Automatic Abstraction for Worst-Case Analysis of Discrete Systems

Felice Balarin

Cadence Berkeley Laboratories

Abstract

Recently, a methodology for worst-case analysis of discrete systems has been proposed [1, 2]. The methodology relies on a user-provided abstraction of system components. In this paper we propose a procedure to automatically generate such abstractions for system components with Boolean transition functions. We use a binary decision diagram (BDD) of the transition function to generate a formula in Presburger arithmetic representing the desired abstraction. Our experiments indicate that the approach can be applied to control-dominated embedded systems.

1. Introduction

System verification is hard because system responses need to be checked for all legal behaviors of the environment. Typically, there are infinitely many such behaviors. Even when the problem can be reduced to enumerating finitely many internal system states, their number is usually prohibitive. Using abstractions and implicit state enumeration can simplify the problem, but complete verification is at best at (and often beyond) the limit of existing computers.

An alternative approach is the worst-case analysis, where the system response is analyzed only for the most demanding behaviors of the environment. In [1, 2], a general methodology for worst-case analysis of systems with discrete observable signals was proposed. The methodology can be used to verify different properties of systems, such as power consumption, timing performance, or resource utilization. It consists of three main phases:

1. choosing an abstract representation of signals, called a *signature*,
2. building abstractions (called σ-*abstractions*) of system components,
3. analyzing σ-abstractions and interpreting results.

The methodology requires that an ordering be defined in the domain of signatures, so that it can be precisely determined if a signal is "worse" than the other one. In addition, signatures need to preserve sufficient information such that the usage of resource of interest (e.g. time, power, memory bandwidth,...) can be accurately estimated. No such a domain is suitable for all systems and all types of analysis, and only a knowledgeable user can choose the right one. However, we have observed that a class of domains seems useful in many practical cases. This class consists of linear constraints on the number of occurrences of interesting events. Since we allow linear constraints to be combined to form arbitrary formulas in Presburger arithmetic, we call this class *Presburger event-count constraints*, or *PECCs* for short.

Using PECCs, it is possible to define signal sets like: *" All signals which contain between three and five memory accesses."* which can be represented with formula $3 \leq n_{\text{access}} \leq 5$. Similarly the class of signals satisfying: *"All signals in which the system starts, but never returns to one of the initial states."* can be represented with $\sum_i n_i = 1$, where n_i is the number of times the system visits some initial state i. PECCs are also usually sufficient to express information necessary to answer important questions about the performance of the system (e.g. *"How much processor time is required to process all the inputs?"*, *"How many output events will be generated?"*,...).

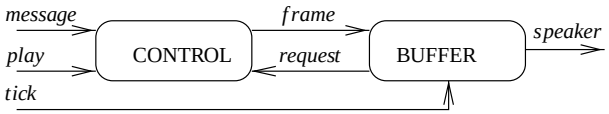
Once a signature is chosen, the appropriate σ-abstractions need to be constructed. For the analysis to be "worst-case", σ-abstractions have to be conservative, i.e. they need to predict a system response that is at least as "bad" as the real response. For a given signature, there exists a well-defined *least-conservative* (or *tightest*) σ-abstraction of any system component. One may ask if such a tightest σ-abstraction can be generated automatically. The answer of this question depends on how system components are specified. If components are specified by an arbitrary piece of code, then the problem undecidable (it is not hard to see that it subsumes the halting problem). However, we will show in this paper that the problem is solvable if the components are given as Boolean functions, mapping input events to output events.

The main contribution of this paper is a procedure to construct the tightest σ-abstraction of a system component

with Boolean transition function, assuming that signal sets are represented with PECCs. We also show that PECCs are natural to define sets of signatures, in the sense that for any component and any given set of input signals with PECC definable signatures, the set of signatures of corresponding output signals is also PECC definable.

Voice mail pager Throughout this paper we use as an example a voice mail pager shown in Figure 1. To facilitate future references, we briefly explain its behavior. The pager receives messages from the environment. Each *message* consists of up to five frames, and each frame contains fifty samples. The CONTROL module stores messages internally (in variable *frames*) and initiates playing of the most recent message (by generating the *frame* event) when the user requires so (by generating the *play* event). The CONTROL module also generates a *frame* if some are available, and the BUFFER module makes a *request*. The BUFFER module starts playing the message after it receives the initial *frame* from CONTROL. A message is played by sending to the *speaker* one sample per every *tick* of the real-time clock. When there are fewer than 20 samples left to play, the BUFFER sends a *request* for the next frame to the CONTROL.

