Implementation of a Compositional Performance Analysis Algorithm for Probabilistic I/O Automata*

Eugene W. Stark, Giridhar Pemmasani

Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794 USA

Abstract. In previous papers, we defined the *probabilistic I/O automata* model for specification and modeling of probabilistic concurrent systems, and we showed how certain performance measures for such systems could be computed compositionally, one component at a time, without the need for explicit construction of the full global state space. In this paper, we report on our experiences in constructing and testing a computer implemention of these compositional analysis algorithms. Our implementation, which is coded in the functional programming language Standard ML, uses exact rational arithmetic to calculate performance measures, and it is also capable of producing symbolic rational function expressions that describe the dependence of performance measures on a system parameter.

1 Introduction

This research is aimed at the specification, modeling, and performance analysis of computational systems that exhibit concurrent, probabilistic, and real-time behavior. We are primarily interested in finite-state models of such systems, and have as our objective the construction of efficient tools that can analyze automatically various performance parameters and correctness properties of such models for realistic systems. Communication protocols and embedded systems are examples of the types of application domains to which we feel our methods could usefully be applied. We have developed a particular model, called "probabilistic I/O automata" (PIOA) for representing such systems [WSS97], and we have devised compositional methods for analyzing these representations for certain properties [SS98]. Because compositional methods can avoid some of the problems of state-space explosion, our compositional methods are potentially applicable to much larger examples than existing, non-compositional techniques.

In this paper, we report on our experiences in constructing and testing a computer implemention of the compositional analysis algorithms presented in [SS98]. Our implementation, which is coded in the Standard ML functional programming language, uses exact rational arithmetic to calculate compositionally,

* Research supported in part by AFOSR Grant F49620-96-1-0087. E-mail addresses: `stark@cs.sunysb.edu`, `giri@cs.sunysb.edu`

one component at a time, performance measures for systems of PIOAs. Examples of the types of performance measures that can be handled with our approach are the probability of the system performing a sequence of actions in a specified, possibly infinite, set, or the expected time it will take for the system to perform such a sequence. Our code is also capable of calculating symbolic rational function expressions that describe the dependence of performance measures on a system parameter. For example, for a communication protocol, we could compute a formula that shows how the expected time for communication varies with the probability of message loss by the communication medium.

We have successfully applied our code to analyze, within several hours of CPU time, systems that would have over 70 million global states, if the global state space were constructed explicitly. Smaller examples, with global state spaces in the hundreds of thousands, run in a few minutes.

2 Probabilistic I/O Automata and Their Behaviors

2.1 Probabilistic I/O Automata

Probabilistic I/O automata (PIOA) [WSS97] are an adaptation of the I/O automata model developed by Nancy Lynch and her students [Lyn96], which they have successfully applied to the hierarchical specification and verification of distributed algorithms. In this section, we recall the basic definitions of PIOAs and related notions from [WSS97]. Following the presentation in [SS98], we give here only enough of the formalism to set a context for discussing our implementation of the analysis algorithms. The reader wishing full details, with proofs and discussion of the intuitions underlying the model, is referred to [WSS97] and [SS98].

A *finite probabilistic I/O automaton* is a tuple $A = (Q, q^{\mathrm{I}}, E, \mu, \rho)$, where

- Q is a finite set of *states*;
- $q^{\mathrm{I}} \in Q$ is a distinguished *start state*;
- E is a finite set of *actions*, partitioned into disjoint sets of *input*, *output*, and *internal* actions, which are denoted by E^{in}, E^{out}, and E^{int}, respectively, with the actions in $E^{\mathrm{loc}} = E^{\mathrm{out}} \cup E^{\mathrm{int}}$ called *locally controlled*;
- $\mu : (Q \times E \times Q) \to [0, 1]$ is the *transition probability* function, which is required to satisfy the following stochasticity conditions:
  1. $\sum_{r \in Q} \mu(q, e, r) = 1$, for all $q \in Q$ and all $e \in E^{\mathrm{in}}$.
  2. For all $q \in Q$, if there exist $e \in E^{\mathrm{loc}}$ and $r \in Q$ such that $\mu(q, e, r) > 0$, then $\sum_{r \in Q} \sum_{e \in E^{\mathrm{loc}}} \mu(q, e, r) = 1$,
- $\rho : Q \to [0, \infty)$ is the *rate function*, which is required to satisfy the following condition: for all $q \in Q$, we have $\rho(q) > 0$ if and only if there exist $e \in E^{\mathrm{loc}}$ and $r \in Q$ such that $\mu(q, e, r) > 0$.

As discussed in [WSS97] and [SS98], the definitions of μ and ρ above reflect the intuition we wish to capture concerning the execution of a system of PIOAs. Upon arrival in a state q, a PIOA chooses randomly the length of time it will spend in that state before executing its next "locally controlled" transition. The random choice is made, independently of other system components, according to an exponential holding time distribution whose mean is the reciprocal $1/\rho(q)$ of the rate $\rho(q)$ associated with that state. When the time to execute the next locally controlled transition arrives, the probabilistic transition relation is consulted to determine the specific action to be performed and the new state. An input action may be applied by the environment at any time. In that case, the probabilistic transition relation governs the choice of the next state, and whatever holding time had been chosen for the previous state is abandoned.

A *finite execution fragment* for a probabilistic I/O automaton A is an alternating sequence σ of states and actions of the form

$$q_0 \xrightarrow{e_0} q_1 \xrightarrow{e_1} \dots \xrightarrow{e_{n-1}} q_n,$$

such that for each k with $0 \leq k < n$, either $e_k \in E$ and $\mu(q_k, e_k, q_{k+1}) > 0$, or else $e_k \notin E$ and $q_{k+1} = q_k$. (Admitting the case $e_k \notin E$ permits us to consider execution fragments for systems containing A also as execution fragments for A, even if these fragments happen to contain actions in which A does not participate.) An execution fragment σ as above is called an *execution* if $q_0 = q^{\mathrm{I}}$ (the distinguished start state). We use the term *trace* to refer to a sequence of actions. The *trace of* σ, denoted $\mathrm{tr}(\sigma)$, is the sequence of actions $e_0 e_1 \dots e_{n-1}$ appearing in σ.

A finite collection $\{A_i : i \in I\}$ of probabilistic I/O automata, where $A_i = (Q_i, q_i^{\mathrm{I}}, E_i, \mu_i, \rho_i)$, is called *compatible* if for all $i, j \in I$ with $i \neq j$, we have $E_i^{\mathrm{out}} \cap E_j^{\mathrm{out}} = \emptyset$ and $E_i^{\mathrm{int}} \cap E_j = \emptyset$. The *composition* of such a collection is a PIOA $(Q, q^{\mathrm{I}}, E, \mu, \rho)$, where Q is the cartesian product $\prod_{i \in I} Q_i$, the initial state q^{I} is the tuple $\langle q_i^{\mathrm{I}} : i \in I \rangle$, and the set E of actions is $\bigcup_{i \in I} E_i$, with $E^{\mathrm{out}} = \bigcup_{i \in I} E_i^{\mathrm{out}}$, $E^{\mathrm{int}} = \bigcup_{i \in I} E_i^{\mathrm{int}}$, and $E^{\mathrm{in}} = E \setminus (E^{\mathrm{out}} \cup E^{\mathrm{int}})$. Suppose $q \in Q$ and $r \in Q$ are $\langle q_i : i \in I \rangle$ and $\langle r_i : i \in I \rangle$, respectively. Then we define $\rho(q) = \sum_{i \in I} \rho_i(q_i)$. Adopting the convention that $\mu_i(q_i, e, r_i) = 1$ if $e \notin E_i$, we define $\mu(q, e, r) = (\rho_j(q_j)/\rho(q)) \prod_{i \in I} \mu_i(q_i, e, r_i)$ if $e \in E_j^{\mathrm{loc}}$ for some $j \in I$, otherwise $\mu(q, e, r) = \prod_{i \in I} \mu_i(q_i, e, r_i)$.

