Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 175 (2007) 153–167

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Q-Automata: Modelling the Resource Usage of Concurrent Components[1]

Tom Chothia[2]

CWI, Kruislaan 413, Amsterdam, The Netherlands

Jetty Kleijn[3]

LIACS, Leiden University, P.O.Box 9512, NL-2300 RA Leiden, The Netherlands

Abstract

Q–automata are introduced to model quality aspects of component-based software. We propose Q-algebras as a general framework that allows us to combine and choose between quality values. Such values are added to the transitions of automata, which represent components or channels. These automata can be composed by a product construction yielding a more complex Q-automaton labelled with the combined costs of its components. Thus we establish compositionality of quality of service based on an algebra of quality attributes associated with processes represented by automata.

Keywords: Q-Automata, component-based systems, concurrency, quality of service, compositionality

1 Introduction

This paper introduces Q-automata, which are designed to model trust and quality aspects of component-based software. Quality of Service (QoS) aspects concern non-functional properties such as availability, response time, memory usage, etc. Following work on constraint semirings [10,15], 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

[1] Research supported by the ITEA project Trust4All

[2] Email: `Tom.Chothia@cwi.nl`

[3] Email: `kleijn@liacs.nl`

doi:10.1016/j.entcs.2007.03.009

of components and their composition. There is also a large amount of theoretical and implementational work on using automata for representing components in distributed and reactive systems, which may be of use.

Components will be represented by automata, in a similar way to other work [7,14,17,20,21], but the transitions of our automata will have an additional cost label to indicate the impact of taking that transition on the quality attributes of the system. The resulting Q-automata can be composed using a product construction, leading to a new (higher-level) Q-automaton. 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 perform their actions simultaneously without having to synchronise (e.g., in a communication). This is because, for multi-threaded programs, 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 (or "cost") 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.

Semirings have been proposed as a framework for composing and relating QoS parameters [10,15,19]. Assuming a suitable level of abstraction for managing QoS constraints and a metric for the actual QoS values, constraint semirings provide an algebraic structure with two operations, one to select among values and the other to combine values into a new QoS value. Thus compositionality of QoS values is guaranteed in this approach. We extend constraint semirings to Q-algebras, which have one more operator to combine QoS values. In this way, it becomes possible both to combine costs when they occur sequentially and also when they occur concurrently. Moreover, the costs of such different realisations can be compared within the algebra. In general QoS values are tuples with each component representing a particular aspect: the entries can be of different kinds (numerical to indicate latency, access rights of a service, memory usage, etc.) and, as is the case for constraint semirings, a finite number of Q-algebras may be combined leading to tuples of values, since the product of Q-algebras is again a Q-algebra.

We consider automata with QoS values added as additional labels to the individual transitions indicating their use of resources when executed. A product construction is defined to combine the resulting Q-automata into more complex Q-automata. The product can perform any combination of actions of the original automata simultaneously and it can synchronise matching input and output actions to become an internal action. This most general product accommodates many different styles of communication between components; to enforce one particular style of communication, such as requiring a single end point for each communication channel, a restriction operator may be applied to remove certain transitions.

The components' transitions are combined into new transitions the costs of which are computed from the costs of their constituent transitions. The algebraic structure of the costs domain makes it possible to compare costs, to compute the cost of a transition path (a computation) in a Q-automaton, and to compose the costs of multiple transitions taken concurrently in a composite automaton.

The contributions of this paper are:

- a new model of composable automata, which includes a concept of concurrent transitions for modelling concurrent, communicating components,
- an extended cost algebra to compute different combinations of costs,
- combining the automata and the algebra, leading to the framework of Q-automata,
- and showing how both ordinary components and different types of communication channels can be modelled in this framework.

Weighted automata have a simple weight or cost on each transition and have been extensively studied since the early days of computer science [16,26]. These automata have been shown to have practical applications, an example of which is the work of Mohri et al. on speech processing [23,24]. Buchholz and Kemper [12] describe a method of model checking the class of these automata that only use positive weights. The work we present here 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]. Probability values also combine well with automata, for example Segala [27] adds simple probabilities to transitions and, more recently, Baier and Wolf look at Markov style probabilistic time delays as labels for constraint automata [6]. We hope that at least some of these costs can be subsumed into our cost algebra framework.

Priced or weighted timed automata [2,9] 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 [11].

