Event Count Automata: A State-based Model for Stream Processing Systems

Samarjit Chakraborty Linh T. X. Phan P. S. Thiagarajan
Department of Computer Science, National University of Singapore
E-mail: {samarjit, phanthix, thiagu}@comp.nus.edu.sg

Abstract

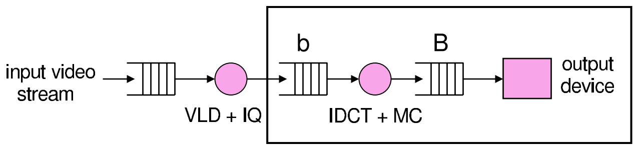
Recently there has been a growing interest in models and methods targeted towards the (co)design of stream processing applications; e.g. those for audio/video processing. Streams processed by such applications tend to be highly bursty and exhibit a high data-dependent variability in their processing requirements. As a result, classical event and service models such as periodic, sporadic, etc. can be overly pessimistic when dealing with such applications. In this paper, we present a new model called Event Count Automata (ECA) for capturing the timing properties of such streams. Our model can be used to cleanly formulate properties relevant to stream processing on heterogeneous multiprocessor architectures, such as buffer overflow/underflow constraints. It can also provide the basis for developing analysis methods to compute delay/timing properties of the processed streams under different scheduling policies. Our ECAs, though similar in flavor to timed and hybrid automata, have a different semantics, are more light-weight, and are specifically suited for modeling stream processing applications and architectures. We present the basic aspects of this model and illustrate its modeling potential. We then apply it in a specific stream processing setting and develop an analysis technique based on the formalism of Colored Petri Nets (CPNs). Finally, we validate our modeling and analysis techniques with the help of preliminary experimental results generated using the CPN simulation tool.

1 Introduction

Models and algorithms specifically targeted towards "stream processing" have recently been the subject of extensive study in the embedded systems domain (see for example [23, 12]), as well as in other areas of computer science such as databases. A large number of applications such as audio/video and network packet processing can be naturally modelled, designed and analyzed using the "stream" abstraction. Such applications often handle multiple and potentially infinite streams of data and run on devices ranging from mobile phones, PDAs, set-top boxes and network routers. The widespread use of such applications has led to new "stream-centric" programming languages and compilers [10], processor architectures [11] and design methodologies [18, 26].

In line with these developments, in this paper we introduce a simple and novel modeling formalism called *Event Count Automata* to study stream processing systems. In our view of such systems, an application is partitioned and mapped onto multiple processing elements (PEs). A stream (or possibly multiple streams) of data enters a PE, gets processed by the task/function mapped onto this PE, and the processed stream enters another PE for further processing. Between any two neighboring PEs, there is a buffer which stores the partially processed stream. After a series of such steps, the stream is fully processed and leaves the system. This view covers a variety of architectures targeted towards stream processing (such as the one described in [22]). In this setup, we wish to address questions of the following form: Will any of the buffers in the architecture overflow? What is the maximum delay experienced by any stream? When multiple streams are being processed by the architecture, does there exist schedulers for the PEs, such that the delay experienced by a given stream is less than a predefined upper bound? These and other related questions naturally arise in the system-level design of (hardware and software) architectures of stream processors. But there are several reasons why these questions turn out to be difficult for many applications: (i) The arrival process of a stream often tends to be highly bursty [25]. As a result, most of the classical event models (such as periodic and sporadic) are either not appropriate, or tend to be overly pessimistic. (ii) The processing time required for data items in a stream can vary considerably. For applications like video processing, the ratio of the worst-case and the average load on a processor can be as high as a factor of 10 [22]. (iii) The input-output rates of a task can also vary, i.e. a single data item consumed by a task can result in multiple (processed) data items at the output and vice versa.

A Motivating Example: Figure 1 shows an MPEG-2 decoder application, which has been partitioned and mapped onto two PEs. The first PE runs the VLD and IQ tasks, while the second runs the IDCT and MC tasks. The (coded)



VLD: Variable Length Decoding IDCT: Inverse Discrete Cosine Transform
IQ: Inverse Quantization MC: Motion Compensation

Figure 1. An MPEG-2 decoder application, partitioned and mapped onto two PEs.

input bitstream enters this system at a constant rate and gets stored in the input buffer. This buffer is read by the first PE, whose output is a stream of partially decoded macroblocks which are written into the buffer b. This buffer is read by the second PE and the resulting stream of fully decoded macroblocks is written into a playout buffer B. Finally, B is read by the output video device at a pre-specified constant rate. We are interested in analyzing the part of the architecture enclosed by the rectangular box. More specifically, given the rate at which the stream of partially decoded macroblocks is written into the buffer b, the clock frequency with which the second PE is being run, and the rate at which the B is being read, we would like to verify that neither buffer overflows and that the buffer B does not underflow. One can also ask: given the sizes of b and B, what is the minimum clock frequency with which the PE can be run?

Although the coded bitstream enters this system at a constant rate, the number of bits consumed by the first PE to generate each partially decoded macroblock varies because of the VLD task. Further, the number of processor cycles required to produce each such partially decoded macroblock is also not constant. Therefore, when the first PE runs at a constant clock frequency, the stream of partially decoded macroblocks that gets written into the buffer b will be highly bursty. Further, each partially decoded macroblock also requires a variable number of processor cycles to get processed by the IDCT + MC tasks. Therefore, to answer the above questions it is necessary to realistically characterize this burstiness in the arrival pattern of the stream and also the variability in its processor cycle requirements.

A model for specifying timing constraints associated with such streams has been recently studied in [7, 15, 19, 27]. The basic idea is to specify upper and lower bounds on the number of data items or events that can arrive within any specified length of time. Suppose a stream is composed of a sequence of data "items" where an item might be a bit, a partially decoded macroblock, or a fully decoded macroblock, depending on which stage in the architecture it is in. Let $x(t)$ denote the number of items that arrive at the buffer b during the time interval $[0, t]$. We say that this arrival process is bounded by the *arrival curve* $\alpha = (\alpha^l, \alpha^u)$ if the following inequalities hold.

$$\alpha^l(\Delta) \leq x(t+\Delta) - x(t) \leq \alpha^u(\Delta), \ \forall t, \Delta \geq 0$$

$\alpha^l(\Delta)$ and $\alpha^u(\Delta)$ are therefore lower and upper bounds on the number of items that can arrive within *any* time interval of length Δ. Typically, such bounds would be specified for a finite number of time interval lengths $\Delta_1, \ldots, \Delta_k$ and such a specification would capture a *class* of arrival patterns $x(t)$ that satisfy the following inequalities.

$$\alpha^l(\Delta_i) \leq x(t+\Delta_i) - x(t) \leq \alpha^u(\Delta_i), \ \forall t \geq 0, \ i = 1, \ldots, k$$

Similarly, to model the variable processing requirements of the data items, a *service curve* $\beta = (\beta^l, \beta^u)$ specifies lower and upper bounds on the number of items that can be processed by the PE within any time interval of a specified length. Let $c(t)$ denote the number of items that can be processed during the time interval $[0, t]$. Such a *service* is bounded by $\beta = (\beta^l, \beta^u)$ if the following inequalities hold.

$$\beta^l(\Delta) \leq c(t+\Delta) - c(t) \leq \beta^u(\Delta), \ \forall t, \Delta \geq 0$$

Again, such a bound on the service would typically be specified for a finite number of time interval lengths. Now, given the bounds α on the arrival process of a stream and the service curve β offered by a PE, [7] presented a calculus using which it is possible to compute (i) the maximum number of data items that can be backlogged at the input of the PE, which is a measure of the minimum buffer size required (e.g. the size of b in Figure 1), (ii) the maximum delay that can be suffered by any item, (iii) bounds on the timing properties of the processed stream (again in terms of an arrival curve), and (iv) bounds on the *remaining* service (in terms of a service curve).

For example,

$$backlog \leq \sup_{\Delta \geq 0} \{\alpha^u(\Delta) - \beta^l(\Delta)\} \quad (1)$$

$$delay \leq \sup_{t \geq 0} \{ \inf_{\Delta \geq 0} \{\Delta : \alpha^u(t) \leq \beta^l(t+\Delta)\}\} \quad (2)$$

Bounds on the timing properties of the processed stream and the remaining service can also be computed similarly [7]. As a stream is processed by multiple PEs, these bounds get successively transformed and as a result the timing properties of the fully processed stream can be computed in a compositional manner.