The definitions of μ and ρ above express the intuitive idea that the various component PIOAs are in a race to see which of them will execute the next locally controlled action. This competition will be won by the component that has chosen the smallest holding time in its respective state, and the probability that any given component will win the competition is given by the ratio of the rate for the local state of that component over the sum of the rates for the local states of all of the components. The time the system remains in a particular global state before executing the next locally controlled action is the minimum of the times that each component spends in its respective local state. This time is governed by an exponential distribution, whose rate is the sum of the rates of the distributions for each of the components.

In this paper, we shall generally be concerned with the composition of compatible 2-element sets $\{A, B\}$ of PIOAs, and we use the notation $A \| B$ to denote such a composition.

2.2 Rated Traces, Observables, and Behaviors

Let E be a set of actions. A (finite) *rated trace α over E* consists of an alternating sequence of the form:

$$d_0 \xrightarrow{e_0} d_1 \xrightarrow{e_1} \ldots \xrightarrow{e_{n-1}} d_n,$$

where the d_k are nonnegative real numbers, called the *rates*, and the e_k are actions in E. The sequence $e_0, e_1, \ldots e_{n-1}$ is called the *trace of* α, and we denote it by $\mathrm{tr}(\alpha)$. A rated trace is an abstraction of an execution fragment, in which specific states have been replaced by their rates. There is a precise sense in which all execution fragments having the same rated trace, for a PIOA A, are probabilistically indistinguishable from each other under PIOA composition.

We use $\mathrm{RTraces}(E)$ to denote the set of all rated traces over E. We also use the notation $(d)_E$, or just (d), when E is clear from the context, to denote the *empty rated trace* in $\mathrm{RTraces}(E)$, consisting of the single rate d and no actions.

An *observable* over a set of actions E is a mapping $\Phi : \mathrm{RTraces}(E) \to \mathcal{R}$, where $\mathcal{R}$ denotes the set of real numbers. If A is a PIOA with action set E_A and E is any other set of actions, then the *E-behavior* of A is the *transformation of observables*:

$$\mathcal{B}_E^A : (\mathrm{RTraces}(E \cup E_A) \to \mathcal{R}) \to (\mathrm{RTraces}(E) \to \mathcal{R})$$

whose definition is given by a certain weighted summation formula, reminiscent of an expectation over rated traces. The complete definitions require more space than is available here, and can be found in [SS98]. The next few paragraphs outline the basic ideas.

Intuitively, an observable $\Phi : \mathrm{RTraces}(E \cup E_A) \to \mathcal{R}$ represents a performance measure applicable to a system consisting of A and possibly other components. The function $\mathcal{B}_E^A$ takes such a performance measure Φ and produces a new performance measure $\mathcal{B}_E^A \Phi$ that incorporates the effect of A, and which consequently is applicable to just the remaining system components. This performance measure $\mathcal{B}_E^A \Phi$ is itself an observable, which takes as its argument a rated trace α that represents constraints imposed on A by a particular execution of the other components. Given Φ and α, the quantity $(\mathcal{B}_E^A \Phi)\alpha$ is obtained by enumerating the set $C_A(\alpha)$ of all executions σ of A that are "compatible" with the constraint α, "combining" each such execution σ with α to obtain a new constraint "$\sigma \oplus \alpha$" that takes into account both σ and α, and forming the weighted summation

$$\sum_{\sigma \in C_A(\alpha)} \Phi(\sigma \oplus \alpha) w_A(\sigma),$$

where the "weight" $w_A(\sigma)$ of an execution σ is defined as the product of the probabilities of the transitions in σ, times the product of the rates associated with each state in σ from which an action $e \in E_A^{\mathrm{loc}}$ occurs.

The following key facts about PIOA behaviors were shown in [SS98]:

1. Behaviors are a *compositional* description of PIOAs, in the following sense: Suppose A and B are compatible PIOAs, and E is a set of actions. Then $\mathcal{B}_E^{A\|B} = \mathcal{B}_E^A \circ \mathcal{B}_{E \cup E_A}^B = \mathcal{B}_E^B \circ \mathcal{B}_{E \cup E_B}^A$.

2. Certain performance measures for PIOAs can be expressed in terms of the application of their behaviors to observables. In particular, the probability of a closed PIOA (*i.e.* one having no input actions) performing an execution having a trace that lies in a specified "target set" (we refer to this as the *completion probability* for the set), or the expected time for a closed PIOA to perform an execution having a trace in a specified target set (we refer to this as the *expected completion time* for the set), can be expressed in this way.

As a specific example of how a performance measure can be represented by an observable, suppose we are interested in the probability that a given closed PIOA will eventually perform a particular action a. The relevant observable is the function Φ whose value on a rated trace α of the form $d_0 \xrightarrow{e_0} d_1 \xrightarrow{e_1} \dots \xrightarrow{e_{n-1}} d_n$ is given by $\prod_{k=0}^{n-1} \frac{1}{d_k}$, if $e_{n-1} = a$ and $e_k \neq a$ for all other k, and whose value is 0 for all other α. Given a PIOA A, the quantity $(\mathcal{B}_{\emptyset}^{A} \Phi)(0)$ is then the desired probability.

More generally, it is a consequence of the summation formula defining PIOA behavior that, for a closed PIOA A, any performance measure that can be described as the expectation of a function f on finite delayed traces, can be expressed as the application of the behavior of A to a suitable observable. Included among such performance measures are all transient properties of A that can be expressed in terms of the sequence of actions performed in an execution, without referring to the specific internal states traversed. In addition, inasmuch as certain types of long-run average performance measures can be expressed as limits of transient properties "as time goes to infinity," it is also possible to use our observable/behavior paradigm for such properties as well.

Taken together, facts (1) and (2) above imply that the composition of a compatible collection of PIOAs can be analyzed for certain performance measures by calculating the behavior of the composite PIOA in a compositional, component-by-component fashion, and then extracting the quantities of interest from the resulting behavior. In this approach, the composite PIOA is never computed explicitly. Alternatively, such performance measures could be evaluated in a global, non-compositional fashion by first performing an explicit calculation of the composite PIOA, and performing a more traditional Markovian analysis on it. The advantage of the compositional approach over the global approach is that the former offers the possibility of applying reductions after each component is treated, with a consequent reduction in the total space required for the analysis and a corresponding increase in the size of the systems that can be treated.

