Q-Automata: Modelling the Resource Usage of Concurrent Components (pre-print version) 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
We introduce Q-automata 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.

Key words: 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 [6,17], 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

1 Research supported by the ITEA project Trust4All
2 Email: Tom.Chothia@cwi.nl
3 Email: kleijn@liacs.nl

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
to the analysis of the behaviour 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 [8,10,13,14,20], 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 to compose and relate QoS parameters [6,12,17]. 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.

Model Checking for a Class of Weighted Automata [21].
Weighted Timed Automata [?] Price timed automata [?]

Although they are in general undecidable [?]

Weighted automata have a simple weight or cost on each transition and have been extensively studied since the early days of computer science [9,18]. Timed automata models label transitions with costs representing the time they take [1]. Probability values also combine well with automata, for example Segala [19] adds simple probabilities to transitions and, more recently, Baier and Wolf look at Markov style probabilistic time delays as labels for constraint automata [5]. We hope that at least some of these costs can be subsumed into our cost algebra framework. The automata model itself is similar to team automata [10,20,21] and related models like I/O automata [14,13] and Interface automata [8]. A number of systems attempt to model quantitative aspects of computation by adding semiring costs to process calculi, [4,16,17], 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 possible infinite maximum cost of an automata can be calculated in finite time and how the maximum cost of the product of two automata
is limited by their individual costs. We also mention an implementation of our automata in language Maude [7].

2 Q-Algebra

To compute and analyse QoS values in a standard way we develop a general framework based on the approach of De Nicola et al. [17]. First we recall the concept of a constraint semiring:

**Definition 2.1** A *constraint semiring* is a structure \( R = (C, \oplus, \otimes, 0, 1) \) where \( C \) is a set, \( 0, 1 \in C \), and \( \oplus \) and \( \otimes \) are binary operations on \( C \) such that:

- \( \oplus \) is commutative, associative, idempotent and has identity 0
- \( \otimes \) is associative and has identity 1
- \( \otimes \) distributes over \( \oplus \) and has 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 \( c_2 \). In the case of a total order \( c_1 \oplus c_2 \) will equal whichever value is “least”. The 0 element, as the identity of \( \oplus \), is the least possible cost value and the 1 element, as the identity of \( \otimes \), is the neutral cost value.

A few examples:
- (shortest) time: \( ([\mathbb{R}_+ \cup \{\infty\}], \min, +, \infty, 0) \)
- bandwidth: \( (\mathbb{N} \cup \{\infty\}, \min, \max, \infty, 0) \)
- data encrypted: \( (\{\text{true, false}\}, \lor, \land, \text{false, true}) \)
- 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 as when they are 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 \( R = (C, \oplus, \otimes, \oplus, 0, 1) \) such that
$R_{\ominus} = (C, \ominus, \otimes, 0, 1)$ and $R_{\oplus} = (C, \oplus, \otimes, 0, 1)$ are c-semirings. $C$ is called the domain of $R$.

The $\oplus$ operator is used to combine two values concurrently, $c_1 \oplus c_2$ 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: $(\mathbb{R}_+ \cup \{\infty\}, \min, +, \max, \infty, 0)$
- bandwidth: $(\mathbb{N} \cup \{\infty\}, \min, \max, +, \infty, 0)$

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

**Definition 2.3** Given two Q-algebras $R_1 = (C_1, \oplus_1, \otimes_1, \ominus_1, 0_1, 1_1)$ and $R_2 = (C_2, \oplus_2, \otimes_2, \ominus_2, 0_2, 1_2)$, their product $R = (C, \oplus, \otimes, 0, 1)$ is the Q-algebra defined by

- $C = C_1 \times C_2$,
- $(c_1, c_2) \oplus (c'_1, c'_2) = (c_1 \oplus_1 c'_1, c_2 \oplus_2 c'_2)$,
- $(c_1, c_2) \otimes (c'_1, c'_2) = (c_1 \otimes_1 c'_1, c_2 \otimes_2 c'_2)$,
- $(c_1, c_2) \ominus (c'_1, c'_2) = (c_1 \ominus_1 c'_1, c_2 \ominus_2 c'_2)$,
- $0 = (0_1, 0_2)$,
- $1 = (1_1, 1_2)$.

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 $1 \leq i \leq n$, $R_i = (C_i, \ominus_i, \otimes_i, \oplus_i, 0_i, 1_i)$ 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 $R = (C, \oplus, \otimes, 0, 1)$ 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 $1 \leq i \leq n$,
- $0 = (l_1 : 0_1, \ldots, l_n : 0_n)$,
- $1 = (l_1 : 1_1, \ldots, l_n : 1_n)$,
- $(l_1 : c_1, \ldots, l_n : c_n) \oplus (l_1 : c'_1, \ldots, l_n : c'_n) = (l_1 : (c_1 \oplus c'_1), \ldots, l_n : (c_n \oplus c'_n))$,
- $(l_1 : c_1, \ldots, l_n : c_n) \otimes (l_1 : c'_1, \ldots, l_n : c'_n) = (l_1 : (c_1 \otimes c'_1), \ldots, l_n : (c_n \otimes c'_n))$,
- $(l_1 : c_1, \ldots, l_n : c_n) \ominus (l_1 : c'_1, \ldots, l_n : c'_n) = (l_1 : (c_1 \ominus c'_1), \ldots, l_n : (c_n \ominus c'_n))$. 