However, the calculus presented [7] is purely functional and can not model setups where the processing of a stream depends on the *state* of the system. As an example, in Figure 1 the second PE might implement a blocking write for the buffer B (i.e. the PE stalls if B is full). Hence, the service offered by this PE not only depends on the service curve β but also on the *state* of the buffer B. Hence, Eqns. (1) and (2) can no longer be used and the maximum backlog in b and the delay experienced by the stream would also depend on the size of the buffer B and the consumption process from this buffer. Another example is when the *amount* of service offered by the PE depends on the fill level of one or more buffers. Finally the arrival pattern itself might be history-dependent due to the behavior of the external components generating them.

Our Contribution: The main contribution of this paper is an automata-based model called *Event Count Automata* (ECA) using which flexible state-based specifications of arrival and service patterns can be obtained. In addition, they can be used to model and analyze setups where the processing of a stream depends on the state of the system. For example, the service provided by a PE might depend on the state of the buffer or on the arrival process of the stream. Similarly, the arrival process of the stream might itself depend on factors such as the buffer fill level and the consumption process of the stream by an output device, which can generate back-pressures in networks to constrain the arrival pattern. Apart from the ability to model such architectural features, important analysis problems that a system designer will be interested in can now be formulated as standard verification questions.

In addition, ECAs can naturally model a wide variety of protocols and scheduling policies. In contrast, even standard scheduling policies such as EDF and FCFS are difficult to capture within the framework of the calculus developed in [7, 19, 27]. As a result, the computed bounds on the maximum buffer fill, delay, timing properties of the processed stream, etc., will not be tight.

Timed automata [4] have been used to specify the arrival patterns of tasks so that schedulability analysis can be performed using the zone-based reachability analysis techniques for timed automata [20, 16]. Further, timed automata has also been used for task graph and job-shop scheduling problems [1, 2]. However, for many of the analysis questions thrown up by stream processing applications, the *exact* arrival time of the data items is not *relevant*. Rather, it is the *number* of items that can arrive within a specified interval of time is what is of relevance, and this is precisely what is recorded by ECAs. That said, ECAs do bear a strong resemblance at the syntactic level to timed and in fact (rectangular) hybrid automata [4, 13]. However, at the semantic level ECAs are quite different and much simpler. This will become clear once we explain in more detail our model.

In the next section we present ECAs and its basic theory. In Section 3 we model multiprocessor architectures as networks of ECAs. We then show how such networks can be syntactically converted to Colored Petri Nets for the purpose of carrying out analysis. This is followed by an experimental case study in Section 4 to illustrate the usefulness of the model. Finally, in the concluding section, we outline several fruitful lines of future work.

2 Event Count Automata

An Event Count Automaton (ECA) captures arrival patterns by counting the number of data items that arrive in a unit interval of time[1]. This count is used to update the *count variables* associated with the automaton. Guards based on

[1]We assume a suitable granularity of time has been fixed.

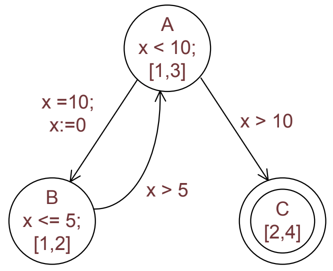


Figure 2. An example ECA.

the values of these count variables can be used to determine at which discrete time instances should the automaton take a transition. In this paper the guards will be formed as conjunctions of atomic predicates of the form $x\#c$ where x is a count variable, $\# \in \{<, \leq, >, \geq\}$ and c is an integer constant. As a part of a transition -assumed to be instantaneous- some of the count variables can be reset to 0. Thus at any given instant, the value of a count variable will record the number of items that have arrived since this count variable was last reset. We also specify for each state of the automaton, a rate interval $[l, u]$ with $l \leq u$. This specifies that in the current mode, in each unit interval of time, at least l and at most u items will arrive. Finally, we associate a state invariant predicate with each state to constrain the values of the count variables for which the automaton is allowed to stay in the state.

An example of an ECA is shown in Figure 2 with just one count variable x. The arrival pattern represented by this automaton occurs in three modes. In the initial A-mode, at least 1 and at most 3 items will arrive in each unit of time. At any time instant, if the total number of items that have arrived exceeds 9 then the mode B is entered if the current count is exactly 10 or the mode C is entered if the current count exceeds 10. In the mode B after at most 5 units of time and not earlier than 3 units of time, the mode A is re-entered (follows from the minimum/maximum number of items that can arrive at each mode). Once the mode C is entered it is never left again. Thus $(A, 0) \dashrightarrow (A, 3) \dashrightarrow (A, 5) \dashrightarrow (A, 7) \dashrightarrow (A, 10) \rightarrow (B, 0) \dashrightarrow (B, 2) \dashrightarrow (B, 4) \dashrightarrow (B, 6) \rightarrow (A, 6) \dashrightarrow (A, 9) \dashrightarrow (A, 11) \rightarrow (C, 11) \dashrightarrow (C, 15)$ is a possible run of the automaton representing the arrival pattern $3\ 2\ 2\ 3\ 2\ 2\ 2\ 3\ 2\ 4$. As this example suggests, we do not make use of action labels to model external events triggered by interactions with reactive components. We shall say more about this in the concluding section.

2.1 The Basic Theory

An ECA is a tuple $\mathcal{A} = (S, s_{in}, X, Inv, R, \rightarrow, F)$ where

- S is a set of control (mode) states and $s_{in} \in S$ is the initial state.
- X is a set of count variables.

- Inv is a function, which assigns to each state s, an invariant constraint $Inv(s) \in \Phi(X)$ where $\Phi(X)$ is the set of constraints over X given by:

  $$\Phi(X) = x \leq c \mid x < c \mid x \geq c \mid x > c \mid \varphi_1 \wedge \varphi_2$$

  Here, and in what follows, c ranges over non-negative integers.

- ρ is a rate function, which assigns to each state $s \in S$ an interval $\rho(s) = [l, u]$ that represents the lower and upper bounds on the arrival rate of events when the system is in state s.

- $\rightarrow \subseteq S \times \Phi(X) \times 2^X \times S$ is a transition relation where, as usual, 2^X is the set of subsets of X Further, we require that $s \neq s'$ for each transition (s, φ, Z, s').

- $F \subseteq S$ is a set of final states.

An event count automaton describes arrival patterns of the form $n_1\ n_2\ ...\ n_k$ with n_i denotes the number of items arriving during the interval $[i-1, i)$. At time i the automaton instantaneously makes a move. This could consist of it staying put in its current state or taking a transition and moving to a new state. As a part of each such move, the automaton signals the number of items that have arrived in the previous unit time interval. This dynamics is captured by the automaton $TS_{\mathcal{A}}$ -called the *associated behavioral automaton* - whose states are *configurations* of the form (s, V) where s is a state of $\mathcal{A}$ and V is a *valuation* which assigns a non-negative integer $V(x)$ -its current value- to each count variable in X. The initial configuration is (s_{in}, V_{in}) where V_{in} is *initial* valuation; often, this will be the valuation that assigns 0 to each count variable. If at time instance i, the automaton is in the configuration (s, V) and k items arrive in the next unit time interval, then at time $i+1$, it can make the move $(s, V) \overset{k}{\Rightarrow} (s', V')$ where (s', V') must satisfy the following conditions. For stating these conditions and for later usage, it will be convenient to adopt some notational conventions.

Suppose V is a valuation and φ is a constraint. Then the notion V satisfying φ -denoted $V \models \varphi$- is inductively defined in the expected manner: $V \models x\#c$ iff $V(x)\#c$ with $\# \in \{<, \leq, >, \geq\}$. Further, $V \models \varphi_1 \wedge \varphi_2$ iff $V \models \varphi_1$ and $V \models \varphi_2$. Secondly, for a non-negative integer n, we define $V+n$ to be the valuation given by: $(V+n)(x) = V(x)+n$ for every x. Thus $V+n$ simply increments every count variable by n, starting with the valuation V. Finally, for $Z \subseteq X$, $V \mid Z$ is the valuation given by $(V \mid Z)(x) = 0$ if $x \in Z$. Otherwise, $(V \mid Z)(x) = V(x)$. Thus, $V \mid Z$ resets all the count variables in Z to 0. All other count variables are as assigned by V. The transition relation $\Rightarrow$ can now be defined.