The automata model we propose is similar to team automata [7,8,17] and related models like I/O automata [20,21] and interface automata [14]. A number of systems attempt to model quantitative aspects of computation by adding semiring costs to process calculi, [5,15,25], these do not distinguish between the concurrent and sequential compositions of costs.

This paper is intended to be a first presentation of our notions and ideas and so, also due to lack of space, it is relatively informal with only a preliminary sketch of initial results. In the next section we introduce our cost algebra and then in Section 3 we define our automata. Restriction and channel communication are discussed in Section 4 where it is shown how the synchronous style of communication between automata can be used to model various kinds of channels. Finally in the concluding Section 5 we briefly discuss how, as future work, the possibly infinite maximum cost of an automaton can be calculated in finite time and how the maximum cost of the product of two automata is limited by their individual costs.

2 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. [15]. 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 the set of all subsets of users

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 $R = (C, \oplus, \otimes, \oplus\!\!\!\!\!\mid\;, \mathbf{0}, \mathbf{1})$ such that $R_\otimes = (C, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ and $R_{\oplus\!\!\!\!\mid} = (C, \oplus, \oplus\!\!\!\!\!\mid\;, \mathbf{0}, \mathbf{1})$ are c-semirings. C is called *the domain of R*.

The $\obar$ operator is used to combine two values concurrently, $c_1 \obar c_2$ 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: $(\mathbb{N} \cup \{\infty\}, \mathtt{min}, \mathtt{max}, +, \infty, 0)$

The product of two Q-algebras is defined component-wise:

Definition 2.3 Given two Q-algebras $R_1 = (C_1, \oplus_1, \otimes_1, \obar_1, \mathbf{0}_1, \mathbf{1}_1)$ and $R_2 = (C_2, \oplus_2, \otimes_2, \obar_2, \mathbf{0}_2, \mathbf{1}_2)$, their product $R = (C, \oplus, \otimes, \obar, \mathbf{0}, \mathbf{1})$ is the Q-algebra defined by
— $C = C_1 \times C_2$,
— $(c_1, c_2) \oplus (c_1', c_2') = (c_1 \oplus_1 c_1', c_2 \oplus_2 c_2')$,
— $(c_1, c_2) \otimes (c_1', c_2') = (c_1 \otimes_1 c_1', c_2 \otimes_2 c_2')$,
— $(c_1, c_2) \obar (c_1', c_2') = (c_1 \obar_1 c_1', c_2 \obar_2 c_2')$,
— $\mathbf{0} = (\mathbf{0}_1, \mathbf{0}_2)$,
— $\mathbf{1} = (\mathbf{1}_1, \mathbf{1}_2)$.

It is easy to see that the product of two Q-algebras is indeed a Q-algebra. Sometimes elements of a product in two different Q-algebras represent the same resource. When we take the product of these algebras we do not want to replicate these entries, but rather combine them. This can be achieved by using a Q-algebra with members that are tuples of labelled elements. The following definitions add labels to the algebra, which can be used for comparison when the product of two (labelled) Q-algebras is calculated. These definitions allow us to write values such as: (power:2, cpu:10) and (power:7, errors:1), rather than (2,10) and (7,1) and for their product to be the (power:9, cpu:10, errors:1) rather than (2,10,7,1). These products are only defined if the operations on identically labelled algebras are the same.

Definition 2.4 Let $n \geq 1$ and let for each $1 \leq i \leq n$, $R_i = (C_i, \oplus_i, \otimes_i, \obar_i, \mathbf{0}_i, \mathbf{1}_i)$ be a Q-algebra. We associate a distinct *label* l_i with each R_i ($l_i \neq l_j$ if $i \neq j$). Then $R = (C, \oplus, \otimes, \obar, \mathbf{0}, \mathbf{1})$ is a *labelled Q-algebra (over* $(l_1 : R_1), \ldots, (l_n : R_n)$*)* if
— $C = (\{l_1\} \times C_1) \times \ldots \times (\{l_n\} \times C_n)$; thus each element $c \in C$ is a labelled tuple of the form $c = (l_1 : c_1, \ldots, l_n : c_n)$ with $c_i \in C_i$ for all $1 \leq i \leq n$,
— $\mathbf{0} = (l_1 : \mathbf{0}_1, \ldots, l_n : \mathbf{0}_n)$,
— $\mathbf{1} = (l_1 : \mathbf{1}_1, \ldots, l_n : \mathbf{1}_n)$,
— $(l_1 : c_1, \ldots, l_n : c_n) \oplus (l_1 : c_1', \ldots, l_n : c_n') = (l_1 : (c_1 \oplus_1 c_1'), \ldots, l_n : (c_n \oplus_n c_n'))$,
— $(l_1 : c_1, \ldots, l_n : c_n) \otimes (l_1 : c_1', \ldots, l_n : c_n') = (l_1 : (c_1 \otimes_1 c_1'), \ldots, l_n : (c_n \otimes_n c_n'))$,
— $(l_1 : c_1, \ldots, l_n : c_n) \obar (l_1 : c_1', \ldots, l_n : c_n') = (l_1 : (c_1 \obar_1 c_1'), \ldots, l_n : (c_n \obar_n c_n'))$.