5
Clearly, 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 \( \text{labels}(R) \). We use the notation \( \text{proj}_l \) where \( l \in \text{labels}(R) \), to extract from \( R \) the Q-algebra identified by the label \( l \): \( \text{proj}_l(R) = R_l \). 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 \( \text{labels}(R) = \text{labels}(\tilde{R}) \) and \( \text{proj}_l(R) = \text{proj}_l(\tilde{R}) \) for all labels \( l \in \text{labels}(R) \).

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

**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 \( \text{labels}(R) \cup \text{labels}(\tilde{R}) \) as its set of labels and defined by \( \text{proj}_l(R \bowtie \tilde{R}) = \text{proj}_l(R) \) if \( l \in \text{labels}(R) \) and \( \text{proj}_l(R \bowtie \tilde{R}) = \text{proj}_l(\tilde{R}) \) if \( l \in \text{labels}(\tilde{R}) \).

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 \( \mathbb{M}(X) \).

**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,
- \( R = (C, \oplus, \ominus, \odot, 0, 1) \) is a labelled QoS algebra with domain \( C \) of costs,
- and \( T \subseteq S \times \mathbb{M}(\text{Act}) \times C \times S \) is the set of transitions.

The set of actions of \( P \), written \( \text{Act} \), 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 \( A^* = A \) the set of
internal actions of $P$. The sets $A^O$, $A^I$, and $A^r$ are assumed to be pairwise disjoint. Finally, we set $Act = A^O \cup A^I \cup A^r$.

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 $(s_0, B_1, c_1, s_1), \ldots, (s_{n-1}, B_n, c_n, s_n)$ with $(s_i, B_i, c_i, s_{i+1}) \in T$ 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 $\gamma = (s_0, B_1, c_1, s_1), \ldots, (s_{n-1}, B_n, c_n, s_n)$ be a computation as specified in Definition 3.2. Then the cost of $\gamma$ is $1$, if $n = 0$ and $c_1 \otimes \cdots \otimes c_n$ 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” $\odot$ 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 a single action (synchronisation) but rather allows simultaneous occurrence of multiple actions. In addition, similar to the communication set-up of CCS [15] or CSP [11], 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 $\odot$ operator of the product algebra.

In the formal definition of product automata, we use the auxiliary function $sync$ defined in the following way.

**Definition 3.4** Let $A$ be a set of action names and let $Act$ be its associated
set of actions as defined in Definition 3.1. We first introduce a relation \( \Rightarrow \) over pairs of multisets of actions differing only in the occurrences of communications that took place between these multisets:
Let \( m_1 \) and \( m_2 \) be two multisets over \( \text{Act} \). 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 \( m'_2(a) = m_1(a) - 1 \),
- \( m'_1(b) = m_1(b) \) and \( m'_2(b) = m_2(b) \) for all other actions \( b \in \text{Act} \).

or if \( (m_2, m_1) \Rightarrow (m'_2, m'_1) \) as above.

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

Thus \( \text{sync}(m, m') \) 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 \( \mathcal{Q} \)-automata such that \( R_1 \) and \( R_2 \) are consistent. Then their product, denoted by \( P_1 \boxtimes P_2 \), is the \( \mathcal{Q} \)-automaton defined as \( P_1 \boxtimes P_2 = \langle S, t, A, R, T \rangle \) with:
- \( S = S_1 \times S_2 \) and \( t = (t_1, t_2) \),
- \( A = A_1 \cup A_2 \),
- \( R = R_1 \Join R_2 \),
- \( T = T_1^\text{new} \cup T_2^\text{new} \cup T^\text{joint} \) where:
  - \( T_1^\text{new} = \{ (s, t), B, c, (s', t) : (s, B, c, s') \in T_1 \text{ and } t \in S_2 \} \),
  - \( T_2^\text{new} = \{ (s, t), B, c, (s, t') : s \in S_1 \text{ and } (t, B, c, t') \in T_2 \} \), and
  - \( T^\text{joint} = \{ (s, t), B, c, (s', t') : \exists (s, B_1, c_1, s') \in T_1, (t, B_2, c_2, t') \in T_2 \text{ such that } B \in \text{sync}(B_1, B_2) \text{ and } c = c_1 \cup c_2 \} \).