The rest of this paper is organized as follows. In Section 2 we review the worst-case analysis proposed in [1, 2]i, and give brief introduction to BDDs and Presburger arithmetic. We propose the procedure for automatic σ-abstraction generation in Section 3. In Section 4 we show how automatic σ-abstraction fits into overall timing analysis, using the voice mail pager as an example. We present some experimental results in Section 5, and give some final remarks and indications of the future work in Section 6.




```
   module CONTROL { frameType frames[3]; integer last := 0;
1    if( present( message )) {
2      frames := message;
3      last := 3; }
4    if( (present( play ) || present( request ) && last > 0 ) {
5      emit frame( frames[ last -- ] ); } }

   module BUFFER { sampleType samples[50]; integer last := 0;
6    if( present( frame )) {
7      samples := frame;
8      last := 50; }
9    if( present( tick ) && last > 0)) {
10     emit speaker( samples[ last -- ] );
11     if(last = 20)) {
12       emit request(); } } }
```

Figure 1. Voice mail pager structure and behaviors of components.

2. Background

In this section we give relevant definitions, and briefly review the worst-case analysis proposed in [1]. In that work, a *system* is formally defined to be a set of executions. An *execution* is just a function mapping time points in some interval of real numbers to some set of *signal values*.

2.1. Signals, transition functions and BDDs

In [1] signal values may be arbitrary. However, we only consider systems which signal values are assignments to several Boolean variables. We use $x, y, z, \ldots$ to denote those variables. We assume that value 1 indicates the presence of some *event*, while value 0 indicate its absence.

Formally, a system is a set of execution, but since this set is typically infinite, it must be indirectly specified. The theory in [1] does not depend on any particular specification, but we restrict our attention to system specified as follows:

- system components are given by Boolean *transition functions* which determine the presence or absence of output events based on the input events,
- the set of *scheduling rules* is given to determine when some component needs to be evaluated,
- when components are evaluated, they "consume" all their inputs.

A useful way of thinking about these rules is to interpret input signal value 1 as putting a token at the inputs of the component. When a component is executed all tokens are removed. If between two executions more than one token is put on some input, all but the last one will be lost (or dropped, or over-written). When executed, the component can put a token on some of its outputs by setting its value to 1. In that case we say that the component *emits* the output. This framework is general enough to include several specification formalisms proposed for reactive embedded systems such as Esterel [5] and POLIS [4].

We make no assumptions about the scheduling rules, but we do require that the component transition functions are represented with *binary decision diagrams (BDDs)*. This assumption is not fundamental, but it simplifies the presentation of our approach.

A BDD [6] is a data structure that represents a Boolean function. A function $f : \{0,1\}^n \mapsto \{0,1\}$ is represented with a directed acyclic graph with node set V and edge set $E \subset V \times V$. The node set V has two distinguished *terminal* nodes 0 and 1 which can have no out-going edges. All

other nodes must have exactly two out-going edges, one *positive* labeled with 1, and one *negative* labeled with 0. All nodes in $V - \{0,1\}$ must have at least one predecessor, except for the unique *root* node. The *labeling function* $\lambda : V - \{0,1\} \mapsto \{x_1, \ldots, x_n\}$ labels each node with some input variable. A total order must be defined on the set of input variables $\{x_1, \ldots, x_n\}$, and λ must levelize the BDD with respect to that order, i.e. for each $(i, j) \in E$ such that $j \neq 0, 1$ it must be that $\lambda(i)$ precedes $\lambda(j)$.

BDDs are evaluated by traversing a path from the root node to one of the terminal nodes. At node i the value of $\lambda(i)$ is checked. If it is 1, the positive out-going edge of i is followed to reach the next node. Otherwise, the negative edge is followed.

Even though BDDs represent only single-output Boolean function directly, some multi-output function $f : \{0,1\}^n \mapsto \{0,1\}^m$ can be represented with the single-output function $c_f : \{0,1\}^{n+m} \mapsto \{0,1\}$ defined by:

$$c_f(a_1 \cdots a_n, b_1 \cdots b_m) = 1 \text{ iff } f(a_1 \cdots a_n) = (b_1 \cdots b_m) \quad . \tag{1}$$

It follows that while evaluating a multi-output function, if the current node is labeled with an output variable, one must choose the unique out-going edge (and thus also output value) that leads to the terminal node 1.

