choice value then the component will deallocate 8k followed by another 8k.

It is up to the user of our tools to decide if leaking information about a particular variable is a problem or not. In this case we may restore some secrecy to the choices by deallocating the memory in the same way for each choice, as done in Figure

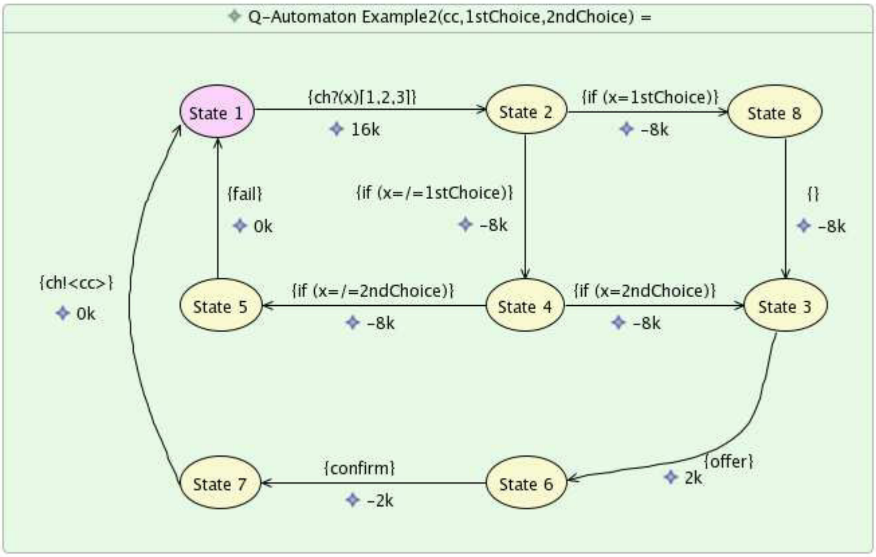

Fig. 4. An Example Extended Q-Automaton That Hides the Choices

4. For this automata our tools tell us that $vsd(1) = 2 : 2, vsd(2) = 2 : 3$ and $vsd(3) = 2 : 3$. While this is not perfect secrecy, these increased secrecy degrees show that the attacker cannot learn which value was the first choice and which was the second.

5 Conclusion

We have presented a method for automatically checking the secrecy of values inside components, even when the attacker can monitor resource usage. We base our framework on Q-automata and Q-algebras to provide a simple model of components and a wide range of possible costs. We use scripts and μCRL to automatically calculate our measures of secrecy degree from an automaton.

This work forms part of the analysis methods developed within the Trust4All project. This is an ITEA project aimed at developing a programming environment for "trusted" components that will come with information on their resource usage. The methods and tools presented here will be used to check real components developed as part of this project.

We are currently developing an Eclipse based graphical user interface to make building the automata models easier. Figures 2 and 4 are in fact screen shots of our editor. We will integrate our scripts for running μCRL and calculating the security values into this framework and make it publicly available. We are also interested in looking at more flexible ways of matching costed transitions. One possibility might be to use an approximate bisimulation to allow a margin of error when matching transitions. Another direction might be to define a bisimulation that allows a number of transitions in one system to be matched by another number of transitions in the other system. For instance, when modelling time we could allow two sequential five-second transitions to look the same as one ten-second transition.

Acknowledgement

We would like to thank Simona Orzan for useful discussion concerning checking security in μCRL and Stephanie Kemper for discussion concerning the automata model.

References

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

[2] R. Alur, S. La Torre, and G.J. Pappas. Optimal paths in weighted timed automata. *Theoretical Computer Science*, 318(3):297–322, 2004.

[3] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, J. Romijn, and F.W. Vaandrager. Minimum-cost reachability for priced timed automata. In *Proc. 4th Workshop on Hybrid Systems: Computation and Control*, volume 2034 of *LNCS*, pages 147–161. Springer, 2001.

[4] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, 1997.

[5] S. C. C. Blom, W. J. Fokkink, J. F. Groote, I. van Langevelde, B. Lisser, and J. C. van de Pol. μCRL: A toolset for analysing algebraic specifications. In *Proc. 13th Conference on Computer Aided Verification*, volume 2102 of *LNCS*, pages 250–254. Springer, 2001.

[6] S. C. C. Blom and S. Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. *Software Tools for Technology Transfer*, 7(1):74–86, 2005.

[7] S. C. C. Blom and S. Orzan. Distributed state space minimization. *Software Tools for Technology Transfer*, 7(3):280–291, 2005.

[8] P. Bouyer, T. Brihaye, and N. Markey. Improved undecidability results on weighted timed automata. *Information Processing Letters*, 98(5):188–194, 2006.

[9] I. Cervesato. Towards a Notion of Quantitative Security Analysis. In *Proc. 1st Workshop on Quality of Protection*, pages 12–26, 2005.

[10] T. Chothia and J. Kleijn. Q-automata: Modelling the resource usage of concurrent components. In *Proc. 5th International Workshop on Foundations of Coordination Languages and Software Architectures*, 2006.

[11] T. Chothia, S. Orzan, J. Pang, and M. Torabi Dashti. A framework for automatically checking anonymity with μCRL. In *Proc. 2nd Symposium on Trustworthy Global Computing*, LNCS. Springer, 2006.

[12] V. Cortier, M. Rusinowitch, and E. Zalinescu. Relating two standard notions of secrecy. In *Proc. 20th Workshop on Computer Science Logic*, volume 4207 of *LNCS*, pages 303–318. Springer, 2006.

[13] R. De Nicola, G. Ferrari, U. Montanari, R. Pugliese, and E. Tuosto. A process calculus for QoS-aware applications. In *Pro. 7th Conference on Coordination Models and Languages*, volume 3454 of *LNCS*, pages 33–48. Springer, 2005.

[14] S. Eilenberg. *Automata, Languages, and Machines.* Academic Press, 1976.

[15] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proc. 16th Annual International Cryptology Conference*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.

[16] F. Laroussinie and P. Schnoebelen. The state explosion problem from trace to bisimulation equivalence. In *Proc. 3rd Conference Foundations of Software Science and Computation Structures*, volume 1784 of *LNCS*, pages 192–207. Springer, 2000.

[17] C. Meadows. A cost-based framework for analysis of denial of service networks. *Journal of Computer Security*, 9(1/2):143–164, 2001.

[18] T. S. Messerges, E. A. Dabbish, and R. H. Sloan. Examining smart-card security under the threat of power analysis attacks. *IEEE Transactions on Computers*, 51(5):541–552, 2002.

[19] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[20] R. Milner. *Communication and concurrency*. Prentice-Hall, 1989.

[21] N. R. Potlapally, A. Raghunathan, S. Ravi, N. K. Jha, and R. B. Lee. Satisfiability-based framework for enabling side-channel attacks on cryptographic software. In *Proc. 8th Conference on Design, Automation and Test in Europe*, pages 18–23. EDAA, 2006.

[22] M. P. Schützenberger. On the definition of a family of automata. *Information and Control*, 4(2-3):245–270, 1961.