$(s, V) \overset{k}{\Rightarrow} (s', V')$ iff the following conditions are satisfied.

- $l_s \leq k \leq u_s$ where $\rho(s) = [l_s, r_s]$

- If $s \neq s'$ then there exists a transition of the form (s, φ, Z, s') such that $V+n \models \varphi$ and $V' = (V+n) \mid Z$. Furthermore, V' satisfies $Inv(s')$.

- If $s = s'$ then $V' = V+n$ and $V' \models Inv(s)$. Moreover there does *not* exist a transition of the form (s, φ, Z, s'') such that $V+n \models \varphi$ and $V' = (V+n) \mid Z$ with $V' \models Inv(s'')$.

Some remarks are in order. Firstly, if at a state s the number of items that arrive in a unit interval does not fall in the range $\rho(s)$ (i.e. the arrival violates the arrival rate constraint associated with s) then these items are simply discarded. We have imposed this convention mainly for convenience. Through a more sophisticated transition relation and the use of the state invariants we can easily record and detect violations of the rate bounds specified for the states. Secondly, all our states are "urgent" in the sense, if at a configuration, at least one transition leading out of the state is enabled, then one of these transitions *must* be taken. The automaton stays put in its current state only if *none* of the transitions going out of the state is enabled. Again, we choose this route mainly for convenience as also in assuming implicitly that the automaton will never get stuck. Finally, given the "urgent" semantics, state invariants are not really needed but they are very convenient to have around.

To define $TS_{\mathcal{A}}$, let $\rho(s) = [l_s, u_s]$ for each s in S and let $\hat{n}$ be the maximum of the set of integers $\{u_s \mid s \in S\}$. Thus $\hat{n}$ is the maximum number events that can arrive in any mode in the arrival patterns specified by $\mathcal{A}$. Then $TS_{\mathcal{A}} = (S \times VAL, [0, \hat{n}], (s_{in}, V_{zero}), \Rightarrow, \hat{F})$ given by:

- VAL is the set of valuations.

- $[0, \hat{n}] = \{0, 1, \ldots \hat{n}\}$, called the *alphabet* of $\mathcal{A}$.

- $\hat{F} = \{(s, V) \mid s \in F\}$ is the set of final states of $TS_{\mathcal{A}}$.

- S, s_{in}, V_{in} are as in the definition of an ECA and $\Rightarrow$ is as defined above.

We note that VAL is an infinite set and hence $TS_{\mathcal{A}}$ is a potentially infinite state automaton accepting (and rejecting) sequences over the finite alphabet $\{0, 1, \ldots, \hat{n}\}$. To bring this out, let $\sigma = k_1 k_2 \ldots k_m$ be a sequence in $[0, \hat{n}]^*$. Then a run of $\mathcal{A}$ over σ is a sequence $(s_0, V_0) \overset{k_1}{\Rightarrow} (s_1, V_1) \overset{k_2}{\Rightarrow} \ldots (s_m, V_m)$ with $(s_0, V_0) = (s_{in}, V_{in})$. The run is *accepting* if (s_m, V_m) is in $\hat{F}$ (in other words, $s_m \in F$) and σ is said to be accepted by $\mathcal{A}$. It is in this sense, the sequence $3\ 2\ 2\ 3\ 2\ 2\ 2\ 3\ 2\ 4$ is accepted by the automaton of Figure 2, while the sequence $3\ 5$ is *not* accepted by this automaton. We let $\mathrm{L}(\mathcal{A})$ be the set of sequences accepted by $\mathcal{A}$. It represents the arrival patterns specified by the automaton.

The basic theory of ECAs follows from the fact that the associated behavioral automaton of an ECA can be quotiented into a *finite* state automaton which accepts the same set of finite sequences. This can be established by using arguments similar to but much simpler than the ones used for constructing the regional automaton for a timed automaton [4]. Given below are the main details.

Let $\mathcal{A}$ be an ECA and $TS_{\mathcal{A}} = (S \times VAL, [0, \hat{n}], (s_{in}, V_{in}), \Rightarrow, \hat{F})$ be its associated behavioral automaton. For each count variable x, let c_x be the maximum of all the integers that appear in constraints involving the variable x. For instance, in the ECA shown in Figure 2, $c_x = 10$. We say that two valuations V and V' are *equivalent*, denoted as $V \approx V'$, iff for every count variable x, either $V(x) = V'(x)$ or $V(x) > c_x$ and $V'(x) > c_x$. Two configurations (s, V) and (s', V') of $TS_{\mathcal{A}}$ are equivalent -written again for convenience as $(s, V) \approx (s, V')$- iff $s = s'$ and $V \approx V'$. Clearly $\approx$ partitions the infinite set of states of $TS_{\mathcal{A}}$ into a finite number of equivalence classes. Indeed, the number of these classes is bounded by $\mid S \mid \times (c + 1)$ where c is the maximum of all the c_x values (corresponding to the different count variables).

Configurations belonging to an equivalence class exhibit the same behavior in the following sense: Suppose $(s, V) \approx (s, V')$ and $(s, V) \overset{k}{\Rightarrow} (s_1, V_1)$. Then there exists a configuration (s_1, V_2) such that $(s, V') \overset{k}{\Rightarrow} (s_1, V_2)$ and furthermore, $(s_1, V_1) \approx (s_1, V_2)$.

To see this, we first observe that if $V \approx V'$ then $V \models \varphi$ iff $V' \models \varphi$ for *any* constraint φ. This can be easily established by induction on the structure of φ. For instance, suppose φ is an atomic constraint of the form $x \leq c$. If $V \models (x \leq c)$ then $V(x) \leq c$. Clearly $c_x \geq c$ and hence it can not be the case that $V(x) > c_x$. This implies that $V'(x) = V(x)$ since $V \approx V'$. Hence $V'(x) \leq c$, which leads to $V' \models (x \leq c)$. The remaining cases can be disposed off equally easily.

Now let $(s, V) \approx (s', V')$. Then $s = s'$ and $V \approx V'$. Suppose $(s, V) \overset{k}{\Rightarrow} (s_1, V_1)$. Consider first the case $s_1 = s$. Then $V_1 = V + n$ and $V_1 \models Inv(s)$. But then $V \approx V'$ implies $V + n \approx V' + n$ and as observed above $V + n \models Inv(s)$ implies $V' + n \models Inv(s)$. Thus by the definition of the $\Rightarrow$ relation, we have $(s, V') \overset{k}{\Rightarrow} (s, V_2)$ where $V_2 = V' + n$ and $V_1 = V + n \approx V' + n = V_2$.

Next consider the case $s_1 \neq s$. Then there exists a transition of the form (s, φ, Z, s_1) such that $V + n \models \varphi$ and $V_1 = (V + n) \mid Z$. Furthermore, V_1 satisfies $Inv(s_1)$. Set $V_2 = (V' + n) \mid Z$. Using the fact $V \approx V'$, it is easy to verify that $V' + n \models \varphi$, $V_2 \models Inv(s_1)$ and $V_2 \approx V_1$. Thus $(s, V') \overset{k}{\Rightarrow} (s_1, V_2)$ with $(s_1, V_1) \approx (s_1, V_2)$ as required.

We now define the quotiented version of $TS_{\mathcal{A}}$ to be the finite state automaton $REG_{\mathcal{A}} = \langle \hat{S}, \{0, 1, \ldots \hat{n}\}, \Rightarrow, \langle s_{in}, V_{in} \rangle \rangle$ where:

- $\hat{S} = \{\langle s, V \rangle \mid s \in S$ and V is a valuation$\}$ where $\langle s, V \rangle$ is the $\approx$-equivalence class containing (s, V). In other words, $\langle s, V \rangle = \{(s, V') \mid V \approx V'\}$.

- $\langle s, V \rangle \overset{k}{\Rightarrow} \langle s', V' \rangle$ iff there exists (s, V_1) in $\langle s, V \rangle$ and (s', V_2) in $\langle s', V' \rangle$ such that $(s, V_1) \overset{k}{\Rightarrow} (s', V_2)$.

Clearly, $REG_{\mathcal{A}}$ can be effectively constructed using the presentation of $\mathcal{A}$. It should also be clear that the language of finite strings accepted by $REG_{\mathcal{A}}$ is exactly $\mathrm{L}(\mathcal{A})$.