Consider, for example, the CONTROL module in Figure 1. Its transition function can be represented as a Boolean function with 7 inputs inputs and 5 outputs. Inputs *ms*, *pl* and *rq*, indicate the presence of *message*, *play* and *request* events. A token on the input $l0$ ($l1, l2, l3$) indicates that the current value of variable *last* is 0 (1,2,3 respectively). Output fr indicated the emission of the $frame$ event. Putting a token on output $n0$ ($n1, n2, n3$) indicates that the new value of *last* is 0 (1,2,3 respectively).

The BDD representing the transition function of CONTROL is shown in Figure 2. For readability, terminal node 0 and all incident edges are not shown. All the nodes with the same label are placed on the same horizontal level as the label itself. For example, the path $2 \to 4 \to 7 \to 11 \to 15 \to 16 \to 20 \to 21 \to 24 \to 27 \to 1$ indicates that if *play* is present, *message* is not, and the value of *last* is 2, then $frame$ is emitted, and *last* is set to 1.

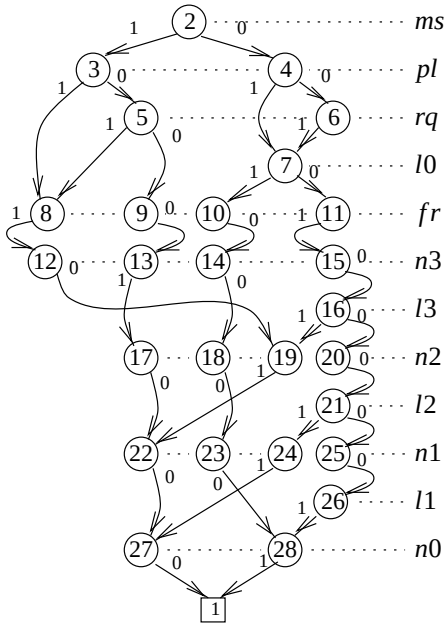


Figure 2. BDD for the transition function of CONTROL.

2.2. Signatures, σ-abstractions and PECCs

A *signature* is an abstract representation of executions (or their segments). Formally, a signature σ is a mapping from the set of executions to some set of *signature values*. A signature must satisfy some mild technical conditions, and a partial order $\leq$ needs to be defined on the set of signature values. While results in [1] apply to any signature satisfying these constraints, we consider only PECCs. PECCs are subsets of the *event-count space*. For the system whose signal values are assignments to Boolean variables $x, y, z, \ldots$, the event-count space is the set of assignments to integer variables $x^{\#}, y^{\#}, z^{\#}, \ldots$. We assume that for a given execution, the value $x^{\#}$ corresponds to the number of occurrences of the event x in that execution.

The signature maps any execution to a point in the event-count space. However, since many executions can be mapped to a single point, it is quite possible that input signals with the same signature produce outputs with different signatures. For this reason we need to be able to define and manipulates sets of points in the event-count space. We use formulas of the *Presburger arithmetic* [8] to define such sets. Formulas of Presburger arithmetic include linear equalities and inequalities (with integer coefficients) over some set of non-negative integer variables, as well as their combinations using standard Boolean operators $\vee, \wedge, \neg$ and existential quantification. With each formula we can associate a set of integer vectors which represent all satisfying assignment to free variables in the formula. Sets represented by Presburger formula can be effectively represented, manipulated and compared for inclusion and equivalence. The best known bound for these operations is triply exponential, but there are algorithms that are much more efficient for many practical cases [9].

We require that only variables from the event-count space may appear free in the formula. Therefore, each Presburger formula defines a set of points in the event-count space (points that satisfy the formula). We are now ready to precisely define PECCs: they are subsets of the event-

count space definable by formulas of Presburger arithmetic. However, we will make no distinction between a PECC and a formula that defines it.

The theory in [1] requires that a partial order $\leq$ be defined on the set of signature values. We use set inclusion for these purpose, i.e. given two PECCs X and Y we say that $X \leq Y$ if $X \subset Y$. Equivalently, we could say that $X \leq Y$ if Y implies X.

For example, if it is known that between any two *message* events at least 200 time units must elapse, then all the signals of length T must satisfy $200(ms^{\#} - 1) \leq T$. If a particular signal is of length 1000 and only two messages appear in it, its signature $ms^{\#} = 2$ is "better" than $200(ms^{\#} - 1) \leq 1000$ and we write $[ms^{\#} = 2] \leq [200(ms^{\#} - 1) \leq 1000]$.