3 Representable Observables

The key idea that makes it possible to actually calculate performance parameters of PIOAs using the compositional approach outlined in the previous section is the notion *representable observable*. This is an observable (*i.e.* a function from rated traces to real numbers) whose value on a rated trace can be obtained by running a kind of vector automaton on that rated trace. Our representable observables can be viewed as generalizations of the classical notion of *linear representation* for *formal power series* [BR84].

Formally, let $\mathrm{Obs}(E)$ denote the set of all observables $\Phi : \mathrm{RTraces}(E) \to \mathcal{R}$. Let $\mathrm{Rat}(x)$ denote the set of all rational functions of a single parameter x. Such functions can be expressed as quotients of polynomials with integer coefficients. For n a nonnegative integer, an *n-dimensional representation* of an observable $\Phi \in \mathrm{Obs}(E)$ consists of

- An n-dimensional row vector C with rational numbers as entries.
- An n-dimensional column vector $D(x)$ with entries in $\mathrm{Rat}(x)$,
- For each $a \in E$, an $n \times n$ matrix $M_a(x)$, with entries in $\mathrm{Rat}(x)$,

such that for all rated traces $\alpha \in \mathrm{RTraces}(E)$ of the form:

$$d_0 \xrightarrow{e_0} d_1 \xrightarrow{e_1} \dots \xrightarrow{e_{m-1}} d_m,$$

the quantity $\Phi(\alpha)$ is given by the formula:

$$\Phi(\alpha) = C \left(\prod_{k=0}^{m-1} M_{e_k}(d_k) \right) D(d_m),$$

An observable $\Phi \in \mathrm{Obs}(E)$ is called *representable* if there exists an n-dimensional representation of Φ, for some n.

As indicated above, a representation is essentially a kind of automaton that computes an observable (*i.e.* a function on rated traces). The states of the automaton are n-dimensional row vectors of real numbers, with the vector C serving as the initial state. If the automaton is in state X, and the next portion of the input is $d \xrightarrow{a}$, then the automaton multiplies the current state vector by the matrix $M_a(d)$, and advances the input pointer. Upon reaching the end of the input, if the current state is X' and the single remaining rate is d, then the row vector X' is multiplied by the column vector $D(d)$, to obtain a scalar, which becomes the output produced by the automaton.

The main results of [SS98] were that algorithms exist (see details later in this section) for the following operations on PIOAs and representations:

Application: Suppose A is a PIOA. If Φ is a representable observable in $\mathrm{Obs}(E)$, and $E_A \subseteq E$, then $\mathcal{B}^A_E \Phi$ is also a representable observable in $\mathrm{Obs}(E)$. Moreover, a representation of $\mathcal{B}^A_E \Phi$ can be computed from a representation of Φ.

Minimization: There exists an algorithm that, given an n-dimensional representation of an observable $\Phi \in \mathrm{Obs}(E)$, outputs an m-dimensional representation of Φ, which is minimal in the sense that any other representation of Φ has dimension at least m.

Restriction: Suppose A is a PIOA, and Φ is a representable observable in $\mathrm{Obs}(E \cup E_A)$ for an arbitrary finite set of actions E. Then a representation of $\mathcal{B}^A_E \Phi$ can be computed from a representation of $\mathcal{B}^A_{E \cup E_A} \Phi$.

In addition, we showed that representable observables exist corresponding to a class of performance measures that includes completion time and completion probability for target sets that are *regular* in the usual automata-theoretic sense. Thus, completion probabilities or expected completion times for the closed composition of a compatible collection $\{A_1, A_2, \ldots A_m\}$ of PIOAs may be calculated compositionally as follows:

Compositional Analysis Method:

1. Let $E = E_{A_1} \cup E_{A_2} \cup \ldots \cup E_{A_m}$, and construct a representation for the appropriate starting observable $\Phi_0 \in \mathrm{Obs}(E)$, corresponding to the particular performance measure to be analyzed.
2. Treating the component PIOAs A_i one at a time, construct, successively, representations for the observables Φ_1, Φ_2, ..., Φ_m, where $\Phi_k = \mathcal{B}_E^{A_k}\Phi_{k-1}$, for $k = 1, 2, \ldots, m$. The construction of the representation of Φ_k from Φ_{k-1} is done by first using the Application algorithm to apply the behavior of A_k, and then using the Minimization algorithm to minimize the dimension of the result. The order in which components are applied is inessential as far as the correctness of the method is concerned.
3. Once a representation of $\Phi_m = \mathcal{B}_E^A\Phi_0$ has been obtained, use the algorithm for restriction to compute a representation for $\mathcal{B}_\emptyset^A\Phi_0$, and evaluate this representation on the empty rated trace $\langle 0 \rangle$. The resulting scalar value is the desired performance measure.

To prepare the way for discussing the implementation of the compositional analysis technique outlined above, in the next few subsections we discuss in more detail the mathematical constructions underlying the Application, Minimization, and Restriction algorithms.

3.1 Application

The algorithm for application takes as input a PIOA A and an n-dimensional representation

$$R = (C, D(x), \{M_a(x) : a \in E\})$$

of an observable Φ in $\mathrm{Obs}(E)$, where $E_A \subseteq E$, and it constructs a representation R' for the observable $\mathcal{B}_E^A\Phi$. If the PIOA A has m states: $q_1, q_2, \ldots, q_m$, with q_1 the distinguished start state, then the representation R' will be the mn-dimensional representation

$$R' = (C', D'(x), \{M'_a(x) : a \in E\}),$$

where C', $D'(x)$, and $M'_a(x)$ are given in n-dimensional block form as follows:

$$C' = (\, C \; 0 \ldots 0 \,) \qquad\qquad D'(x) = \begin{pmatrix} D(x + \rho_A(q_1)) \\ D(x + \rho_A(q_2)) \\ \ldots \\ D(x + \rho_A(q_m)) \end{pmatrix}$$

$$(M'_a)_{ij}(x) = \begin{cases} \mu_A(q_i,a,q_j)M_a(x+\rho_A(q_i))\rho_A(q_i), & \text{if } a \in E_A^{\mathrm{loc}}, \\ \mu_A(q_i,a,q_j)M_a(x+\rho_A(q_i)), & \text{otherwise.} \end{cases}$$

In the above, an expression such as $D(x+\rho_A(q_1))$ denotes the result of substituting the linear polynomial $x+\rho_A(q_1)$ for the variable x in each entry of $D(x)$.

3.2 Minimization

The algorithm for minimization takes as input an n-dimensional representation

$$R = (C, D(x), \{M_a(x) : a \in E\}),$$

of an observable $\Phi \in \mathrm{Obs}(E)$, and ouptuts an m-dimensional representation

$$R' = (C', D'(x), \{M'_a(x) : a \in E\}),$$

which is minimal. The algorithm involves performing two separate dimension-reducing steps, which we refer to as *Reachability* and *Observability*, which are in a sense dual to each other, and which can be applied in either order.