Towards stating our main result concerning ECAs, let Σ be a finite alphabet and $L \subseteq \Sigma^*$. We will say that L is ECA-definable iff there exists an ECA $\mathcal{A}$ such that $\mathrm{L}(\mathcal{A}) = L$. We note that in this case Σ must be of the form $\{0, 1, \ldots, \hat{n}\}$.

Our main result concerning ECAs can now be stated.

Theorem 1

1. *Let Σ be a finite alphabet of the form $\{0, 1, \ldots, \hat{n}\}$ and $L \subseteq \Sigma^*$. Then L is ECA-definable iff L is a regular subset of Σ.*

2. *The class of ECA-definable languages is closed under boolean operations.*

3. *Given an ECA $\mathcal{A}$, one can effectively determine if $\mathrm{L}(\mathcal{A})$ is empty.*

Proof: See [8].

Relationship to Hybrid and Timed Automata: Readers familiar with timed and hybrid automata [4, 3, 13] would have noticed their resemblance to ECAs. Indeed, the behavioral equivalence $\approx$ we defined is inspired by the -much more sophisticated - regional equivalence associated with timed automata. This resemblance however is a purely syntactic one since the clock variables of timed automata are used to record the *times* at which interesting events happen in a real time system whereas the count variables of an ECA merely count the *number* of items that arrive in a unit time interval; the times at which these items arrive within the interval are not of interest and are not recorded.

ECAs Representing Arrival and Service Curves: Arrival and service curves can be systematically represented as ECAs. There is no distinction between an ECA representing an arrival curve and one representing a service curve, apart from the fact that the former bounds the arrival process of data items and the latter bounds the processing of these items. An example will illustrate the main idea. Consider the upper arrival curve shown in Figure 3, along with its ECA representation.

This ECA will change its current state at every time instant. All the states are associated with the state invariant *true*. At the end of the first (unit) time interval, the constraint $x_0 \leq 3$ verifies that at most 3 items have arrived during the previous interval (which is also redundantly ensured by the rate vector $[0, 3]$ associated with the state s_0).

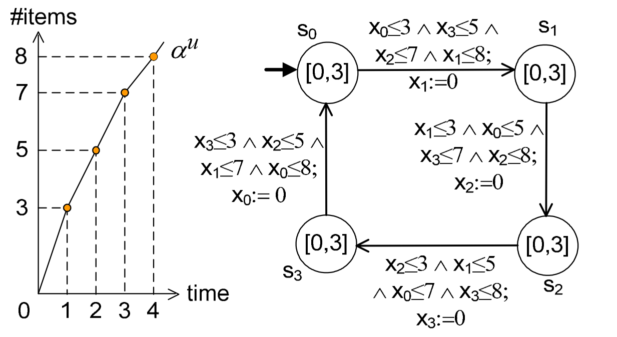

#items
8
7
5
3
0
1 2 3 4
time
α^u
s_0
s_1
s_2
s_3
[0,3]
$x_0 \leq 3 \wedge x_3 \leq 5 \wedge$
$x_2 \leq 7 \wedge x_1 \leq 8;$
$x_1{:=}0$
$x_1 \leq 3 \wedge x_0 \leq 5 \wedge$
$x_3 \leq 7 \wedge x_2 \leq 8;$
$x_2{:=}0$
$x_2 \leq 3 \wedge x_1 \leq 5$
$\wedge x_0 \leq 7 \wedge x_3 \leq 8;$
$x_3{:=}0$
$x_3 \leq 3 \wedge x_2 \leq 5 \wedge$
$x_1 \leq 7 \wedge x_0 \leq 8;$
$x_0{:=}\,0$

Figure 3. An arrival curve and its corresponding ECA.

The constraint $x_0 \leq 5$ associated with the transition from s_1 will ensure that at most 5 items have arrived in the previous two unit intervals. It is easy to see how the four count variables are read from and reset in a cyclic fashion to enforce the constraints specified by the upper arrival curve with four entries (specifying constraints over time intervals of lengths 1 to 4). A service curve (see Section 1) can also be similarly represented using an ECA.

Networks of ECAs: In most stream processing applications, it is useful to compose ECAs with the help of point-to-point buffers. Suppose $\mathcal{A}$ and $\mathcal{A}'$ are two ECAs and B denotes a buffer. $\mathcal{A}$ represents the arrival process of items into B and $\mathcal{A}'$ represents the pattern in which these items are processed by a processor. The idea is that k items are deposited in B whenever the behavioral automaton of $\mathcal{A}$ makes a transitions of the form $(s_1, V_1) \overset{k}{\Rightarrow} (s_2, V_2)$ whereas m items are removed from B whenever the behavioral automaton of $\mathcal{A}'$ makes a transition of the form $(s_1', V_1') \overset{m}{\Rightarrow} (s_2', V_2')$. This complex structure consisting of two ECAs communicating via a buffer can then be described as a Petri net. Indeed, for more complex networks, one can obtain succinct descriptions by using Colored Petri Nets (CPNs) [14] instead of ordinary Petri nets. This will be brought out in more detail in the next section.

3 Modeling Multiprocessor Architectures

We will use networks of ECAs to model stream processing on multiprocessor architectures, where different parts of an application run on different PEs which communicate via buffers (see Figure 1).

Our goal here is to show that it is straightforward to extract an *executable* specification from a network of ECAs and perform analysis using this executable representation. We deploy here Colored Petri Nets (CPNs) to represent a system of ECAs communicating via buffers. Before getting into the details, we wish to emphasize that CPNs are not the only possible vehicle to serve as an executable mechanism for networks of ECAs. We have chosen CPNs mainly because they are supported by a good tool environment including a simulator [9]. We also wish to emphasize that a *global* model of a complex network of ECAs will inevitably lead to state explosion and hence one will have to adapt one or more of the existing techniques for coping with the state explosion problem in order to obtain practically feasible analysis methods. Our goal here however is to lay the groundwork for building such analysis methods backed up by tool support. We discuss this issue in more detail in Section 4.3.

We now wish to show how the CPN representation of a network of ECAs can be constructed and used to answer various questions about the timing properties of the stream processing system under study. Often, these questions will be about quality-of-service constraints associated with these streams. Further, a number of common problems such as optimal buffer sizing, determining optimal schedulers, etc. can also be formulated as standard analysis or verification problems using the CPN. We begin with brief introductory remarks on CPNs.

Colored Petri Nets: Colored Petri Nets (CPNs) are a powerful extension of ordinary Petri nets in which the tokens have attributes associated with them. These attributes can correspond to complex data types and the transitions can encode intricate manipulations of these data types. This is so because, in a CPN, the firing of a transition depends not only on the number of tokens being present on its input places; in addition, the "colors" of the tokens on its input places must also satisfy the constraints imposed by the arc-expressions associated with its input arcs. Similarly, the colors of the tokens placed by a transition on its output places are constrained by the arc-expressions associated with the output arcs of the transition. As a result, CPNs constitute a flexible and powerful graphical notation for describing a variety of applications such as communication protocols, distributed reactive systems, automated production systems and work flow processes.

CPN Tools [9] is a tool kit developed by the CPN Group, University of Aarhus, Denmark. This tool kit provides support for editing, simulating and analyzing CPNs. It has a GUI, that allows users to draw CPNs and declare variables or functions. Besides features such as incremental syntax checking, code generation, etc., the tool kit has a simulator and a state space analysis tool. Using the state space analysis tool, full and partial state spaces can be generated and analyzed. Detailed information concerning the theory, analysis and applications of CPNs including tool support can be found in [14]. Information concerning other simulation and analysis tools for CPNs can be found at: www.informatik.uni-hamburg.de/TGI/PetrNets/Tools/db.html.

CPNs for ECAs: Because of its structure, an ECA can be systematically encoded as a CPN. Furthermore, by exploiting the regular structure of ECAs representing arrival or service curves, such ECAs can be encoded as CPNs which contain only two places and one transition. Examples illustrating the encoding of ECAs as CPNs may be found in [8]. It is crucial to note here that the encoding of an ECA as a CPN follows a purely syntactic construction and does not involve the behavioral automaton describing the semantics of the ECA. Hence, this encoding can be done very efficiently. Further, we show in the next subsection that the composition of different CPNs (each corresponding to an ECA) can also be done in a syntactic fashion.