While signals are represented with signatures, system components are represented with σ-*abstractions*. A σ-abstraction is valid if it satisfies some minor requirements detailed in [1, 2], and one major one: it must be a conservative predictor of system behavior. More precisely, a function F is a valid σ-abstraction of a component f if for every input execution trace s:

$$\sigma(f(s)) \leq F(\sigma(s)) \quad . \tag{2}$$

Equivalently, we may say that F must be an *abstract interpretation* [7] of f.

For example, since every *message* in Figure 1 contains 3 *frames*, the number of *frame* events emitted by the CONTROL module is at most three times larger than the number of received messages. Therefore, a valid σ-abstraction of CONTROL could map PECC $ms^{\#} = 2$ into $fr^{\#} \leq 6$. Similarly, since a *frame* is emitted only if response to *play* or *request* events, mapping $pl^{\#} + rq^{\#} = 1$ into $fr^{\#} \leq 1$ is a valid abstraction. We will show that such abstractions can be automatically generated from the BDD of the transition function. The key intermediate step in the generation process is finding input-output constraints imposed by the transition function. In the case of CONTROL they are $fr^{\#} \leq 3ms^{\#}$ and $fr^{\#} \leq pl^{\#} + rq^{\#}$ (for the same reasons stated above).

The methodology for worst case analysis proposed in [1, 2] requires the user to provide a σ-abstraction candidate which is then verified by formal or informal means. In this paper, we will show that if the set of signature values is that the set of PECCs, then it is possible to automatically construct σ-abstractions of systems components with Boolean transition function. Using this result, the methodology from [1] can be improved, by making it more automatic and less dependent on the user.

3. Automatic abstraction

Given a BDD-represented transition function f with inputs $x_1, \ldots, x_n$ and outputs $y_1, \ldots, y_m$ we want to find s σ-abstraction function F which maps PECCs over variables $x_1^{\#}, \ldots, x_n^{\#}$ to PECCs over variables $y_1^{\#}, \ldots, y_m^{\#}$ such that (2) is satisfied. We do so in two steps: first we define PECC TF (representing transition function) over variables $x_1^{\#}, \ldots, x_n^{\#}$ and $y_1^{\#}, \ldots, y_m^{\#}$, and then we define F such that for every PECC X over $x_1^{\#}, \ldots, x_n^{\#}$:

$$F(X) \stackrel{\text{def}}{=} \exists x_1^{\#}, \ldots, x_n^{\#} : X \wedge TF \quad . \tag{3}$$

For example, the TF for CONTROL would ideally be $fr^{\#} \leq 3ms^{\#} \wedge fr^{\#} \leq pl^{\#} + rq^{\#}$. (Definition (3) is easily extended to the case where X depends on some additional variables $x_{n+1}^{\#}, \ldots, x_{n+m}^{\#}$ which do not correspond to inputs of f. In that case the appearance of X in the right-hand side of (3) needs to be replaced with $(\exists x_{n+1}^{\#}, \ldots, x_{n+m}^{\#} : X)$.)

To define TF we introduce a pair v^i, p^i of auxiliary integer variables for each node i of BDD (V, E) representing the transition function f. Intuitively, v^i represents the number of times node i is visited while processing some input signal. The variable p^i represents the number of times $\lambda(i)$ is 1 when i is visited. In other words, p^i represents the number of times the positive out-going edge of i is traversed.

The PECC TF is a conjunction of many constraints that can be classified as either fanin, fanout, input, output or consistency constraint. The *fanout constraint* FO_i is defined for each node $i \in V - \{0, 1\}$. It simply states that the positive edge of i cannot be visited more times than i itself:

$$FO_i \stackrel{\text{def}}{=} p^i \leq v^i \quad . \tag{4}$$

Similarly, the *fanin constraint* FI_i states that a node can only be visited after visiting some of its in-coming edges. Formally, for every node $i \in V - \{0, 1\}$ that is not the root:

$$FI_i \stackrel{\text{def}}{=} v^i = \sum_{(j,i) \in E} \varepsilon_{(j,i)} \quad , \tag{5}$$

where *edge expression* $\varepsilon_{(j,i)}$ is defined by:

$$\varepsilon_{(j,i)} \stackrel{\text{def}}{=} \begin{cases} p^j & \text{if } (j,i) \text{ is labeled with 1,} \\ v^j - p^j & \text{if } (j,i) \text{ is labeled with 0.} \end{cases} \tag{6}$$

It is not hard to see that $\varepsilon_{(j,i)}$ represents the number of times edge (j,i) is visited, be it positive or negative. For example, the fanin constraint for node 27 in Figure 2 is $v^{27} = v^{22} - p^{22} + p^{24}$.