A labelled Q-algebra is also a Q-algebra. Given a labelled Q-algebra R as specified above, its set of labels $\{l_1, \ldots, l_n\}$ is denoted by $\mathtt{labels}(R)$. We use the notation $\mathtt{proj}_l$ where $l \in \mathtt{labels}(R)$, to extract from R the Q-algebra identified by the label l, e.g. $\mathtt{proj}_{l_i}(R) = R_i$. Note that the ordering is arbitrary; up to the ordering of the underlying Q-algebras and their labels, the labelled Q-algebra as just defined is unique. In the sequel, we will therefore identify any two n-dimensional labelled Q-algebras R and $\tilde{R}$ whenever $\texttt{labels}(R) = \texttt{labels}(\tilde{R})$ and $\texttt{proj}_l(R) = \texttt{proj}_l(\tilde{R})$ for all labels $l \in \texttt{labels}(R)$.

We say that two labelled Q-algebras R and $\tilde{R}$ are *consistent* whenever $\texttt{proj}_l(R) = \texttt{proj}_l(\tilde{R})$ for every common label $l \in \texttt{labels}(R) \cap \texttt{labels}(\tilde{R})$.

Definition 2.5 Let R and $\tilde{R}$ be two consistent, labelled Q-algebras. Then the *labelled product of R and $\tilde{R}$*, denoted by $R \bowtie \tilde{R}$, is the labelled Q-algebra with $\texttt{labels}(R) \cup \texttt{labels}(\tilde{R})$ as its set of labels and defined by $\texttt{proj}_l(R \bowtie \tilde{R}) = \texttt{proj}_l(R)$ if $l \in \texttt{labels}(R)$ and $\texttt{proj}_l(R \bowtie \tilde{R}) = \texttt{proj}_l(\tilde{R})$ if $l \in \texttt{labels}(\tilde{R})$.

Observe that $R \bowtie \tilde{R}$ as above is a labelled Q-algebra. Moreover, as desired, when taking the product of two consistent, labelled Q-algebras, underlying Q-algebras with the same label are not replicated. Finally, it should be noted that the ordinary product of (unlabelled) QoS algebras can be viewed as a special case of the labelled product, namely by giving each algebra its own label. In the rest of this paper we will be dealing only with labelled Q-algebras, even when not referring explicitly to labels.

3 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 $(m_1 + m_2)(x) = m_1(x) + m_2(x)$ for all $x \in X$.

Definition 3.1 A *Q-automaton* is a structure $P = \langle S, t, A, R, T \rangle$ where:
— S is a 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 $T \subseteq S \times \mathbb{M}(Act) \times C \times S$ 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 set $Act = A^O \cup A^I \cup A^\tau$.

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

Definition 3.2 Let P be a Q-automaton specified as in Definition 3.1. 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 3.3 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 3.2. 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 contrast with the traditional synchronous product as applied in the composition of I/O automata and interface automata and its generalisation in team automata, a product of Q-automata is not based on combined executions of single actions (synchronisation) but rather allows simultaneous occurrence of multiple actions. In addition, similar to the communication set-up of CCS [22] or CSP [18], pairs of input and output actions $a?$ and $a!$ may synchronise (communicate) and together become the internal action a. A product automaton has all possible combinations of its components' transitions. Thus, when two components in a certain combination of states can perform matching input and output actions, their product will have from this combined state a communicating transition, but also a transition with those two actions separate and non-communicating (hence still available for communication with other components). Moreover, transitions in which only one of the components is active (either with an input or with an output action) will also be present in the product. For each new transition obtained as a combination of components' transitions, its cost is computed from the costs of its constituents using the $\oplus$ operator of the product algebra.