3.1 Modeling an Architecture as a CPN

We consider the architecture model introduced in Section 1, i.e. multiple PEs process one or more streams in a pipelined fashion and exchange data (partially processed streams) through unidirectional buffers. Let us first consider the single stream case. Let ECA_α be the ECA that bounds the arrival process of a single input stream. ECA_{β_1}, ECA_{β_2}, ..., ECA_{β_n} are the ECAs bounding the service offered by PE$_1$, PE$_2$, ..., PE$_n$, respectively, where n is the number of PEs in the architecture. ECA_c is the ECA representing the consumption of the processed stream. b denotes the input buffer, B_i the intermediate buffer between PE$_i$ and PE$_{i+1}$, and B_n is the output buffer between PE$_n$ and the output device. For simplicity, we will restrict our discussion to the case of only two PEs, i.e. $n = 2$. The extension to multiple PEs is trivial.

The CPNs encoding the different ECAs can now be composed into a single CPN (representing the full architecture) using additional places and arcs. This construction is as follows. We create three places of type `integer`, each of which represents a buffer in the architecture (recall that we have two processors, and hence three buffers). For each transition in a CPN, we add an arc between this transition and some of the newly added places (representing the buffers). Some of these arcs encode the arrival of data items into a buffer and the others encode the consumption (or processing) of these items. Associated with each outgoing (incoming) arc from (to) each of these places is an integer variable whose value represents the number of items in the corresponding buffer before (after) these items arrive or are taken from the buffer. Hence, the value of the token in the place corresponding to a buffer denotes the number of items in the buffer at the current time. In addition, we use another place to control the order in which data items in the stream arrive and the order in which they are processed by the different PEs. Arcs between this place and the transitions of the different CPNs are added and are guarded in a manner that respects the sequential order of the arrival and processing of the items in the stream. In other words, this is to ensure that an item is processed by PE_2 only after it has been processed by PE_1.

Figure 5 shows a CPN, called *CP*, which is composed of four communicating CPNs. The ECAs corresponding to

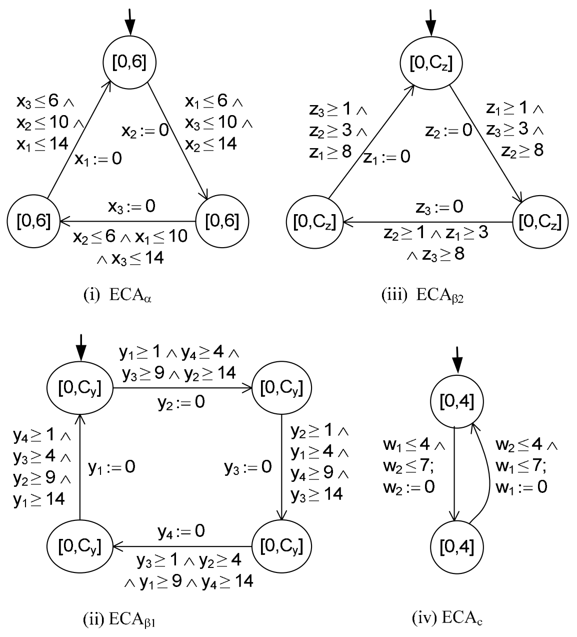

[0,6]
$x_3 \le 6 \wedge$ $x_2 \le 10 \wedge$ $x_1 \le 14$
$x_1 := 0$
$x_2 := 0$
$x_1 \le 6 \wedge$ $x_3 \le 10 \wedge$ $x_2 \le 14$
[0,6]
$x_3 := 0$
[0,6]
$x_2 \le 6 \wedge x_1 \le 10$ $\wedge x_3 \le 14$
(i) ECA$_\alpha$
[0,C$_z$]
$z_3 \ge 1 \wedge$ $z_2 \ge 3 \wedge$ $z_1 \ge 8$
$z_1 := 0$
$z_2 := 0$
$z_1 \ge 1 \wedge$ $z_3 \ge 3 \wedge$ $z_2 \ge 8$
[0,C$_z$]
$z_3 := 0$
[0,C$_z$]
$z_2 \ge 1 \wedge z_1 \ge 3$ $\wedge z_3 \ge 8$
(iii) ECA$_{\beta 2}$
[0,C$_y$]
$y_1 \ge 1 \wedge y_4 \ge 4 \wedge$ $y_3 \ge 9 \wedge y_2 \ge 14$
$y_2 := 0$
[0,C$_y$]
$y_4 \ge 1 \wedge$ $y_3 \ge 4 \wedge$ $y_2 \ge 9 \wedge$ $y_1 \ge 14$
$y_1 := 0$
$y_3 := 0$
$y_2 \ge 1 \wedge$ $y_1 \ge 4 \wedge$ $y_4 \ge 9 \wedge$ $y_3 \ge 14$
[0,C$_y$]
$y_4 := 0$
[0,C$_y$]
$y_3 \ge 1 \wedge y_2 \ge 4$ $\wedge y_1 \ge 9 \wedge y_4 \ge 14$
(ii) ECA$_{\beta 1}$
[0,4]
$w_1 \le 4 \wedge$ $w_2 \le 7;$ $w_2 := 0$
$w_2 \le 4 \wedge$ $w_1 \le 7;$ $w_1 := 0$
[0,4]
(iv) ECA$_c$

Figure 4. ECAs describing the arrival process of a stream, its processing (service provided) by two PEs and finally its consumption by an output device (see Figure 1). The initial valuations of the count variables for ECA_α, ECA_{β_1}, ECA_{β_2} and ECA_c are $(0,0,0)$, $(0,14,14,14)$, $(0,8,8)$ and $(0,0)$ respectively.

these four CPNs are shown in Figure 4. *CP* models the complete architecture (consisting of two PEs) specified by the four ECAs given in Figure 4.

From the construction, it can be shown that the execution of *CP* is a sequence of steps, each of which begins with a transition in CP_α, followed by a transition in CP_{β_1}, then a transition in CP_{β_2} and finally a transition in CP_c being fired. CP_α, CP_{β_1}, CP_{β_2} and CP_c are the CPNs encoding ECA_α, ECA_{β_1}, ECA_{β_2} and ECA_c respectively (shown in Figure 4). One execution step in CP corresponds to the execution/processing that is carried out by the architecture in one unit of time. As a result, the status of *CP* after each execution step represents the (stable) status of the architecture after it has finished the equivalent of one time unit of execution. In particular, suppose the system starts at time $t = 0$, then at time $t = i$, *CP* finishes i execution steps and the values of the tokens in p_α, p_β, p_c are the number of items left in the buffers b, B_1 and B_2 respectively. p_α, p_β and p_c are denoted by p_a, p_b and p_c respectively in Figure 5.

3.2 Analysis

The resulting *CP* described above can be used to analyze various properties of a multiprocessor architecture in which the processing of a stream is described using a set of ECAs.

Computing Maximum Buffer Fill Levels: The maximum fill level of a buffer can be obtained by computing the

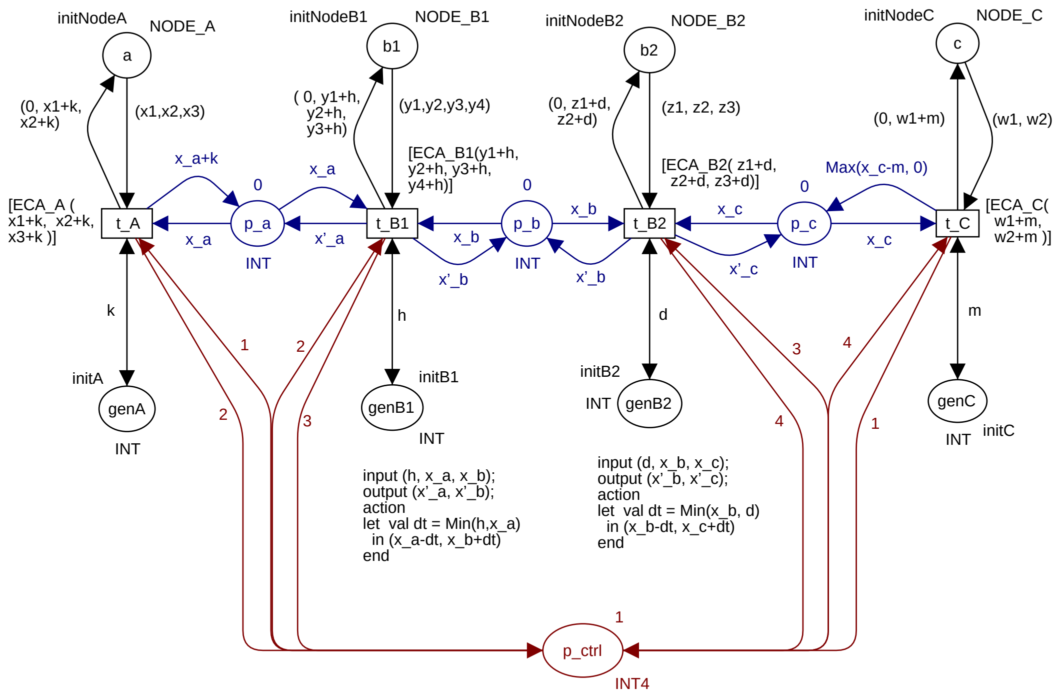


Figure 5. The Colored Petri Net corresponding to the specification in Figure 4.

upper bound on the value of the token in the corresponding place in *CP*. In particular, the fill levels of b, B_1, B_2 are the values of the tokens in the places p_α, p_β and p_c at the end of each execution step. The upper bound on the value of a token in a place in a CPN can be checked by examining the *Best Upper Multi-set Bounds* after running the state space analysis routine on the CPN [9].