It is convenient to assume that a fanin constraint is defined for the root node as well. Such a constraint represent a limit on the number of times the transition function f is evaluated for a given input signal. This can often be deduced from the scheduling rules. For example, it might be that f is not evaluated unless there are some new input events present. If that if the case, than the CONTROL module cannot execute more times than the total number of its input events, i.e. the number of *message*, *play*, and *request* events combined. Therefore, the fanin constraint for

the root node in Figure 2 would be: $v^2 \leq ms^{\#} + pl^{\#} + rq^{\#}$. Even if nothing is known about the scheduling, to simplify our notation, we still assume that a (trivial) fanin constraint is defined for the root node.

The input constraint IO_x is defined for each input variable x of f. To understand it, observe that the positive edge of node i is visited only if the variable $\lambda(i)$ is 1. Also note that some variable x cannot be 1 more than $x^{\#}$ times. Therefore:

$$IO_x \stackrel{\text{def}}{=} \sum_{i:\lambda(i)=x} p^i \leq x^{\#} \ . \tag{7}$$

For example, $p_5 + p_6 \leq rq^{\#}$ in Figure 2.

Inequality (7) cannot be strengthen to an equality for two reasons. First, there may be paths in the BDD which do not visit any node labeled x, and second we may not generally assume that the BDD is evaluated every time there is a new event on x. Depending on the scheduling policy, it may be that some x events are lost, i.e. it may be that x occurs more than once between two BDD evaluations.

Output constraints are similar but tighter, because an output x can become 1 only if the positive edge of some BDD node labeled x is visited. (This is true if transition function is indeed a function, i.e. if it has not "don't-care" outputs. Our approach can also be extended to handle "don't-care" outputs, but for clarity of presentation, we omit these extensions in this paper.) Formally, for every output variable x, we define the *output constraint* IO_x as follows:

$$IO_x \stackrel{\text{def}}{=} x^{\#} = \sum_{i:\lambda(i)=x} p^i \ . \tag{8}$$

For example, $fr^{\#} = p^8 + p^9 + p^{10} + p^{11}$ in Figure 2. A careful reader might protest the terms p^9 and p^{10} because they correspond to input-output pairs that are *not* in the transition function. Indeed, they are forced to 0 by consistency constraints.

According to (1) every valid traversal of a BDD representing a multi-output function must finish at the terminal node 1. Therefore, we define the following *consistency constraint* for each edge $(i,0) \in E$:

$$C_i \stackrel{\text{def}}{=} \varepsilon_{(i,0)} = 0 \ . \tag{9}$$

For example, $v^6 - p^6 = v^8 - p^8 = p^9 = p^{10} = \cdots = 0$ in Figure 2.

Finally, we form TF by combining all constraints defined by (4)–(9), and eliminating all the auxiliary variables, i.e.:

$$TF \stackrel{\text{def}}{=} \exists \underbrace{v^i, p^i, v^j, p^j, \ldots}_{\{i,j\ldots\}=V-\{0,1\}} : \bigwedge_i (FI_i \wedge FO_i) \wedge \bigwedge_x IO_x \wedge \bigwedge_{(i,0)\in E} C_i, \tag{10}$$

where i is some BDD node other than 0 or 1, and x is some input or output variable of f.

The σ-abstraction F defined by (3) is intended to approximate the *true transition signature* $F^{\top}$ defined by:

$$F^{\top}(X) \ = \ \{\sigma(f(s)) : \sigma(s) \in X\} \ ,$$

where X is some subset of the input event count space, possibly a PECC. It is not hard to see that conservativeness requirement (2) is satisfied if $F^{\top}(X) \leq F(X)$ for every PECC. We will show that an even stronger relation holds, under a mild assumption on the scheduling policy. The assumption we need is that given n assignments to input variables of a component, it is possible to find an input signal s such that while processing s the component is executed exactly n times, and the input values for the i'th execution are exactly those of the i'th assignment. For most scheduling rules, this can be accomplish by timing the assignments far enough from each. Since our framework is not appropriate for reasoning about the scheduling rules, we will just say that a scheduling policy with this property is *well-behaved* and state it explicitly as an assumption.

Theorem 1 *For any well-behaved scheduling policy:*

$$F(X) = F^{\top}(X)$$

for every PECC over $x_1^{\#}, \ldots, x_n^{\#}$.