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 3.4 Let A be a set of action names, Act be its associated set of actions as defined in Definition 3.1 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 $m'_1(a?) = m_1(a?) - 1$,
— $m'_2(a!) = m_2(a!) - 1$,

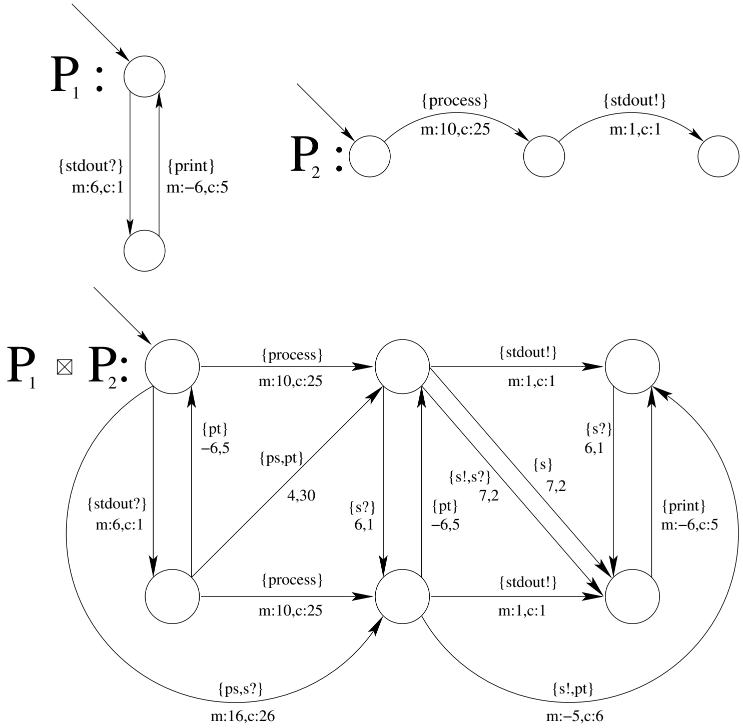

Fig. 1. Simple Automata and Their Product

— 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.

Definition 3.5 Let $P_1 = \langle S_1, t_1, A_1, R_1, T_1\rangle$ and $P_2 = \langle S_2, t_2, A_2, R_2, T_2\rangle$ be two Q-automata such that R_1 and R_2 are consistent. 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
— $S = S_1 \times S_2$,
— $t = (t_1, t_2)$,
— $A = A_1 \cup A_2$,
— $R = R_1 \bowtie R_2$,
— $T = T_1^{new} \cup T_2^{new} \cup T^{joint}$ 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 $c = c_1 \oplus c_2\}$.

As an example of Q-automata and their products we give a simple system in

Figure 1. The automaton P_1 listens on the channel *stdout* and then prints to the screen (an internal action). Automaton P_2 does some internal processing and then sends a message over the *stdout* channel. Each of these actions assigns a certain amount of memory and requires a certain amount of the CPU. The P_1 automaton assigns the memory it needs when it receives a request and frees this memory while printing.

The product of P_1 and P_2 is also displayed in Figure 1, to make this more readable we drop the labels and shorten the action names in the centre of the figure. The automaton $P_1 \boxtimes P_2$ can still receive messages from other automata on the channel *stdout*. This is just as it should be; all components should be able to print to the screen, not just the first component to be added. The product automaton contains concurrent actions, we see that if P_1 receives a call from some third party then $P_1 \boxtimes P_2$ might process and print at the same time. This is of particular interest because it produces the most CPU expensive transition of the whole system and so might make a good test case when testing an implementation of these automata.

We also note that even if automaton P_1 sends a message on channel *stdout* at the same time that P_2 receives a message on *stdout*, it does not automatically mean that P_1 and P_2 have communicated. Indeed, P_1 may receive a message on channel *stdout* from some other component while P_2 sends a message to some other component over the same channel. This is the difference between the $\{stdout\}$ and the $\{stdout?, stdout!\}$ action sets and it allows us to prove that constructing products of Q-automata is associative.

4 Restriction and Communication via Channels

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 4.1 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 $T'_P = \{(s, m, c, s') : (s, m, c, s') \in T_P \text{ and } m(\alpha) = 0\}$. For a set of actions $X = \{\alpha_1, \alpha_2, ..., \alpha_n\}$, we define $P \backslash X$ as $(..(P \backslash \alpha_1) \backslash \alpha_2) ... \backslash \alpha_n)$.