Reachability: Let S be the (column) vector space consisting of all solutions Y to the (infinite) system of all equations of the form:

$$C(\prod_{k=0}^{l-1} M_{a_k}(d_k))Y = 0, \tag{1}$$

where $a_0, a_1, \ldots, a_{l-1}$ range over actions in E, and $d_0, d_1, \ldots, d_{l-1}$ range over nonnegative real numbers (or equivalently, nonnegative rational numbers). Let $S^\perp$ denote the orthogonal complement of S; then $S^\perp$ is the least subspace of $\mathcal{R}^n$ (rows) containing the start vector C and closed under multiplication on the right by matrices of the form $M_{a_k}(d_k)$. That is, $S^\perp$ is the least subspace of $\mathcal{R}^n$ containing all row vectors X that are *reachable* from the start vector C by running the representation R on some rated trace.
If the solution space S has dimension zero, then do nothing. Otherwise, suppose S has dimension $k > 0$, and let $m = n - k$ be the dimension of $S^\perp$. Let P be an $n \times m$ matrix having the elements of an orthogonal basis for $S^\perp$ as its columns, and let Q be the $m \times n$ matrix whose rows are the same basis vectors for $S^\perp$, only with each vector divided by the square of its euclidean norm, so that the equation $QP = I$ is satisfied. Then it can be shown (see [SS98]) that the representation:

$$R' = (CP, QD(x), \{QM'_a(x)P : a \in E\})$$

is an m-dimensional representation of the observable Φ.

Observability: Let S be the (row) vector space consisting of all solutions X to the (infinite) system of all equations of the form:

$$X(\prod_{k=0}^{l-1} M_{a_k}(d_k))D(d_l) = 0.$$

Then S is the greatest subspace of $\mathcal{R}^n$ that consists entirely of row vectors X that are *unobservable*, in the sense that it is impossible to obtain a nonzero value by starting from X and running representation R on any rated trace. If the solution space S has dimension zero, then do nothing. Otherwise, suppose S has dimension $k > 0$, and let $S^{\perp}$ denote the orthogonal complement of S, so that $S^{\perp}$ has dimension $m = n - k$. Letting P, Q, and R' be as in the Reachability case above, it can once again be shown that the representation R' is an m-dimensional representation of the observable Φ.

Our minimization algorithm for representations of observables can be seen as a variant of a classical algorithm of Schützenberger [Sch61a,Sch61b,BR84] for minimization of linear representations of formal power series.

3.3 Restriction

In [SS98] we defined an operation of *restriction*:

$$[\text{ - }]_{E'} : \mathrm{Obs}(E) \to \mathrm{Obs}(E')$$

on observables, which, when applied to an observable $\Phi \in \mathrm{Obs}(E)$, produced a certain observable $[\text{ - }]_{E'} \in \mathrm{Obs}(E')$, where $E' \subseteq E$. The restriction operation was defined in such a way that the following property is satisfied: for all PIOAs A, all sets of actions E, and for all observables $\Phi \in \mathrm{Obs}(E \cup E_A)$, we have:

$$\mathcal{B}_E^A \Phi = [\mathcal{B}_{E \cup E_A}^A \Phi]_E$$

Restriction on observables therefore gives us a way to obtain $\mathcal{B}_E^A \Phi$ from $\mathcal{B}_{E \cup E_A}^A \Phi$. This is of interest to us, because our compositional analysis method applied to a PIOA A involves determining the observable $\mathcal{B}_\emptyset^A \Phi$. The Application operation by itself would only permit us to determine $\mathcal{B}_{E_A}^A \Phi$; however Restriction can then be applied to this to yield $\mathcal{B}_\emptyset^A \Phi$.

As shown in [SS98], the Restriction operation can be performed algorithmically in terms of representations. The algorithm takes as input an n-dimensional representation $R = (C, D(x), \{M_a(x) : a \in E\})$ of an observable $\Phi \in \mathrm{Obs}(E)$. Define

$$\hat{M}(x) = \sum_{a \in E \setminus E'} M_a(x),$$

and let $\hat{M}(x)^*$ denote the Kleene star $\hat{M}(x)^* = I + \hat{M}(x) + \hat{M}(x)^2 + \ldots$, which, if it exists, can be calculated as $\hat{M}(x)^* = (I - \hat{M}(x))^{-1}$. The algorithm outputs the representation:

$$R = (C, \hat{M}(x)^* D(x), \{\hat{M}(x)^* M_a(x) : a \in E'\}),$$

which can be shown to be a representation of the observable $[\Phi]_{E'}$.

3.4 Example

Let A be a two-state closed PIOA A, with states $Q = \{q, q'\}$, actions $E = E^{\text{out}} = \{a, b\}$, with $\rho(q) = 3$, $\rho(q') = 0$, and with $\mu(q, b, q) = \mu(q, a, q') = 1/2$ as the only nonzero values of μ. Let $R = (C, D(x), \{M_a(x), M_b(x)\})$ be the two-dimensional representation with

$$C = \begin{pmatrix} 1 & 0 \end{pmatrix} \qquad D(x) = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \qquad M_a(x) = \begin{pmatrix} 0 & 1/x \\ 0 & 0 \end{pmatrix} \qquad M_b(x) = \begin{pmatrix} 1/x & 0 \\ 0 & 0 \end{pmatrix}$$

The reader may verify that this representation defines an observable Φ that maps a rated trace $\alpha = d_0 \xrightarrow{e_0} d_1 \xrightarrow{e_1} \ldots \xrightarrow{e_{n-1}} d_n$ to a nonzero value if and only if $e_{n-1} = a$ and $e_k = b$ for all $k < n-1$, and in this case $\Phi(\alpha) = \prod_{k=0}^{n-1}(1/d_k)$. Moreover, the Application construction may be used to calculate a four-dimensional representation for $\mathcal{B}_E^A \Phi$:

$$C = \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix}$$

$$D(x) = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} \qquad M_a(x) = \begin{pmatrix} 0 & 0 & 0 & \frac{3}{2(x+3)} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad M_b(x) = \begin{pmatrix} \frac{3}{2(x+3)} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

from which it can be observed that

$$(\mathcal{B}_E^A \Phi)\alpha = \frac{1}{2^n} \prod_{k=0}^{n-1} \frac{3}{d_k + 3},$$

the probability of A performing the uninterrupted sequence of actions $b^{n-1}a$, in an environment that visits, successively, states with rates $d_0, d_1, \ldots, d_n$.

Finally, the Restriction construction may be used to calculate a representation for $\mathcal{B}_\emptyset^A \Phi$ and show that its value on a rated trace (d) is the constant 1 (independent of d), the probability that A eventually performs action a in an arbitrary environment. This calculation amounts to summing the quantity $(1/2)^n(3/(0+3))^n$ over all $n > 0$.

4 Implementation

We have constructed a computer implementation of the compositional analysis method for PIOAs described above. The implementation consists of about 10,000 lines in the programming language Standard ML, about three-quarters of which we wrote ourselves, and about one-quarter of which are subroutines from the code libraries distributed with the Standard ML of New Jersey system. Standard ML was chosen for the implementation because: (1) it is a very-high-level language that was originally designed specifically for symbolic processing; (2) its garbage-collected automatic storage management allowed us to rapidly prototype a system with many fewer lines of code than would have been required, say, in C++; (3) it has an advanced modules facility that allowed us to experiment rapidly with a variety of combinations of algorithms and data representations; (4) it has a good compiler (SML of New Jersey) that produces efficient native code whose execution speed is within striking distance of C++ for symbolic processing applications.