Proof. **(sketch)** To show $F^{\top}(X) \leq F(X)$ consider an arbitrary signal s with signature $(\bar{x}_1^{\#}, \ldots, \bar{x}_n^{\#}) \in X$, and let $\bar{v}^i$ and $\bar{p}^i$ be the actual number of times node i and the positive edge of i are visited while processing s. Also let $(\bar{y}_1^{\#}, \ldots, \bar{y}_m^{\#})$ be the signature of $f(s)$. Argue that assigning $\bar{x}_1^{\#}, \ldots, \bar{x}_n^{\#}$, $\bar{y}_1^{\#}, \ldots, \bar{y}_m^{\#}, \bar{v}^i, \bar{p}^i, \bar{v}^j, \bar{p}^j \ldots$ to $x_1^{\#}, \ldots, x_n^{\#}, y_1^{\#}, \ldots, y_m^{\#}, v^i, p^i, , v^j, p^j, \ldots$ must satisfy (4)–(9), and hence also the right hand side of (3). It follows that $\sigma(f(s)) \in F(X)$.

To show $F(X) \leq F^{\top}(X)$, let $(\bar{y}_1^{\#}, \ldots, \bar{y}_m^{\#})$ be some element of $F(X)$, and let $(\bar{x}_1^{\#}, \ldots, \bar{x}_n^{\#}) \in X$ and $\bar{v}^i, \bar{p}^i, \bar{v}^j, \bar{p}^j, \ldots$ be such that assigning $\bar{x}_1^{\#}, \ldots, \bar{x}_n^{\#}$, $\bar{y}_1^{\#}, \ldots, \bar{y}_m^{\#}, \bar{v}^i, \bar{p}^i, \bar{v}^j, \bar{p}^j, \ldots$ to $x_1^{\#}, \ldots, x_n^{\#}, y_1^{\#}, \ldots, y_m^{\#}, v^i, p^i, v^j, p^j, \ldots$ satisfies (4)–(9). Also, let r denote the root BDD node. Construct $\bar{v}^r$ different assignments to $x_1, \ldots, x_n$ such that:

1. for each $k = 1, \ldots, n$: x_k is 1 in exactly $\bar{x}_k^{\#}$ assignments,
2. for each $i \in V$: the evaluation of exactly $\bar{v}_i$ assignments visits node i, and the evaluation of exactly $\bar{p}_i$ assignments visits the positive edge of i,
3. for each $k = 1, \ldots, m$: to reach terminal node 1 and avoid node 0, the positive edges of some node labeled with y_k must be visited in exactly $\bar{y}_k^{\#}$ assignments (i.e. y_k is emitted $\bar{y}_k^{\#}$ times).

Argue by induction on BDD levels that such a construction is possible: show that valid assignments can be made

to some variable x assuming valid assignments exists to all variables preceding x. Finally, since the scheduler is assumed to be well-behaved, it is possible to find input signal s such that the signature of s is $(\overline{x}_1^\#, \ldots, \overline{x}_n^\#)$ and the signature of $f(s)$ is $(\overline{y}_1^\#, \ldots, \overline{y}_m^\#)$. Thus, $(\overline{y}_1^\#, \ldots, \overline{y}_m^\#) \in F^\top(X)$. □

The part $F^\top(X) \leq F(X)$ shows that constraints (3)–(10) indeed define a valid σ-abstraction that can be used in the worst-case analysis. Note that to prove this direction we did not need the assumption that the scheduling policy is well-behaved.

The part $F(X) \leq F^\top(X)$ shows that (3)–(10) defines the best possible σ-abstraction. It also shows that restricting ourselves to PECC definable subsets of event count spaces causes no additional loss of information, because the true transition signature of a PECC definable set is also PECC definable.

4. Timing analysis of the voice mail pager

The automatic abstraction described in the previous section is only one step of the worst case methodology. In this section we use the voice mail pager to show how this step fits in the overall timing analysis flow.

The first step of the timing analysis is to generate σ-abstractions of all system components, including the environment. For the CONTROL and BUFFER module that means applying the automatic abstraction procedure from Section 3. However, in case of CONTROL, evaluating (10) would not give the desired result $fr^\# \leq 3ms^\# \wedge fr^\# \leq pl^\# + rq^\#$. That is because the BDD in Figure 2 does not reflect the fact that $l0 \cdots l3, n0 \cdots n3$ all represent the same state variables. The complete explanation of how finite-state machines are handled in this framework are beyond the scope of this paper, and come be found in [3]. Still, for the completeness of the example we state that desired results can be obtained by first conjoining TF with $l0^\# \leq n0^\# + 1 \wedge n1^\# \leq l1^\# \wedge n2^\# \leq l2^\# \wedge n3^\# \leq l3^\#$ and then existentially quantifying $l0^\# \cdots l3^\#, n0^\# \cdots n3^\#$.