Examples of restrictions are given in Figure 2 (unreachable states are not drawn). The first product automaton is restricted with respect to the *stdout*! output, to model the situation in which only P_1 receives on the *stdout* channel. Thus P_2's output cannot be received by any other component. The second example restricts with respect to both input and output on *stdout* and demonstrates how the "closed" product $P_1 \boxtimes P_2$ would run on its own. This results in a small automaton that may be more useful for testing purposes.

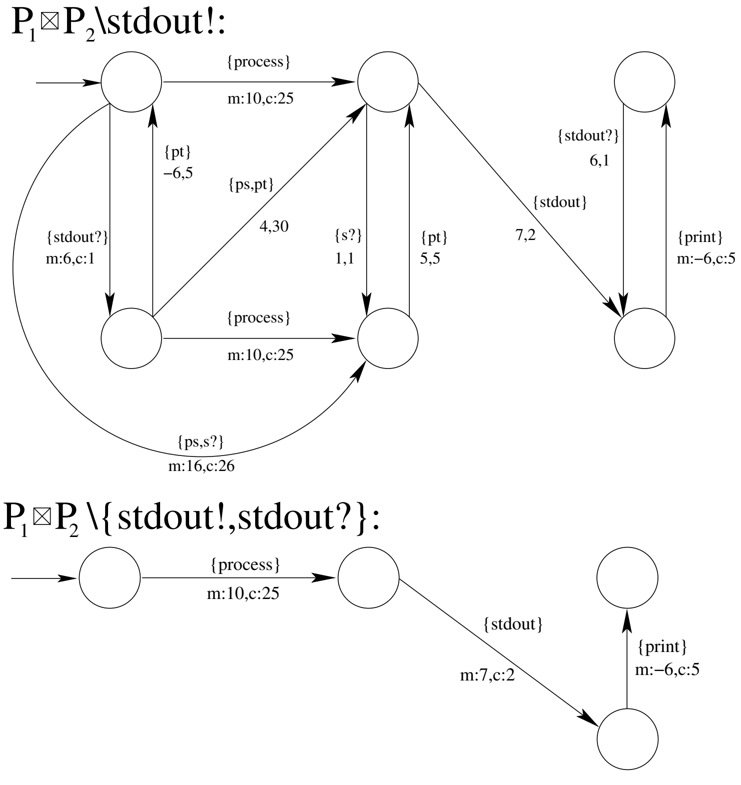

Fig. 2. Restricted Products

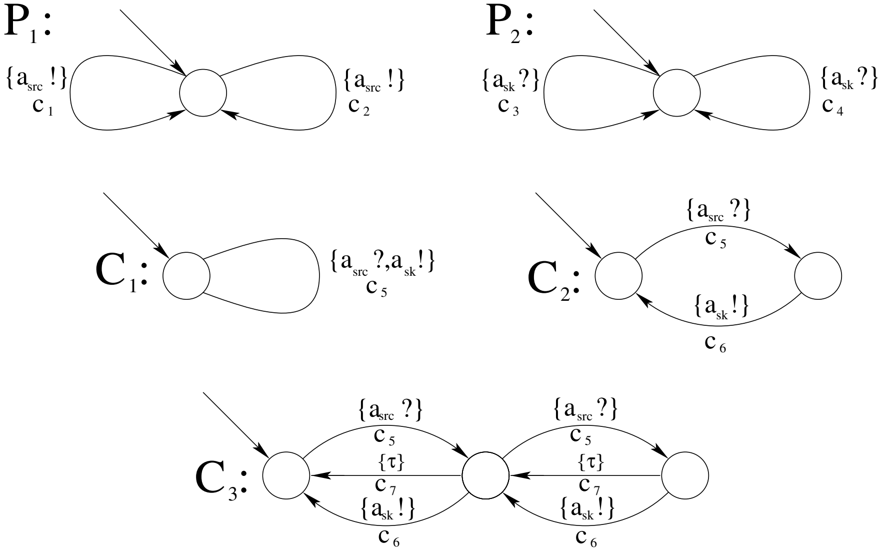

Fig. 3. Automata as Channels

Channel communication

The product construction for automata enforces synchronous communication: an input and output can only communicate and become an internal action if both happen at the same time. Moreover only one input may synchronise with only one output. To model a wider range of communication styles including asynchronous, lossy and multicasting, we explicitly model channels as automata (in a similar way as for constraint automata [3]).