Our system has the following main components:

- Modules for performing basic arithmetic operations on arbitrary-precision integers and rational numbers.
- Modules for calculating with univariate polynomials and rational functions.
- Modules implementing sparse vectors and matrices.
- A PIOA module, which permits the construction of PIOAs, either "from scratch," or as the composition of a compatible collection of previously constructed PIOAs, and which also provides routines for enumerating and iterating over sets of PIOA executions.
- A module implementing representable observables, and the Application, Minimization, and Restriction algorithms.
- A module that permits the specification of regular "target sets" by means of deterministic finite automata (DFAs), and of constructing from these DFAs representations for the associated completion probability and expected completion time observables.

Except for some variables used to turn on debugging printout, and exceptions used to handle error situations, the implementation is coded in a purely functional style.

We have spent several man-months in testing the code and improving its performance. As of this writing, the largest problem on which we have run the system successfully is the analysis of the probability of the first player winning in a model of a four-player, two-point jai-alai match. The model consists of a system of nine component PIOAs, which, if the composition were explicitly constructed, would have a total global state space of 78,764,805 ($= 5 \cdot 7^4 \cdot 9^4$) states. (See Section 5.1 for further details of this example.) The compositional analysis of this system produces an exact rational answer (5/16) in 532 minutes of CPU time on a 333MHz Sparc Ultra. In the next few sections, we mention some of the more important ideas and techniques underlying our implementation.

4.1 Polynomials and Rational Functions

A central role is played in both the theory and the implementation by univariate rational functions, which we have implemented as quotients of univariate polynomials. When we began constructing the implementation, we did not know whether it would be best to use polynomials with arbitrary-precision rational coefficients, polynomials with arbitrary-precision integer coefficients, or polynomials with floating-point coefficients. So, we built generic implementations that can operate with any of these choices. Changing from one choice to another requires only relinking of the modules; changes to the source code are not required. A useful byproduct of our generic approach is that we obtain for free the additional ability to compute using polynomials having polynomials as coefficients.

This permits us, in some cases, to obtain as the result of performance analysis a symbolic expression that shows how the performance quantity of interest varies with a parameter of the model.

The implementation of rational functions as quotients of polynomials requires that the rational functions be maintained in canonical form, so that the greatest common divisor (GCD) of the numerator and denominator is one. (If rational functions are not maintained in canonical form, the size of the representation tends to grow excessively, due to the accumulation of cancellable factors.) We were initially not sure how critical the choice of GCD algorithm would be to the performance of our algorithms, so we implemented three different GCD algorithms: the Euclidean algorithm, the primitive polynomial remainder sequence (PPRS) algorithm and the subresultant polynomial remainder sequence (SPRS) algorithm [Bro71]. In theory, the Euclidean algorithm poses a significant danger of explosion in the size of coefficients of the intermediate polynomials. We found this to be true for our application, and so we do not use this algorithm. The PPRS algorithm handles the explosion problem by removing common factors from the coefficients of the intermediate polynomials produced during the GCD algorithm. The SPRS algorithm has some advantages over the PPRS algorithm, with respect to the use of the underlying GCD operation for coefficients, but in general will produce intermediate polynomials that are not as small as those produced by the PPRS algorithm. However, in our benchmarks the SPRS algorithm has generally produced satisfactory results. If size explosion should prove to be a problem with examples we treat in the future, we will expend the effort to implement one of the newer, modular GCD algorithms.

After experimentation with various combinations of coefficients and GCD algorithms, we settled on arbitrary-precision integer coefficients and the SPRS algorithm as our standard configuration. All the benchmarks reported in this paper were run in this way.

4.2 Sparse Vectors and Matrices

The vectors and matrices arising in the compositional analysis of systems of PIOAs are sparse, and substantial economies of time and space are obtained by taking advantage of this fact. Of the several representations we tried, we have obtained our best performance results by representing sparse vectors as lists, and sparse matrices as trees that map row indices to sparse vectors. Specifically, sparse vectors are represented as triples

```
(dim, low, entries)
```

where `dim` is the dimension of the vector, `low` is an integer offset, and `entries` is a list of pairs `(i,v)`. The meaning of such a pair is that `v` is the nonzero data value at position `i-low+1` in the vector. In general the list `entries` may contain pairs `(i,v)` for which the relationship `low <= i <= low+dim-1` is not satisfied. However, only pairs that do satisfy this relationship actually denote elements of the vector; the other entries are ignored. This kind of representation permits us to implement a constant-time "slicing" operation on vectors, which is useful for some of our constructions on representations.

Sparse matrices are implemented as mappings from row indices to sparse vectors. These mappings, which are implemented using the "binary map" module distributed with the Standard ML of New Jersey system, use trees as the underlying representation. The same `dim/low` scheme is used for matrices as for vectors, to provide for efficient slicing operations.

A significant amount of the time spent in the minimization algorithm is used in computing bases, for the null space S of a large matrix M, and for its orthogonal complement $S^{\perp}$. For this, we use a Gaussian elimination procedure. Our Gaussian elimination routine is unremarkable except that it is coded in a purely functional style, and we have gone to some trouble to make the code run quickly. We use a pivoting technique to determine the row to eliminate against at each iteration. However, the usual technique, of selecting as a pivot at the kth iteration the row whose kth column contains the element of the greatest magnitude, is difficult to apply to non-numeric data such as polynomials or rational functions, because the appropriate notion of magnitude is not clear. So, at present we use as the pivot row the first row encountered that has a nonzero entry in the kth column. Since in our standard configuration we use exact arithmetic, numerical stability is not a concern.

4.3 Algorithms on Representations

In this section, we discuss issues related to the implementation of the Application, Minimization, and Restriction constructions on representations.

Application: Our implementation of the Application algorithm, with which a PIOA behavior is applied to a representation, is a reasonably straightforward transcription of the definition given in Section 3.1. As suggested by the definition we construct the vectors C and $D(x)$, as well as the matrices $M_a(x)$, by "pasting" together their constitutent blocks. As the algorithm is coded in a purely functional style, some care is needed to avoid excessive copying of data structures during the pasting operations. Note that the constructions of $D(x)$ and the $M_a(x)$ require the use of symbolic arithmetic on rational functions, including an operation for performing the symbolic substitution of a linear univariate polynomial $x + d$ for the indeterminate x in a rational function $r(x)$.

The time taken to run the Application algorithm does not contribute significantly to the running time of our benchmark examples, so we do not discuss it further.

Restriction: In the first versions of our compositional analysis technique, we attempted to apply the Restriction algorithm after each component was applied, to restrict the set of actions to just those that interface with the system components remaining to be treated. Our idea here was that performing such restriction would permit the minimization algorithm to produce a greater reduction in dimension. However, a direct implementation of the Restriction algorithm outlined above requires calculating the inverse of the matrix $I - \hat{M}(x)$ symbolically, over the field of rational functions in x. Although this can readily be done for small, sparse matrices, once the number of dimensions increases beyond about 100, or the matrices become more dense, there is a significant danger of explosion in the size of the rational expressions produced in calculating the inverse. We have not found it feasible to apply the Restriction operation to representations having thousands of dimensions.