Applying the similar procedure to BUFFER gives us the TF:

$$sm^\# \leq tk^\# \wedge sm^\# \leq 50\,fr^\# \wedge rq^\# \leq fr^\# \wedge 30(rq^\# - 1)^\# \leq tk^\# \ ,$$

where $sm^\#$ and $tk^\#$ are the number of occurrences of *sample* and *tick* events. This relation can be justified by observing in Figure 1 that at most one *sample* is emitted per each *tick*, and at most 50 *sample* events are emitted per each *frame*. Furthermore, *request* are emitted only after *frame* is received, and between the *frame* and *request* events (and thus also between two *request* events) at least 30 *tick* events must occur, while *last* is being decremented from 50 to 20.

The σ-abstraction of the environment usually has to be manual, because a complete and precise specification of legal behaviors of the environment very rarely exists in some machine-readable form. For external events of the voice-mail pagers, the only constraint is a minimum time between two occurrences. These times are 125, 625, and 5000 time units for *tick*, *message* and *play* events, respectively. Therefore, an input signal of length T must satisfy the following:

$$125(tk^\# - 1) \leq T \wedge 625(ms^\# - 1) \leq T \wedge 5000(pl^\# - 1) \leq T \ .$$

We use this expression as a TF for the environment, and treat T as an input, and $tk^\#$, $ms^\#$ and $pl^\#$ as outputs. To be able to do this, we need to add time as an extra dimension in the event count space.

The second step of timing analysis is to generate the *work-load* function W. Intuitively, given a PECC X, the value of $W(X)$ must provide an upper bound on the processor time needed to process signal with signature in X. To generate such function we need to know execution times of all the basic blocks. Ideally, these times should be exact, but a conservative estimate can also be used. This information can be gathered in various ways, including prototyping, benchmarking, static analysis and simulation (e.g. [10]). Gathering this information is an active research area, beyond the scope of this paper. For simplicity, we assume that the execution time of each executable line in Figure 1 is 10 time units. Also, we assume that modules are executed only if there are some new events present at their inputs. Then, a work-load function $W(X)$ for the pager might be:

$$\begin{aligned} \exists pl^\#, rq^\#, ms^\#, fr^\#, tk^\# : (\exists T : X) \wedge (T \leq \\ 20(pl^\# + rq^\# + ms^\#) + 20ms^\# + 10\,fr^\# + \\ 20(fr^\# + tk^\#) + 20\,fr^\# + 20sm^\# + 10rq^\#) \ . \end{aligned} \quad (11)$$

The second line in (11) corresponds to the CONTROL module, while the third line corresponds to the BUFFER module. The term $20ms^\#$ in (11) is due to lines 2 and 3 in Figure 1, which will be executed only if a new *message* is received. Similarly, the number of executions of line 5 is the same the number of generated *frame* events (hence the term $10\,fr^\#$), and so on.

Finally, we need to find a PECC X satisfying the fix-point equation:

$$X \ = \ F_{\text{CONTROL}}(X) \wedge F_{\text{BUFFER}}(X) F_{\text{environment}} \wedge W(X) \ .$$

Such an X can be found by simple iteration, starting with the initial solution $pl^\# = rq^\# = ms^\# = fr^\# = tk^\# = sm^\# = T = 0$. Projecting such an X onto T gives a bound on the maximum *busy period*, i.e. the longest interval of time the processor is continuously busy. For the voice mail pager, it takes six iterations to find a fix-point:

$$(pl^\#, ms^\#, rq^\# \leq 1) \wedge (tk^\#, fr^\#, sm^\# \leq 2) \wedge T \leq 230 \ .$$

This fix-point indicates that the maximum busy period is 230, and that in that period *tick*, *frame* and *sample* events can occur at most twice, *play*, *mesaage* and *request* can occur at most once.

name	in/out	nodes	time	name	in/out	nodes	time
spe	3/2	10	0.2s	gen	2/1	5	0.2s
acq	2/2	8	0.2s	odo	3/2	9	0.2s
pwme	12/2	19	0.2s	pwmi	2/2	8	0.2s
pwmr	1/1	4	0.2s	staf	2/2	8	0.2s
fcl	5/2	16	0.2s	lac	5/2	17	0.2s
lad	4/2	12	0.2s	lsdp	5/2	17	0.2s
lsk	4/2	12	0.2s	laf	4/2	12	0.2s
saa	5/1	8	0.2s	vafl	4/2	12	0.2s
arb	5/1	8	0.2s	vafm	4/2	12	0.2s
fil	4/2	12	0.2s	vac	5/3	24	0.3s
lsdo	6/3	24	0.3s	vad	10/4	59	8.1s
trs	9/3	46	>1h				