Given a pre-specified buffer size, from the computed maximum buffer fill level, it is possible to verify whether the buffer might overflow. However, in this case, it is not required to compute the upper bound on the value of the token in the place corresponding to the buffer in the CPN. Instead, a more efficient approach would be to add a new place into the CPN to represent an *error* state. Whenever the value of the token exceeds the specified buffer size, a token is put in this new place. Hence, checking for buffer overflow can now be formulated as reachability analysis problem instead of resorting to a full state space exploration.

It is also possible to model buffers that implement blocking write. A PE writing into such a buffer stalls when the buffer is full. To model such a buffer we extend the above CPN with a new transition which is fired after the CPN finishes one execution step. This transition checks if the value of the token in the place representing the buffer exceeds the buffer size. If it does, then the value of this token is readjusted to the maximum buffer size and the difference between the original value and this buffer size is added to the downstream buffers. This process is then cascaded downstream.

Similarly, it is also possible to model PEs which adjust their service depending on the state of the system, e.g. the fill levels of the buffers or the rate at which data items arrive at the input. Finally, we would like to point out that so far we have only been concerned with the processing of a single stream. However, our framework can model the processing of multiple streams in a straightforward manner, the details of which may be found in [8].

4 Experimental Case Study

In this section we present a case study to illustrate the practicality of the ECA model described so far. Towards this, recall the MPEG-2 decoder application introduced in Section 1 (see Figure 1). This application is partitioned and mapped onto two processing elements (PEs). The input to the first PE is a constant bitrate (coded) bitstream, which is processed by the VLD and IQ tasks. This results in a stream of partially decoded macroblocks being written into a buffer b. The second PE reads this buffer and runs the IDCT and MC task on each of these partially decoded macroblocks. It then writes the resulting stream of fully decoded macroblocks into the playout buffer B which is read by the output video device at a pre-specified constant rate. Because of the variability in the execution times of the VLD + IQ tasks and also the variability in the number of bits constituting each partially decoded macroblock, the stream being written into b is highly bursty. We assume that the timing properties of this stream is specified by an *arrival curve* $\alpha^u(\Delta)$ (see Section 1). The execution requirements of the IDCT + MC tasks running on the second PE is also highly variable. We assume that this variability is captured by a *service curve* $\beta_f^l(\Delta)$ which denotes the *minimum* number of partially decoded macroblocks that can be processed by the PE within *any* time interval of length Δ, when it is run at a clock frequency f. Further, we assume that this PE implements a blocking write for the buffer B, i.e. it stalls when B is full.

Now, given $\alpha^u(\Delta)$, $\beta_f^l(\Delta)$, the size of B and the rate at which macroblocks are read out from B by the output device, we would like to compute the maximum fill-level of b (which is a measure of the minimum required size of b). Note that the *effective service* offered by the second PE depends on $\beta_f^l(\Delta)$ and the *state* (or fill level) of B.

4.1 Obtaining $\alpha^u(\Delta)$ and $\beta_f^l(\Delta)$

In general, there are several ways in which $\alpha^u(\Delta)$ and $\beta_f^l(\Delta)$ may be obtained. In many cases, it might be possible to analytically derive these functions (constraints) from a formal specification of the system and its environment. In other cases a simulation- and trace-based analysis might be easier. For our problem, program analysis techniques along with an appropriate modeling of the PE's microarchitecture [17] could have been used to derive these functions. However, for simplicity, we adopted the latter (i.e. simulation-based) approach. Towards this, we collected execution traces of the different tasks by simulating their execution using a model of the PE's microarchitecture. We then analyzed these traces to derive the functions $\alpha^u(\Delta)$ and $\beta_f^l(\Delta)$.

To obtain the execution traces, we used a customized version of the SimpleScalar instruction set simulator [5]. The arrival curve $\alpha^u(\Delta)$ of the stream at the input of the buffer b can be obtained by measuring the execution demands of the VLD and IQ tasks for each macroblock in a video sequence and by taking into account (i) the constant arrival rate of the compressed bit stream at the input of the first PE, and (ii) the number of bits allocated to encode each macroblock in the stream. Following this procedure, we first obtained a function $x(t)$, where $x(t)$ denotes the number of macroblocks arriving at b during the time interval $[0, t]$. From $x(t)$ we then derived $\alpha^u(\Delta)$, following the definition of an arrival curve in Section 1. Typically, a designer would use multiple *representative* video clips to derive $\alpha^u(\Delta)$, such that all these clips satisfy the constraint imposed by $\alpha^u(\Delta)$.

The function $\beta_f^l(\Delta)$ can also be derived similarly. We collected a trace of the execution demands for the pair of tasks IDCT and MC executing on the second PE. For each chosen processor frequency f, we then analyzed this trace to derive the function $\beta_f^l(\Delta)$ using a method similar to the one used for deriving $\alpha^u(\Delta)$ (see also the definition of a service curve in Section 1).

The decoded macroblocks are read out from the playout buffer B at a pre-specified constant rate, which can also be modeled by an arrival (or consumption) curve, say α_c.

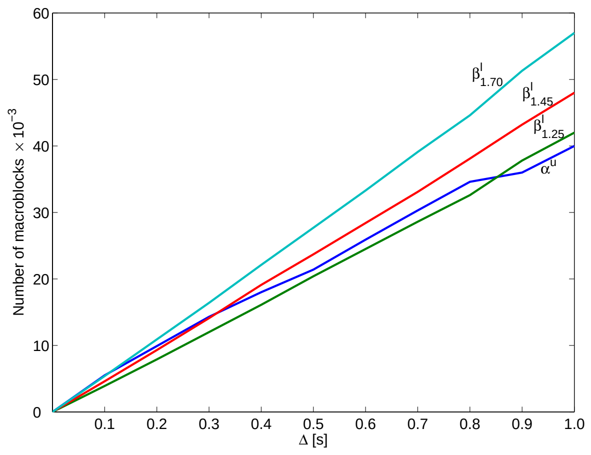


Figure 6. Arrival curve and service curves corresponding to different processor frequencies.

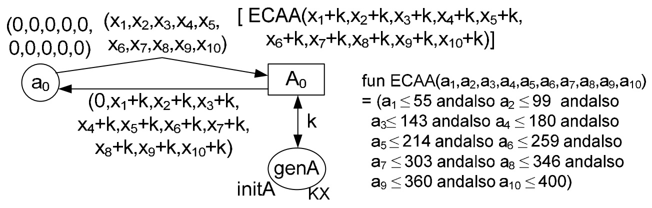
However, we do not need to resort to simulation to obtain α_c, for obvious reasons.

Figure 6 shows the arrival curve $\alpha^u(\Delta)$ and the service curves $\beta_f^l(\Delta)$ for three different PE frequency values, 1.70 GHz, 1.45 GHz and 1.25 GHz, for a 4 Mbps MPEG-2 video clip. From this figure, note that over time intervals of length less than 0.1 secs, the number of macroblocks arriving at b is larger than the minimum number of macroblocks that are guaranteed to be processed within this time interval, for all the three frequency values. However, for larger time intervals (beyond 0.9 secs) the number of macroblocks that are guaranteed to be processed is more than what may arrive. This "switch-over" happens at different time interval lengths for different processor frequencies. Clearly, the smaller the value of f, the larger is the minimum required size of b. However, even a frequency of 1.25 GHz is sufficient to ensure that the required size of b is bounded. The consumption rate of the decoded macroblocks from the playout buffer was set to 4×10^4 macroblocks per second, irrespective of the frequency of the PE.