Fortunately, for the purpose of calculating numerical completion probabilities or expected completion times, it is not necessary to apply the Restriction operation as each component is treated. Rather, the application of Restriction can be delayed until after all components have been treated, and the final representation for the observable $\Phi_m = \mathcal{B}_E^A \Phi_0$ for the entire system has been obtained. At this point, the minimization algorithm has typically produced a significant reduction in the number of dimensions. In addition, extracting the final numerical answer requires only that we evaluate the representation for $[\Phi_m]_\emptyset$ on the empty rated trace (0), not that we compute this representation in full. Inspection of the definition of Restriction shows that the value of $[\Phi_m]_\emptyset$ on the empty rated trace (0) can be expressed solely in terms of particular instances, $D(0)$ and $\{M_a(0) : a \in E\}$, of the components of a representation for Φ_m; in particular the full symbolic representation of $D(x)$ and the $M_a(x)$ is not required. Thus, we can completely avoid symbolic matrix inversion by first "instantiating" the representation for Φ_m at (0) to obtain a "constant" representation:

$$(C, D(0), \{M_a(0) : a \in E\}),$$

and then applying the Restriction algorithm to this to yield the value of $[\Phi_m]_\emptyset$ at (0).

Minimization: The minimization algorithm is what consumes most of the CPU cycles in all but the smallest examples, and it is what we have spent the most time trying to optimize.

The first thing to note is that the observability portion of the minimization algorithm is essentially a "time-reversed" version of the reachability portion, and we can use the reachability code for the observability part if we interchange the role of the starting vector C and the output vector $D(x)$, and transpose all the matrices $M_a(x)$. To eliminate the asymmetry between the scalar vector C and the vector $D(x)$ of rational functions, in our implementation we treat C as a vector of rational functions that happen to be constant.

To achieve full minimality, both the reachability and observability portions of the algorithm have to be run, in general. Most of the time spent in these algorithms involves an iterative search for the spaces of "reachable states" and "unobservable states" described in Section 3.2. Since we do not have any *a priori* idea which of the two spaces will be found more quickly, we dovetail the two iterations in such a way as to spend about the same amount of work on each. Whichever search finishes first has its corresponding reduction applied to the representation under consideration, and the slower search is then restarted on the reduced representation.

Though reachability and observability are in theory just time-reversed versions of each other, in practice they exhibit significant qualitative differences. In particular, we have found that, whereas the reachability algorithm generally tends to run quickly, the observability algorithm often runs much more slowly. We attribute the differences between these two algorithms to the fact that the reachability algorithm involves a search "forward in time" with a low "branching factor" (see below) that is probably related to the amount of concurrency in the system being analyzed, whereas the observability algorithm involves a search "backward in time", which has a higher branching factor. The observability algorithm also tends to destroy sparseness in the intermediate vectors and matrices it produces, further contributing to a slower running time. In our benchmark examples, we have found that observability algorithm runs slowly enough that any speedup from the additional reduction in dimension it might afford is generally overshadowed by the time taken to find that reduction. Our fastest benchmark times at present are generally obtained with the observability portion of the minimization algorithm disabled.

We now describe in more detail the method we use to solve the equations (1) of Section 3.2 to obtain a basis for the space of "reachable states" for a representation. The algorithm is essentially a fixed-point iteration to search for the least subspace of n-dimensional Euclidean space that contains the starting vector C of the representation being minimized, and which is closed under multiplication on the right by the infinite set of matrices $M_a(d)$, where a ranges over the set of actions of the system under consideration, and d ranges over the rationals. The input to the kth iteration is pair of matrices (B_k, C_k), where B_k is a row basis for an m_k-dimensional subspace S_k of n-dimensional Euclidean space, and C_k is a column basis for its orthogonal complement $S_k^{\perp}$. Initially, B_0 is taken to be the $1 \times n$ matrix having the starting vector C as its only row. The output of the kth iteration is a new such pair of matrices (B_{k+1}, C_{k+1}), where the space S_{k+1} spanned by the rows of B_{k+1} is that spanned by the rows of B_k and all vectors of the form $XM_a(d)$ with X a row of B_k, a an action, and d a rational number. Since S_k is a subspace of S_{k+1}, the number of rows m_{k+1} of B_{k+1} is at least as great as m_k. If $m_{k+1} = m_k$, then a fixed point has been reached, yielding a solution to the equations (1). Since the entire computation is being carried out in n-dimensional Euclidean space, we have $m_{k+1} \leq n$, so termination is guaranteed.

Since the matrices of the form $M_a(d)$ comprise an infinite collection, it is obviously not possible to compute (B_{k+1}, C_{k+1}) from (B_k, C_k) by a direct enumeration of all vectors of the form $XM_a(d)$. However, B_{k+1} can still be computed by noting that the vectors in B_{k+1} are precisely all those row vectors X for which all matrix equations of the form $XM_a(x)C_k = 0$ are satisfied identically for all x. This finite system of linear equations, with coefficients in the field of rational functions in x, can be converted into an equivalent finite set of equations with constant coefficients, simply by equating the coefficients of each power of x on the left-hand side separately to zero. Solving these scalar equations yields a basis for B_{k+1}.

In practice, we carry out the above procedure in the following way. Before beginning the iteration process, we preprocess the collection of $n \times n$ matrices $\{M_a(x) : a \in E\}$ from the representation to convert it into a larger collection $\{M_{a,i} : a \in E, i \in I_a\}$ of scalar matrices, which yields the same solution space as the original matrices. This is done by taking each of the matrices $M_a(x)$ and first eliminating the denominators, by multiplying each entry by the least common multiple of the denominators of all the entries. The resulting matrix of polynomials is then "sliced" to obtain a separate matrix of scalar coefficients for each power of x. After the preprocessing step, all further calculations are carried out using scalar arithmetic, rather than arithmetic on rational functions.

The fixed-point iteration is now carried out for the collection $\{M_{a,i} : a \in E, i \in I_a\}$ as follows. The starting matrix B_0 is taken to be the $1 \times n$ matrix having the starting vector C as its only row. (For the observability part of the minimization algorithm, the vector $D(x)$ is used instead, and a "slicing" construction such as that described above for the $M_a(x)$ is used to convert $D(x)$ into a list of constant vectors that become the rows of B_0.) The starting matrix C_0 has as its columns the vectors of a basis for the null space of B_0, which is calculated using Gaussian elimination.

At the kth iteration, we calculate a list of all matrices of the form $B_k M_{a,i}$, and "stack" these matrices atop each other, together with B_k itself, to form a large matrix M. Gaussian elimination is then used to calculate bases for the null space of M and its orthogonal complement. These bases become, respectively, the columns of C_{k+1} and rows of B_{k+1}. In our actual implementation, we use two optimizations that improve the running time of this procedure. The first is targeted at reducing the number of rows in M, thereby reducing the amount of time needed to calculate its null space. For this, we note that most of the rows of the matrices $B_k M_{a,i}$ tend to be linearly dependent on the rows of B_k, and therefore impose no new constraints. So, in constructing the matrix M we include only those rows X of $B_k M_{a,i}$ for which XC_k is nonzero. The second optimization attempts to reduce the amount of redundant work performed in constructing the matrix M over successive iterations of the algorithm. Rather than working with all rows of B_k at each iteration, we maintain at each iteration a smaller list B'_k of rows that are "new," in the sense that they are independent of rows already treated at earlier iterations. Only the rows in B'_k are used in the construction of M.