As an example of \( \mathcal{Q} \)-automata and their products we give a simple system in Figure 1. The automaton \( P_1 \) listens on the channel \( \text{stdout} \) and then prints to the screen (an internal action). Automaton \( P_2 \) does some internal processing and then sends a message over the \( \text{stdout} \) channel. Each of these actions assigns a certain amount of memory and requires a certain percentage 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. The automaton \( P_1 \boxtimes P_2 \) can still receive messages from other automata on the channel \( \text{stdout} \). 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.
Fig. 1. Simple Automata and Their Product

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 automata 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 $P = (S, t, A, R, T)$ be a $Q$-automaton and let $\alpha \in A^O \cup A^I \cup A^r$ be an action of $P$. Then $P \setminus \alpha$, the restriction of $P$ with respect to $\alpha$, is the $Q$-automaton $(S, t, A, R, T')$ with $T'_p = \{(s, m, c, s') : (s, m, c, s') \in T_p$ and $m(\alpha) = 0\}$. For a set of actions $X = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, we define $P \setminus X$ as $(..(P \setminus \alpha_1) \setminus \alpha_2) \ldots \setminus \alpha_n)$. 

9
Examples of restrictions are given in Figure 2 (unreachable states are not drawn). The first product automaton is restricted with respect to `stdout!` output to model that only \( P_1 \) is allowed to receive 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 \( P_1 \otimes P_2 \) would run on its own. This results in a small automaton that may be more useful for testing purposes.

**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 other communication styles including asynchronicity, lossy and multicasting, we explicitly model channels as automata (in a similar way as constraint automata [2]).

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 \( a_{\text{src!}} \) and the other inputs on the sink of \( a \) with the action \( a_{\text{sink?}} \). Here \( a_{\text{src!}} \) and \( a_{\text{sink?}} \) 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 automata will listen on the source of the
channel it represents and send on its sink: $a_{sc}$ 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 com-
ponent automata $P_1$ repeatedly sends on the source of the channel $a$, with
cost $c_1$ or $c_2$. The other component automata $P_2$ 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.

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 of $P_1 \boxtimes C_1 \boxtimes P_2$ would have a looping transition with the la-
bel $\{a_{src}, a_{sk}\}$ 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 com-
unicate using some other channel that might be added later. Furthermore,
independent actions might happen concurrently adding more transitions to
the product.
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 $\tau$ actions that allow a message that has been received on the source of the channel not to be forwarded to its sink.

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 using a compatibility notion for automata and then automatically apply to their product a restriction with respect to certain actions. As an example we take the model of communication used by the MoCha channel communication package [3]. In this system a channel has unique end points, so only one component may read from a channel’s sink and only one component may write to a channel’s source, also there can only be one channel implementation for any channel name. We can formalise these restrictions by saying that a number of automata are point-to-point compatible if no two automata try to input or output on the same name.

If we know that inputs and outputs are unique then we may close an input and output action once it has had a change to synchronise. This is done by the following definition:

**Definition 4.2** Given Q-automata $P_1$ and $P_2$, their point-to-point product is the Q-automaton $P_1 \boxtimes P_2 = P_1 \boxtimes P_2 \setminus (\{a_{src}! : a_{src} ? : a_{sk} ! : a_{sk} ? : a_{sk}! : a_{sk} ! \in A_{P_1} \cup A_{P_2} \} \cup \{a_{src} : a_{src} ! \in A_{P_1} \cup A_{P_2} \} \cup \{a_{src} ? : a_{src} ! \in A_{P_1} \cup A_{P_2} \})$.

When dealing with MoCha, using this definition of product removes many inputs and outputs that we know will not synchronise to become communi-
cation 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.

5 Conclusion and Further work

We have presented an automata model in which actions are labelled with trust or quality values. We defined Q-Algebras as a general model of these values and we 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 modelling these components and that ultimately we will be able to model check our automata and automatically generate test data.

We are particularly interested in the maximum possible cost of an automaton and the computation that produce it. 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, 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 will look for the most efficient algorithm to do this.

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 maximum cost of an automata is an upper bound on their product, i.e., if $P_1$ and $P_2$ have the maximum costs $c_1$ and $c_2$ then the maximum cost of $P_1 \otimes P_2$ is less than or equal to $(c_1 \otimes c_2) \oplus (c_1 \oplus c_2)$.

Maude 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. We define a Maude module for each of our kinds of cost and for each automaton \footnote{The Maude code for the example automata described in this paper are available on-line at: http://homepages.cwi.nl/~chothia/QAutoMaude}. The cost modules include equational definitions of $\oplus$, $\otimes$ and $\oplus$. We define our automata in Maude by using constructors to give the states and then defining rewrite rules between these states using the $\text{rl}$ command. We add a cost to each state, to represent the total cost so far. We may then check cost based properties using Maude's $\text{search}$ 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 make our prototype more powerful and user friendly.

Acknowledgments
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 Staan and Frank Atanassow. We are also grateful to the referees for their constructive criticism.

References