Component automata will write to the "source" of a channel and read from its "sink"; if two components want to communicate on a channel a, one of them outputs on the source of a with the action $a_{src}!$ and the other inputs on the sink of a with the action $a_{sk}?$. Here a_{src} and a_{sk} are different names, associated only by a naming convention, so the product operation does not allow them to synchronise with each other. For a communication to take place the components must use another automaton that represents the channel running between them. Such channel automaton will listen on the source of the channel it represents and send on its sink: $a_{src}?$ and $a_{sk}!$. The two components can then communicate via this channel. Most channel types have one source end and one sink end, but some, such as the synchronous drain which may be used to synchronise two actions, have two source ends or two sink ends and multicast channels may have an arbitrary number of sink ends.

Channel and component automata are illustrated in Figure 3. The component automaton P_1 repeatedly sends on the source of the channel a, with cost c_1 or c_2. The other component automaton P_2 may receive on the sink of the channel a with cost c_3 or in another way with cost c_4. The automata C_1, C_2 and C_3 define three types of channels that the components might use to communicate.

The channel automaton C_1 has a single transition that receives on the source of a channel a while at the same time sends on its sink, i.e., this automaton defines a synchronous communication channel. The channel automaton C_2 first receives on the source of a and then, some time later, sends on its sink. So this defines an asynchronous channel with a buffer large enough to hold one message. Finally the channel automaton C_3 defines a lossy, two-buffer, asynchronous channel. The possibility of losing a message is shown by the τ actions that allow a message that has been received on the source of the channel not to be forwarded to its sink.

As an example of one of these channels being used, Figure 4 shows the closed version of the product $P_1 \boxtimes C_1 \boxtimes P_2$. This represents the two components, P_1 and P_2 using the channel C_1. The double internal action $\{a_{src}, a_{sk}\}$ represents the communication via a. There are four possible combinations depending on how the components P_1 and P_2 perform their communications. These different possibilities yield different costs. The (open) product $P_1 \boxtimes C_1 \boxtimes P_2$ would have a looping transition with the label $\{a_{src}?, a_{sk}!\}$ with costs c_5. This product automaton could also perform the actions $a_{src}!$ and $a_{sk}?$, so making it possible for the components to communicate using some other channel that might be added later. Furthermore, independent actions might happen concurrently adding more transitions to the product.

In general, the multitude of transitions defined for a product automaton, is necessary in order to avoid an "a priori" ruling out of certain kinds of communication. However a restriction to just one kind of communication is possible by automatically

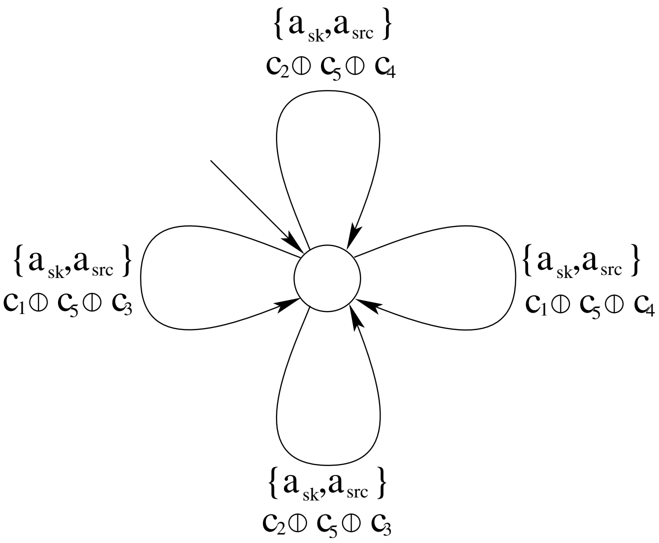
$\{a_{sk}, a_{src}\}$
$c_2 \oplus c_5 \oplus c_4$
$\{a_{sk}, a_{src}\}$
$c_1 \oplus c_5 \oplus c_3$
$\{a_{sk}, a_{src}\}$
$c_1 \oplus c_5 \oplus c_4$
$\{a_{sk}, a_{src}\}$
$c_2 \oplus c_5 \oplus c_3$

Fig. 4. $P_1 \boxtimes C_1 \boxtimes P_2 \backslash (inputs \cup outputs)$

applying a restriction to the product.