4.2 Using ECAs

To represent the arrival and the service curves described above, using ECAs, we first need to fix a suitable granularity of time. This choice is primarily determined by two factors: (i) we would like to have a reasonably small number of constraints $\alpha^u(\Delta_1), \ldots, \alpha^u(\Delta_k)$ defining the arrival (or the service) curve, but at the same time (ii) we would like to capture all the variability represented by these curves. For our experiments, we defined one time unit (i.e. $\Delta = 1$) to be equal to 0.1 sec. We then represented the arrival and the service curves (see Figure 6) using 10 constraints defined over time intervals of length $1, \ldots, 10$.

The resulting ECA consists of 10 states, $s_1, \ldots, s_{10}$, with s_1 the initial state, and also 10 count variables, $x_1, \ldots, x_{10}$. At any integer point of time, each x_i represents the number of items (partially decoded macroblocks


(0,0,0,0,0, 0,0,0,0,0)
(x1,x2,x3,x4,x5, x6,x7,x8,x9,x10)
[ECAA(x1+k,x2+k,x3+k,x4+k,x5+k, x6+k,x7+k,x8+k,x9+k,x10+k)]
a0
A0
(0,x1+k,x2+k,x3+k, x4+k,x5+k,x6+k,x7+k, x8+k,x9+k,x10+k)
k
genA
initA
KX

```
fun ECAA(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10)
= (a1 ≤ 55 andalso a2 ≤ 99  andalso
  a3 ≤ 143 andalso a4 ≤ 180 andalso
  a5 ≤ 214 andalso a6 ≤ 259 andalso
  a7 ≤ 303 andalso a8 ≤ 346 andalso
  a9 ≤ 360 andalso a10 ≤ 400)
```

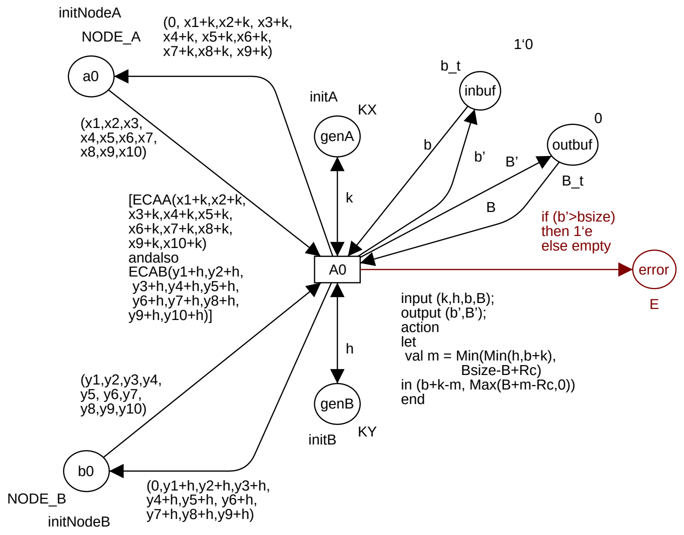
Figure 7. The CPN corresponding to the ECA specifying α^u in Figure 6.

in this case) that have arrived in the preceding time interval of length k, for some value of k between 1 and 10. The guards associated with each transition in the ECA is a conjunction of 10 constraints of the form $x_i \leq \alpha(k)$, where $1 \leq i \leq 10, 1 \leq k \leq 10$. These constraints ensure that the arrival pattern of the data items that arrived at the system till the current time is bounded by α^u. Finally, each state in the ECA is annotated with a range $[0, \alpha^u(1)]$, representing the rate at which items can arrive when the ECA is in that state. The CPN corresponding to this ECA is shown in Figure 7.

The ECA representing a service curve β^l_f and the corresponding CPN can be similarly obtained. We do not show them here due to space constraints. Now, following the discussion in Section 3.1, the CPNs corresponding to these arrival and service curves were composed into a single CPN which is shown in Figure 8. Further, to simplify the representation, the transitions in the two CPNs have been combined into a single transition in this composed CPN. This CPN models the part of the architecture enclosed within the rectangular box in Figure 1. Since the playout buffer is read out at a *constant* rate by the output device, this is not modeled using a separate ECA. Instead this is directly encoded in the composed CPN (where this constant rate is denoted as `Rc` in the code region of the transition in the CPN). Note that the places `inbuf` and `outbuf` model the input buffer and the playout buffer respectively. These places are associated with two integer variables which denote the number of data items in these buffers at any point in time. At each time step, these variables are updated depending on the number of items produced/consumed by `genA`, `genB` and the constant rate of consumption by the output device. The fact that the processor implements a blocking write for the playout buffer is also coded in the code region of the transition in the CPN.

When the size of the input buffer is pre-specified, an item is put in the place `error` whenever the value of the integer variable associated with the place `inbuf` exceeds this pre-specified size. Thus in this case, the problem of checking whether the input buffer can ever overflow can be formulated as a reachability analysis question, i.e. whether there exists a reachable marking M at which $M(\texttt{error}) > 0$.

We used the CPN Tools tool kit [9] to specify the CPN in Figure 8 and ran the state space analysis routine. From the generated state space exploration result we computed


initNodeA
NODE_A
a0
(0, x1+k,x2+k, x3+k, x4+k, x5+k,x6+k, x7+k,x8+k, x9+k)
(x1,x2,x3, x4,x5,x6,x7, x8,x9,x10)
[ECAA(x1+k,x2+k, x3+k,x4+k,x5+k, x6+k,x7+k,x8+k, x9+k,x10+k) andalso ECAB(y1+h,y2+h, y3+h,y4+h,y5+h, y6+h,y7+h,y8+h, y9+h,y10+h)]
initA
KX
genA
k
1'0
b_t
inbuf
b
b'
B'
outbuf
0
B_t
B
if (b'>bsize) then 1'e else empty
A0
error
E
h
genB
initB
KY
(y1,y2,y3,y4, y5, y6,y7, y8,y9,y10)
b0
NODE_B
initNodeB
(0,y1+h,y2+h,y3+h, y4+h,y5+h, y6+h, y7+h,y8+h,y9+h)

```
input (k,h,b,B);
output (b',B');
action
let
 val m = Min(Min(h,b+k),
          Bsize-B+Rc)
 in (b+k-m, Max(B+m-Rc,0))
end
```

Figure 8. A CPN modeling the part of the architecture enclosed within the rectangular box in Figure 1.

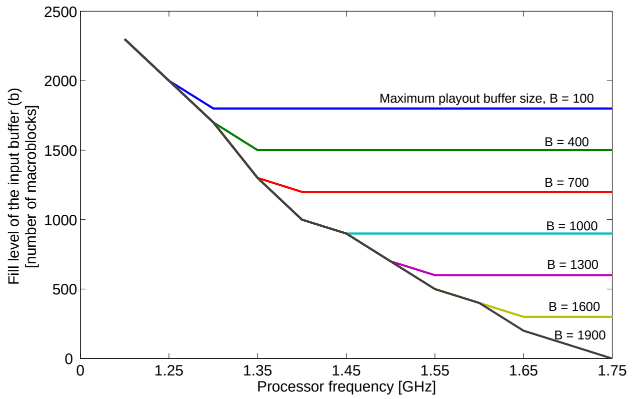


Figure 9. Fill levels of the input buffer for different maximum playout buffer sizes (in macroblocks) and different processor clock frequencies.

the upper bound on the integer variable associated with the place `inbuf` for different sizes of the playout buffer. These results are shown in Figure 9.

This figure shows that for any fixed playout buffer size, as the processor frequency is increased, the fill level of the input buffer decreases. This is because the incoming data items in the input buffer are now processed faster and are written out into the playout buffer (which is read by the output device at a constant rate). However, beyond a certain processor frequency, the maximum fill level of the input buffer stabilizes; this happens exactly when the playout buffer completely fills up (because the processor implements blocking write for this buffer). As a result, the maximum fill level of the input buffer decreases as the playout buffer size is increased. Note that all of these are a consequence of the burstiness in the arrival pattern of the stream and because its execution requirements are highly variable.