Our minimization algorithm can be thought of as essentially a breadth-first search to determine the least subspace of n-dimensional space that contains a specified starting subspace and is closed under multiplication by the $M_{a,i}$. A notion of *branching factor* at the kth iteration can be defined as the ratio of the number of rows of the matrix M to the number of rows in B'_k. It is the number of rows in M that controls the time taken to perform each iteration. In the typical runs of the algorithm that we have observed, the reachability portion of the minimization algorithm tends to have a relatively small, roughly constant branching factor, which leads to a modest and predictable amount of work at each iteration. In contrast, the observability portion of the algorithm generally seems to have larger, less predictable branching factors, and produces larger and denser matrices whose null spaces sometimes take a long time to compute.

5 Test Cases

In this section, we discuss the performance of our implementation on some benchmark examples. To date, we have been primarily concerned with removing bugs from our implementation, improving its performance, and getting an idea of how much reduction might be afforded by the minimization algorithm. For this reason, our benchmark examples have primarily been ones for which we could determine the correct answers by other means, and which had some parameters we could change to produce instances of different sizes and numbers of components.

5.1 Jai-alai Match

Our primary benchmark example has been a model of jai-alai match. In the game of jai-alai, n players occupy seats numbered 1, 2, ..., n. The players in seats 1 and 2 enter into a contest whose outcome depends on skill and chance. After the contest, the players change seats, with the winner of the match taking seat 1, the loser taking seat n, and all the other players shifting up by one seat. The winner of each contest scores one point, and play continues until some player has scored k points.

We used a system of PIOAs to model an n-player, k-point jai-alai match in which all players are equally skilled, so that in each round there is an equal probability that either of the two players will win. Our PIOA model consists of one PIOA called the *referee*, n PIOAs called the *players*, and n PIOAs called the *scorers*, for a total of $2n+1$ component PIOAs. Players keep track of what seat they are currently in, so that the number of states of a player PIOA is dependent on n (specifically, $n+5$). Each scorer keeps track of the total number of points that has been accumulated by a single player, so that the number of states of a scorer PIOA is dependent on k (specifically, $2k+3$). The number of states of the referee is 5, independent of n and k.

The component PIOAs interact to simulate the jai-alai match. At each round, the players in seats 1 and 2 announce themselves to the referee as "champion" and "challenger," respectively. The champion and challenger act asynchronously in this phase. The referee then decides probabilistically whether the outcome of the contest is an "upset" (challenger wins), or "non-upset" (champion wins), and announces the result to all the players and scorers. Upon being informed of the outcome of the contest, the players all change seats in accordance with the rules, and the scorer associated with the player who has won increments its score by one. Once this has been done, this scorer either announces that it has reached the number of points required to win, or else it issues an "ok" to all the players, signalling that the next round can begin. The PIOA model requires that each state of each component be assigned a rate. We assigned a rate of 1 to each state from which a locally controlled action was enabled, and a rate of 0 to other states. This was adequate for the runs reported below, in which we were only interested in the probability of a particular player winning the match. It would have been possible (by using a different observable) for us to calculate a real-time performance measure, such as the expected time to complete the match. In this case, rates other than 1 and 0 could have been used.

We used our implementation to compute the probability that the player who first accumulates k points and wins the match will be the one who occupied seat 1 at the outset. We performed this computation for various combinations of n and k that produced systems with between 5 and 9 components, and with global state spaces ranging from 12,005 ($= 5 \cdot 7^2 \cdot 7^2$) states to 78,764,805 ($= 5 \cdot 7^4 \cdot 9^4$) states. Figure 1 shows the results of various runs on a 333MHz UltraSparc. The timing results reported were obtained with the observability part of the minimization algorithm disabled, which as we have already mentioned, typically gives the best performance.

Players (n)	**Points (k)**	**Components**	**States**	**Output**	**Time**
2	2	5	12,005	1/2	16 seconds
2	3	5	19,845	1/2	43 seconds
2	4	5	29,645	1/2	126 seconds
2	6	5	55,125	1/2	12 minutes
2	8	5	88,445	1/2	48 minutes
3	2	7	878,080	3/8	9 minutes
4	2	9	78,764,805	5/16	532 minutes

Fig. 1. Jai-alai Analysis Results (observability minimization disabled)

We should note that the results are sensitive to the order in which the components are treated. For all the above runs, we applied the referee first, then each of the players, and finally the scorers. For the 2-player, 2-point case, if we alternate the application of players with scorers, then the minimization algorithm does not produce as large reductions at the early stages and the analysis requires over a minute to complete. The decrease in time from the 2-player, 8-point case and the 3-player, 2-point case seems to be due to the increased possibilities for reduction when there are more components in the system.

As discussed earlier, our code is capable of producing symbolic expressions that show the dependence of a performance measure on a parameter. To illustrate this, we ran the two-player, two-point case and the two-player, three-point case in symbolic mode, where the probability of an "upset" was left as an unspecified parameter p. The two-point case takes essentially the same amount of time to run symbolically, and the expression:

$$1 - 2p + 3p^2 - 2p^3$$

was output in 16 seconds of CPU time. For the three-point case, the expression:

$$1 - 3p + 9p^2 - 16p^3 + 15p^4 - 6p^5$$

was output in 46 seconds of CPU time, as opposed to 43 seconds for the non-symbolic analysis.

Figure 2 uses the 3-player, 2-point case to illustrate the effect of compositional analysis with minimization as each component is treated. On the line with each component, we have listed the number of (local) states of that component, the total number of global states for the portion of the system up to and including that component, the dimension of the representation after applying that component but before minimization, and the dimension of the representation after minimization. The dimensions are reported both in the case that observability minimization is enabled, and in the case it is disabled.

			Observ. on		**Observ. off**	
Component	**States**	**Global**	**Start**	**Min.**	**Start**	**Min.**
Referee	5	5	15	6	15	10
Player 1	8	40	48	34	80	72
Player 2	8	320	272	148	576	354
Player 3	8	2,560	1,184	162	2,832	440
Scorer 1	7	17,920	1,134	193	3,080	520
Scorer 2	7	125,440	1,351	116	3,640	288
Scorer 3	7	878,080	812	59	2,016	80

Fig. 2. Performance of Minimization Algorithm

An advantage of the compositional approach is that memory consumption is not a limiting factor with respect to the size of the examples that can be run. For all of the above runs, the size of the heap never exceeded about 120MB. Since Standard ML of New Jersey requires a heap that is at least three times the amount of active data, this means that the amount of active data produced during the runs did not exceed 40MB.

5.2 Alternating Bit Protocol

Besides the jai-alai example discussed above, we have also run our code on a three-component version of the alternating bit protocol, which has about 500 global states. For this example, it takes only a few seconds to compute the expected time required, from the time a message is input by the sender, to the time that message is successfully acknowledged by the receiver.