As an example we may consider a system in which every channel has unique end points, i.e., only one component may read from a channel's sink and only one component may write to a channel's source. As we know that inputs and outputs are unique we may close an input and output action once it has had a chance to synchronise. This is done by the following refinement of the product definition:

Definition 4.2 Given Q-automata P_1 and P_2, their *point-to-point product* is the Q-automaton

$$\begin{aligned} P_1 \boxtimes_p P_2 = P_1 \boxtimes P_2 \backslash (&\{a! : \exists (s,m,c,t) \in (T_{P_1} \cup T_{P_2}) \text{ such that } m(a?) \geq 1\} \\ &\cup \{a? : \exists (s,m,c,t) \in (T_{P_1} \cup T_{P_2}) \text{ such that } m(a!) \geq 1\} \end{aligned}$$

When modelling such systems, using this definition of product automatically removes many inputs and outputs that we know will not synchronise to become communications and so makes our state space smaller and easier to model check. The restriction operator can also be used to define similar restrictive product operators for other types of communication.

Mobile channels

Mobile channels, such as those implemented in the MoCha mobile channel communication package [4], allow components to be linked by channels whose end points can be moved between locations. In this setting components have a number of unique ports on which they may send and receive messages and channels link these ports to allow communication to take place.

We can model these systems using automata to model components and channels, as described in the last subsection, and we can model the mobility aspects of these channels by allowing the channels to move between states that link different end

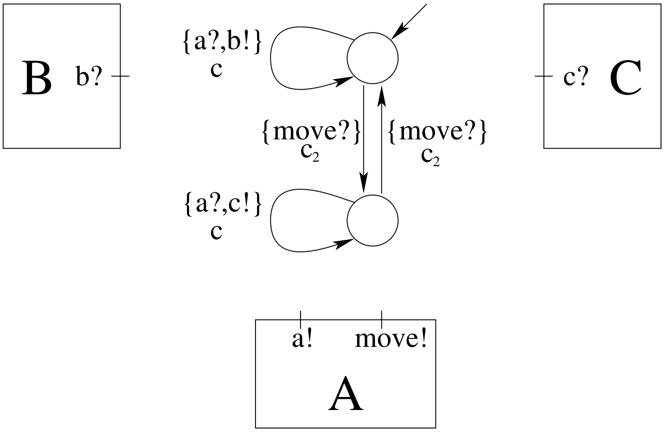

Fig. 5. A Mobile Channel that Moves Between B and C

points. For example, in Figure 5 we have components B and C that listen for input on ports b and c and another component A that sends messages on port a and may also send a *move* signal. Each of these components would be represented by a suitable automaton, for the sake of simplicity we abstract away their details in the figure.

The automaton in the centre of the figure represents a channel that starts off by linking the output port of A to the input port of B, using the transition with the label $\{a?, b!\}$. This performs the same function as the synchronous channel described in the last section, however this channel may also receive a *move* signal from component A, at which point it will change to linking the output on a to the input on c, so moving the channel end from component B to component C. In this way we can define a wide range of channels that can move their end points between any number of predefined ports.

5 Conclusion and Further work

We have presented an automata model in which actions are labelled with trust or quality values. We have defined Q-algebras as a general model of these values and we have showed how these automata and the values on the transitions could be meaningfully combined.

Q-automata were conceived as 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. We hope that the automata model presented here will be suitable for model checking these components and that ultimately we will be able to automatically generate test data.

We are particularly interested in the maximum possible cost of an automaton and the computations leading to that cost. In general, there may be infinitely many computations starting at a given state and so an infinite number of costs to apply the $\oplus$ to in order to find the maximum. However, in the case of a finite set of state, an infinite number of computations can only be generated by looping. A single loop may either add a fixed cost to a computation (leading to a potentially infinite cost), or may set a new, level cost, or might have no effect at all. Therefore a single traverse round a loop is enough to tell you what the effect of an infinite number of traversals of that loop will be. This means that we can calculate the maximum cost in finite time, even if that cost is infinite.

We would also like to develop methods of predicting the behaviour of the product from its individual parts. For instance, we expect that the least upper bound of the concurrent and sequential combination of the maximum cost of two automata is an upper bound on the maximum cost of their product, i.e., if P_1 and P_2 have the maximum costs c_1 and c_2 then the maximum cost of $P_1 \boxtimes P_2$ is less than or equal to $(c_1 \otimes c_2) \oplus (c_1 \oslash c_2)$.