Table 1. Experimental results

5. Experimental results

We have implemented the proposed approach in a combination of POLIS embedded system environment [4] and Omega calculator for manipulating formulas of Presburger arithmetic [9]. POLIS is capable of capturing embedded system designs, optimizing them, and generating hardware or software implementations, but for our purposes it is important that it builds a transition function BDD for each system component. We have extended POLIS to traverse these BDDs and create input Omega files containing formulas (10). The time necessary to create the formulas was always negligible compared to time to read the design in and create the BDDs. Omega reads in the formula and simplifies it. The main simplification is elimination of auxiliary variables. The result is typically a simple conjunction of inequalities over input and output variables.

We had a total of 46 systems components available for our experiments. A half of them resulted in a formula that was too big to handle by Omega's internal data structures. The results for the other half are summarized in Table 1. For most of the examples the total run time is dominated by the constant overhead times. Even for the largest example the run time is quite reasonable. Only one of the examples (trs) could not finish in the alloted time of one hour. This should be expected as the algorithm is very complex in the worst case. Modest run times for most examples lead us to believe that the Omega algorithm is capable of handling many of the larger formulas as well. However, Omega was originally developed for a different application where formulas are relatively more complex than ours but have fewer variables, and the internal data structure were sized accordingly.

Even if a formula is too complex for the Omega algorithm, several options are still available. First is decomposition. It may be possible to break a large formula into several pieces each of which is smaller and has fewer variables. The key in this process is to group constraints depending on the same variables, allowing for early quantification. The other approach to complex formulas is approximation. In particular, if Presburger formulas are interpreted over positive reals rather then positive integers, then operations on formulas such as existential quantification becomes much simpler. However, there is a risk that the σ-abstraction computed using reals may not be as tight as the one computed with integers. Since the primary focus of this paper is the correctness and not efficiency of automatic σ-abstraction, we leave decomposition and approximation for future work.

6. Conclusions

We have proposed a procedure to automatically generate abstractions required for worst-case analysis of discrete systems [1, 2]. Our approach requires that system components be represented with a BDD, which is then used to generate a formula in Presburger arithmetic representing the desired abstraction.

We believe, that the approach can be immediately applied to a significant class of systems, but also that the application area can be widen by weakening some assumptions and improving the efficiency. Probably the most serious restriction of our approach is BDD representation of system components. BDDs (and Boolean functions in general) have shown to be very well suited for control-dominated systems, but they tend to grow unacceptably large if used to represent system with even a modest amount of data manipulation. Therefore, several system specification formalisms have emerged where data and control parts of the system are kept separately, and BDDs are used for control parts only. Our approach should be extended to such specification formalisms.

Finally, the success of our approach ultimately depends and the ability to manipulate efficiently formulas of Presburger arithmetic. In general those manipulations are complex, but we use a very small subset of the Presburger arithmetic: conjunction, existential quantification and equalities and inequalities with small variable coefficients (-1, 0, or 1). It is not known whether there are more efficient algorithms for this subset, at least in average, if not in the worst case.

References

[1] F. Balarin. Worst-case analysis of discrete systems. In *Digest of Technical Papers of the 1999 IEEE International Conference on CAD*, Nov. 1999.

[2] F. Balarin. Worst-case analysis of discrete systems based on conditional abstractions. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES'99)*, July 1999.

[3] F. Balarin. Automatic abstraction for worst-case analysis of finite-state and valued-event systems. submitted to Design Automation Conference , 2000.

[4] F. Balarin et al. *Hardware-software co-design of embedded systems: the POLIS approach.* Kluwer Academic Publishers, 1997.

[5] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In S. Brookes, A. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 369–448. Carnegie-Mellon Univeristy, 1984.

[6] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, June 1990.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. 4th Ann. ACM Symp. on Principles of Prog. Lang.* 1977.

[8] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, Inc., 1972.

[9] W. Kelly, V. Maslov, W. Pugh, E. Roser, T. Shpeisman, and D. Wonnacott. The Omega calculator and library. In *http://www.cs.umd.edu/projects/omega*, Nov. 1996.

[10] K. Suzuki and A. Sangiovanni-Vincentelli. Efficient software performance estimation models for hardware/software codesign. In *Proceedings of the Design Automation Conference*, June 1996.