6 Related Work

In this section, we discuss briefly some related investigations being pursued by other researchers. A significant point of difference with our work is that these other projects perform numerical analysis by extracting an underlying continuous-time Markov chain and solving it numerically, whereas we are primarily interested in exact solutions that avoid the construction of the global state space. Another point of difference involves the fact that our techniques are most directly applicable to computing transient performance measures, whereas the computation of steady-state probabilities is usually the central focus of traditional Markovian analysis. However, it is possible to exploit the symbolic capabilities of our methods in order to compute certain kinds of steady-state parameters. We are currently investigating this approach.

6.1 Generalized Stochastic Petri Nets

Generalized Stochastic Petri Nets (GSPNs) [MBC84,MBC+94] are an extension of Petri Nets to handle probability and timing. Analysis of the model is performed in a classical style, by extracting a continuous time Markov chain and solving the associated system of linear equations for the steady-state probabilities. For example, the "GreatSPN" tool [CFGR95] can perform reachability graph generation and qualitative analyses such as deadlock and livelock detection, as well as Markovian analysis of both steady-state and transient performance characteristics.

Tools such as GreatSPN use iterative techniques to obtain numerical solutions to large sparse linear systems. Systems with sizes on the order of a million variables can be solved in this way. However, as can be seen from Figure 1, even very simple systems quickly reach a million global states, so compositional techniques will be essential if this type of analysis is to go much farther. In general, Petri net-based models, including GSPNs, have traditionally not lent themselves very well to compositional techniques. One approach to this problem has been to try to transport, to the GSPN setting, standard notions of equivalence and composition from process algebras [HHMR97]. Another approach has been to characterize classes of GSPNs for which the underlying Markov chain has a so-called "product form property" [DS92]. Product form properties permit the state-transition graph of a large system to be decomposed into smaller component graphs, in such a way that the solution for large graph is determined in a simple way by the solutions to the components. This permits the component systems to be solved separately, and the solutions combined to obtain the solution for the original system. A difficulty with the product form approach is that product form properties tend to be destroyed by arbitrary synchronization and interaction between components of a system. Thus, if a system is to submit to this type of analysis, it has to have very limited types of interaction between the components. This condition may be too restrictive, in practice.

6.2 Stochastic Process Algebra

In contrast to Petri nets, *process algebras* are designed with compositionality in mind from the start. *Stochastic process algebras* extend traditional process algebras by adding probabilistic and timing information. Several researchers have constructed analysis tools to work with stochastic process algebras. The PEPA workbench [Hil96], permits systems to be described in a hierarchical fashion. Quantitative analysis of a model is based on numerical solution of the underlying Markov chain. However, the process algebraic framework permits the use of bisimulation-based equivalences to perform state-space reduction [Hil95].

The TIPPtool [HM96,HHK+98] is another analysis tool that is based on stochastic process algebra. A variety of analyses are supported, including steady state and transient analysis of an underlying continuous-time Markov chain. These analyses are performed in a global fashion, by constructing the state space of the Markov chain and then solving numerically the associated linear system. However, bisimulation-based congruences are exploited to perform state-space reduction, which increases the size of the models that can be treated. It is stated in [HHK+98] that iterative numerical methods such as Gauss-Seidel on sparse matrix data structures, permit the solution of Markov chains of up to 100000 states. H. Hermanns (private communication) has indicated that using successive overrelaxation, a variant iterative technique, current analysis capabilities are somewhat higher – between a million and ten million states.

7 Conclusion

We are encouraged by the results we have achieved so far with our techniques. However, when one compares the size of the state space of the examples we have been able to treat with the size of the state spaces that arise out from practical examples, one realizes that there is still some distance to go before practical application of this kind of analysis is routine. The fact that we can perform symbolic analysis is exciting, because one is likely to be willing to wait significantly longer to compute a formula that expresses the dependence of a performance measure on a system parameter than one is to compute just one data point. We would like to improve the symbolic capability of our implementation, and to extend it to encompass dependencies on more than one parameter.

The fact that our techniques are time, rather than storage-bound, raises hopes that the size of the problems that can be treated can still be raised significantly by speeding up the code. There are a variety of "low-tech" tricks we feel could be used to squeeze some additional performance out of our algorithms. Recoding in C++ would probably produce a significant speedup, though the necessary C++ code would probably be at least double the size of our ML code. Heuristics for choosing the best order in which to treat components might be helpful. So far, it seems that a good strategy is to apply "large-state" components such as counters later in the analysis rather than earlier. We feel there are also possibilities of "high-tech" improvements in the algorithms, such as making use of deeper properties of the matrices $M_{a,i}$ to speed up the minimization algorithm.

References

[BR84] J. Berstel and C. Reutenauer. *Rational Series and Their Languages*, volume 12 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1984.

[Bro71] W. S. Brown. On Euclid's algorithm and the computation of polynomial greatest common divisors. *Journal of the Association for Computing Machinery*, 18(4):478–504, October 1971.

[CFGR95] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: Graphical editor and analyzer for timed and stochastic Petri nets. *Performance Evaluation*, 24(1-2):47–68, November 1995. special issue on performance modeling tools.

[DS92] S. Donatelli and M. Sereno. On the product form solution for stochastic Petri nets. In *Proc. of Int. Conf. on Applications and Theory of Petri Nets*, Sheffield, UK, 1992.

[HHK+98] H. Hermanns, U. Herzog, U. Klehmet, M. Siegle, and V. Mertsiotakis. Compositional performance analysis with the TIPPtool. In *PERFORMANCE TOOLS '98*, Lecture Notes in Computer Science. Springer-Verlag, 1998. to appear.

[HHMR97] H. Hermanns, U. Herzog, V. Mersiotakis, and M. Rettelbach. Exploiting stochastic process algebra achievements for generalized stochastic Petri nets. In *Proc. of the 7th Int. Workshop on Petri Nets and Performance Models*, St. Malo, June 1997. IEEE Computer Society Press.

[Hil95] J. Hillston. Compositional Markovian modelling using a process algebra. In W. Stewart, editor, *Proceedings of the Second International Workshop on Numerical Solution of Markov Chains: Computations with Markov Chains*, Raleigh, North Carolina, January 1995. Kluwer Academic Press.

[Hil96] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[HM96] H. Hermanns and V. Mertsiotakis. A stochastic process algebra based modelling tool. In M. Merabti, M. Carew, and F. Ball, editors, *Performance Engineering of Computer and Telecommunications Systems*. Springer-Verlag, 1996.

[Lyn96] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[MBC84] M. Ajmone Marsan, G. Balbo, and G. Conte. A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. on Computer Systems*, 2:143–172, 1984.

[MBC+94] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, 1994.

[Sch61a] M. P. Schützenberger. On a special class of recurrent events. *Annals Math. Stat.*, 32:1201–1213, 1961.

[Sch61b] M. P. Schützenberger. On the definition of a family of automata. *Information and Control*, 4:245–270, 1961.

[SS98] E. W. Stark and S. Smolka. Compositional analysis of expected delays in networks of probabilistic I/O automata. In *Proc. 13th Annual Symposium on Logic in Computer Science*, pages 466–477, Indianapolis, IN, June 1998. IEEE Computer Society Press.

[WSS97] S.-H. Wu, S. A. Smolka, and E. W. Stark. Composition and behaviors of probabilistic I/O automata. *Theoretical Computer Science*, 176(1-2):1–38, 1997.