Maude [13] is a high level language based on equational and rewriting logic. It provides an easy framework in which to implement Q-automata and model check a few basic properties[4]. We define a Maude module for each kind of cost and for each automaton. The cost modules include equational definitions of $\oplus, \otimes$ and $\oslash$. We define automata in Maude by using constructors to give the states and then defining rewrite rules between these states using the *rl* command. We may then check cost based properties using Maude's *search* command that performs a breadth first search of all the costed states. For instance, we may check that the memory allocated by a given program never goes above 100k. We will continue this development work with the aim of making our prototype more powerful and user friendly.

Acknowledgements

We are indebted to our local colleagues in this project for many fruitful discussions and in particular to Farhad Arbab, Frank de Boer, Marcello Bonsangue, Andries Stam, and Frank Atanassow. We are also grateful to the referees for their constructive criticism.

References

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

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

[3] F. Arbab, C. Baier, J. J. M. M. Rutten, and M. Sirjani. Modeling component connectors in Reo by constraint automata: (extended abstract). *Electr. Notes Theor. Comput. Sci.*, 97:25–46, 2004.

[4] F. Arbab, F. de Boer, J. Guillen Scholten, and M. Bonsangue. Mocha: A middleware based on mobile channels. In *COMPSAC*, pages 667–673, 2002.

[5] B. Aziz. A semiring-based quantitative analysis of mobile systems. *Electr. Notes Theor. Comput. Sci.*, 157:3–21, 2006.

[6] C. Baier and V. Wolf. Stochastic reasoning about channel-based component connectors. In *Coordination 2006*, volume 4038 of *Lectures Notes in Computer Science*, pages 1–15, 2006.

[4] The Maude code for the example automata described in this paper are available on-line at: http://homepages.cwi.nl/~chothia/QAutoMaude

[7] M.H. ter Beek, C.A. Ellis, H.C.M. Kleijn, and G. Rozenberg. Synchronizations in team automata for groupware systems. *CSCW - The Journal of Collaborative Computing*, 12(1):21–69, 2003.

[8] M.H. ter Beek and H.C.M. Kleijn. Team automata satisfying compositionality. In *Formal Methods'03*, volume 2805 of *Lecture Notes in Computer Science*, pages 381–400, 2003.

[9] G. Behrmann, A. Fehnker, T. Hune, K. Guldstrand Larsen, P. Pettersson, J. Romijn, and F.W. Vaandrager. Minimum-cost reachability for priced timed automata. In *4th International Workshop on Hybrid Systems*, volume 2034 of *Lecture Notes in Computer Science*, pages 147–161, 2001.

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

[11] P. Bouyer, T. Brihaye, and N. Markey. Improved undecidability results on weighted timed automata. *Inf. Process. Lett.*, 98(5):188–194, 2006.

[12] P. Buchholz and P. Kemper. Model checking for a class of weighted automata. *CoRR*, cs.LO/0304021, 2003.

[13] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The maude 2.0 system. In *Rewriting Techniques and Applications 2003*, number 2706 in Lecture Notes in Computer Science, pages 76–87, 2003.

[14] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.

[15] R. De Nicola, G. Ferrari, U. Montanari, R. Pugliese, and E. Tuosto. A process calculus for QoS-aware applications. In *Coordination 2005*, volume 3454 of *Lectures Notes in Computer Science*, pages 33–48, 2005.

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

[17] C.A. Ellis. Team automata for groupware systems. In *GROUP'97*, pages 415–424. ACM Press, 1997.

[18] C.A.R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.

[19] W. Kuich and A. Salomaa. *Semirings, Automata, Languages.* Springer-Verlag, 1986.

[20] N.A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[21] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. In *CWI-Quarterly*, volume 2(3), pages 219–246, 1989.

[22] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1982.

[23] M. Mohri, F. Pereira, and M. Riley. Weighted automata in text and speech processing. In *ECAI'96 Workshop on Extended Finite State Models of Language*, pages 46–50, 1996.

[24] M. Mohri, F. Pereira, and M. Riley. The design principles of a weighted finite-state transducer library. *Theor. Comput. Sci.*, 231(1):17–32, 2000.

[25] V. Natarajan and R. Cleaveland. An algebraic theory of process efficiency. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, page 63. IEEE Computer Society, 1996.

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

[27] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, MIT, Dept. of Elec. Eng. and Comp. Sci., 1995.