However, the "long-term" arrival rate of the stream, the "long-term" rate at which macroblocks are processed by the PE, and the "long-term" consumption rate from the playout buffer are all equal. The input and the playout buffers only capture the burstiness in the stream, which our model accurately captures. Had the arrival pattern of the stream been modeled using periodic/sporadic event models with constant execution time for every data item, then these effects could not be seen.

Based on these results it is possible to determine the optimal processor frequency and sizes of the playout and the input buffers, given the size of the available memory. Running the processor at a frequency higher than this optimal frequency reduces its utilization rate. Our results match well with those obtained using simulation, which was based on a transaction level model of the system architecture written in SystemC and the PE modeled using a customized version of the SimpleScalar instruction set simulator. We again wish to point out that obtaining the results shown in Figure 9 using a purely simulation-based approach is prohibitively expensive even for video clips of very short duration. Further, given that we are interested in an *upper bound* on the buffer fill level, results obtained from simulation can not provide formal guarantees.

4.3 Efficiency Issues: State Space Representation & Analysis Methods

The state spaces of the CPNs we extract from our ECA networks will be very large for most practical problems. Hence, an exhaustive exploration of such state spaces, as we did in our case study, will not be a viable option. However, a variety of techniques are available for coping with large state spaces. Since the emphasis of this paper has been on introducing a new framework, it is not reasonable to explore these techniques here in detail. Below we briefly discuss some such techniques that are particularly relevant for our setting. Our aim is to convince the reader that the framework we presented can be used for realistic problems and that there are a number of reasonable implementation options that would be interesting to study as a part of future work.

Special Data Structures for Symbolic State Space Representation: OBDDs [6] have turned out to be extremely useful for representing large state spaces. Our state spaces however are generated by ECAs and BDDs are not geared towards taking advantage of the additional structure on offer here. Tuples of integers (valuations) and constraints based on these tuples are the specific features we need to handle. We expect specialized forms of BDDs and in particular, the data structure called Interval Decision Diagrams (IDDs) [24] to play a useful role in this context. This is so, since IDDs are particularly suited for succinctly representing multi-valued functions whose domains consist of intervals of integers and more importantly, where the effect of the functions themselves can be described in terms of integer intervals. Hence IDDs -and the accompanying Interval Mapping Diagrams [24]- hold considerable promise for efficiently representing the states of the behavioral automata associated with ECAs as well networks of such automata communicating via bounded buffers.

State Space Invariants: Many of the behavioral properties we wish to determine can be stated as invariant properties: For instance, is the number of items contained in the buffer b bounded by the constant K *for all* reachable configurations? There is a considerable body of work available for efficiently generating invariants for the state spaces of hybrid automata; see for instance a recent work [21] and the references therein. Simplified versions of these techniques will be applicable for verifying a variety of important behavioral properties for ECAs too.

Structural Invariants: There is a also a large body of work for computing the so called S-invariants and T-invariants in the setting of CPNs [14]. The crucial feature of these invariants is that they are *structural*. In other words, they depend on the way places and transitions are connected to each other and the functions associated with the transitions that manipulate the color sets. Consequently, methods of linear algebra -and not state space exploration techniques- can be used to compute these invariants. Admittedly not all the interesting invariant properties of the state space can be caught by S-invariants and T-invariants. Nevertheless, many of the properties of interest in our setting -especially those concerning buffer overflows and underflows- can be cast in the language of S-invariants. Hence, S-invariant techniques specialized to CPNs that are associated with ECA networks will be a powerful analysis tool.

5 Concluding Remarks

In this paper we presented a state-based framework for modeling and analyzing stream processing applications and architectures. It is geared towards accurately modeling the burstiness in data streams associated with applications such as multimedia processing, as well as the variability in the execution requirements associated with such streams. Being state-based, this framework allows for the modeling of different protocols and scheduling policies in a straightforward way. This is not possible in some of the recently proposed frameworks [18, 19, 27] which have inspired our work.

There are several interesting threads that stem from this work. First, it would be worthwhile to explore the state space representation techniques outlined in Section 4.3 in the context of our framework. Second, recall that we model a multiprocessor system using a single global/composed CPN, which we then use for analysis. This can blow up the resulting state space that needs to be explored during the analysis. To avoid this, it might be possible to use composition techniques similar to those in [18, 19, 27], but adapted to our state-based setting. Given the arrival and the service ECAs associated with a PE, the goal is to compute the ECA representing the timing properties of the *processed* stream and the ECA corresponding to the *remaining* service. These then serve as inputs to subsequent processing of the stream by the next PE, and the processing of a second stream by the same PE. Third, we would also like to model the processing of multiple streams, explore scheduling issues in more detail and also model quality-of-service constraints (e.g. only a certain fraction of data items belonging to a stream are required to meet their deadlines). Finally, we can add action labels to the transitions of an ECA. These labels can be used to form synchronized products of ECAs to capture stream processing components interacting with event-based reactive components as suggested in [12].

References

[1] Y. Abdeddaïm, A. Kerbaa, and O. Maler. Task graph scheduling using timed automata. In *IPDPS*, 2003.

[2] Y. Abdeddaïm and O. Maler. Job-shop scheduling using timed automata. In *CAV*, 2001.

[3] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems, LNCS 736*, 1993.

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

[5] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[6] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[7] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *6th Design, Automation and Test in Europe (DATE)*, Munich, Germany, February 2003.

[8] S. Chakraborty, L.T.X. Phan, and P.S. Thiagarajan. Event count automata: A state-based model for stream processing systems, 2005. www.comp.nus.edu.sg/~samarjit/psfiles/ecaTR.ps.

[9] CPN Tools: Computer tool for coloured petri nets. http://wiki.daimi.au.dk/cpntools/cpntools.wiki.

[10] M.I. Gordon et al. A stream compiler for communication-exposed architectures. In *10th Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 291–303, 2002.

[11] U.J. Kapasi et al. Programmable stream processors. *IEEE Computer*, 36(8), 2003.

[12] M. Geilen and T. Basten. Reactive process networks. In *ACM International Conference on Embedded Software (EMSOFT)*, 2004.

[13] T. A. Henzinger. The theory of hybrid automata. In *LICS*, 1996.

[14] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1, 2 and 3 of *Monographs in Theoretical Computer Science*. Springer-Verlag, 1997.

[15] M. Jersak and R. Ernst. Enabling scheduling analysis of heterogeneous systems with multi-rate data dependencies and rate intervals. In *DAC*, 2003.

[16] P. Krcál and W. Yi. Decidable and undecidable problems in schedulability analysis using timed automata. In *TACAS*, 2004.

[17] Y.-T.S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM TODAES*, 4(3):257–279, 1999.

[18] A. Maxiaguine, S. Künzli, S. Chakraborty, and L. Thiele. Rate analysis for streaming applications with on-chip buffer constraints. In *ASP-DAC*, 2004.

[19] A. Maxiaguine, S. Künzli, and L. Thiele. Workload characterization model for tasks with variable execution demand. In *DATE*, March 2004.

[20] C. Norström, A. Wall, and W. Yi. Timed automata as task models for event-driven systems. In *RTCSA*, 1999.

[21] E. Rodríguez-Carbonell and A. Tiwari. Generating polynomial invariants for hybrid systems. In *HSCC*, 2005.

[22] M.J. Rutten, J.T.J. van Eijndhoven, E.G.T. Jaspers, P. van der Wolf, O.P. Gangwal, and A. Timmer. A heterogeneous multiprocessor architecture for flexible media processing. *IEEE Design & Test of Computers*, 19(4):39–50, July-August 2002.

[23] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache aware optimization of stream programs. In *ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.

[24] K. Strehl and L. Thiele. Interval diagrams for efficient symbolic verification of process networks. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 19(8), 2000.

[25] G. Varatkar and R. Marculescu. On-chip traffic modeling and synthesis for MPEG-2 video applications. *IEEE Transactions on VLSI*, 12(1), January 2004.

[26] V.D. Živković, P. van der Wolf, E.F. Deprettere, and E.A. de Kock. Design space exploration of streaming multiprocessor architectures. In *IEEE Workshop on Signal Processing Systems (SIPS)*, San Diego, California, 2002.

[27] E. Wandeler, A. Maxiaguine, and L. Thiele. Quantitative characterization of event streams in analysis of hard real-time applications. In *IEEE RTAS*, 2004.