Interval Diagrams for Efficient Symbolic Verification of Process Networks

Karsten Strehl, *Member, IEEE,* and Lothar Thiele, *Member, IEEE*

Abstract— **In this paper, a representation of multi-valued functions called *interval decision diagrams* (IDDs) is introduced. It is related to similar representations such as binary decision diagrams. Compared to other functional representations with regard to symbolic formal verification approaches, IDDs show some important properties that enable us to verify process networks and related models of computation more adequately than with conventional approaches. Therefore, a new form of transition relation representation called *interval mapping diagram* (IMD) is introduced.**

A novel approach to symbolic model checking of process networks is presented. Several drawbacks of traditional strategies are avoided using IDDs and IMDs. The resulting transition relation IMD is very compact, enabling fast image computations. Furthermore, no artificial limitations concerning buffer capacities or equivalent have to be introduced. Additionally, applications concerning scheduling of process networks are feasible. IDDs and IMDs are defined, their properties are described, and computation methods and techniques are given.

I. Introduction

DURING the last years, a promising approach called *symbolic model checking*, see, e.g., [1], [2], has been applied to many areas of system verification in computer engineering and related research fields and even has been able to enter the area of industrial applications. This approach makes use of *binary decision diagrams* (BDDs) [3], [4] which are an efficient representation of Boolean functions and allow their very fast manipulation. However, when applying this technique to process networks or related models of computation—i.e., models with unbounded system states such as many high-level and dataflow oriented models—, several difficulties occur that question its usefulness in this area.

A. Process Networks

Process network models are commonly used, e.g., for specification and synthesis of distributed systems. They often form the basis of applications such as real-time scheduling and allocation.

We use the term process networks to describe such models of computation characterized as containing concurrent processes which may have complex internal states and buffers for communication. By means of tokens, data items can be written into and read from buffers to exchange data between processes. Buffers often are unidirectional and have a FIFO (*first-in, first-out*) queue behavior, but this is not a required property. A buffer may have assigned a *capacity* which is the maximum number of tokens contained.

K. Strehl and L. Thiele are with the Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), Gloriastrasse 35, 8092 Zurich, Switzerland. eMail: {strehl,thiele}@tik.ee.ethz.ch, WWW: http://www.tik.ee.ethz.ch/~symbolic

Buffers occur in almost every communication scenario between processes. For instance, in real-time operating systems they are often called mailboxes. Process networks are related to programs with finite-state control and integer variables that are accessed by commands like "if $x < 3$ then $y := y + 4$"[1].

Many different classes of process networks exist and are employed in diverse application domains. An example is a recently introduced model of computation called *FunState* [5], [6], [7] which combines dataflow properties with finite state machine behavior and is especially suited to embedded systems design. Many other models of computation, e.g., *dataflow process networks* [8], *computation graphs* [9], and *synchronous dataflow* (SDF) [10], [11], turn out to be special cases of Kahn's process networks [12], [13] and of the FunState model of computation. They have in common that their transition behavior can be represented by a *regular state machine* (RSM) [14]. Since the SDF model is restricted enough, tasks such as static scheduling or the verification of properties are well investigated and may be handled efficiently. The situation is similar for computation graphs. While many dataflow models are sufficiently analyzable by means of balance equation methods, such methods fail for more powerful models due to the complex internal model states.

Typical questions to be answered by means of formal verification of process networks are about the absence of deadlocks or the boundedness of the required memory. The memory boundedness is especially important as process networks in general may not be scheduled statically. Thus, dynamic schedulers have to be deployed which cannot always guarantee compliance with memory limitations without restricting the system model [15].

The process models in addition may be extended to describe one or several dynamic or hybrid scheduling policies. Their behavior then is verified together with the system model. Thus, common properties such as the correctness of the schedule may be affirmed, or artificial deadlocks [15] may be detected. Besides these applications of scheduling analysis described in [16], [7], symbolic model checking principles have been applied to the synthesis area of symbolic scheduling (see [5], [17], [7], for instance), where correct schedules are generated automatically.

A simple example process network from [15] is shown in Figure 1. A, B, C, and D represent processes, while a, b, c, d, and e are unbounded FIFO queues. The internal process states are represented by program code in this example.

[1]It is preferable if these guards and assignments are separable, in contrast to a transition such as "if $x < y$ then $z := x + y$".

The network obeys a blocking read semantics, i.e., a process is suspended when attempting to get data from an empty input channel.

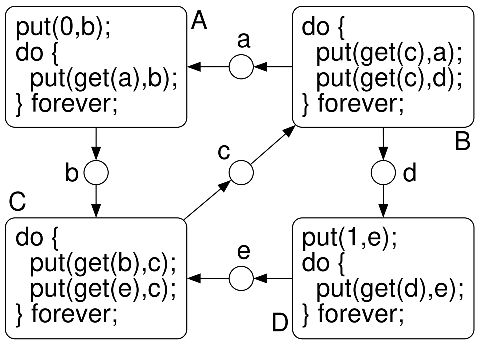


Fig. 1. Simple process network.

Formal verification is able to prove, for instance, that there are never more than two tokens buffered on any communication channel (then the process network is *strictly bounded*), and that the network is non-terminating. Both properties are essential for the development of scheduling policies.

A simple dynamic scheduling example to be modeled in combination with the network is a plain priority-based scheduling policy which executes process B only if A is not enabled, otherwise A is executed. An important question to be answered here is whether B could be blocked forever because A may always be enabled.

B. The Problem

The traditional BDD-based method of automated verification suffers from the fact that a binary representation of the process network and its state is required. One severe problem is that the necessary capacities of the buffers in general are unknown before the analysis process or even may be its desired result. Unfortunately, in order to perform symbolic model checking, the buffer contents represented by an integer number must be binary coded. To save memory and computing power, the coding should be selected such that it covers no more than the integer range necessary—which is unknown. This deficiency could be partially avoided using *multi-valued decision diagrams* (MDDs) [18] with unbounded variable domains instead of BDDs. The decision variables of those diagrams have not only binary but integer values and thus are able to represent the state of a buffer without binary coding. However, problems occur when complementing a finite set described by an unbounded MDD. This results in an infinite set requiring an infinite number of MDD edges which is not possible. One strategy to avoid this is again to bound the variable domain to a finite range such that even complementary sets are finite.

Another difficulty emerges from the very regular behavior [14] of process network transitions that in general consists only of consuming or producing one or a few constant numbers of tokens at a time. Consider a simple dataflow node connecting two buffers. Its firing behavior with respect to its outgoing buffer consists of adding, e.g., one token at a time. When representing this behavior using BDDs or MDDs, a huge part of the transition relation decision diagram (DD) is necessary to model explicitly all possible pairs of a buffer's state and its successor state after the firing, e.g., $\{(x, x')\} = \{(0,1), (1,2), (2,3), \ldots, (n-1, n)\}$. Again, the upper bound n just has to be introduced not to make the transition relation DD too complex and thus intractable. Neglecting all other transitions, each of the state pairs results in at least one DD node. Hence, merely the described part of the complete transition relation needs at least n nodes.

C. Proposed Approach

To overcome the above-mentioned limitations of conventional symbolic model checking of process networks and related models, we present an approach that uses *interval decision diagrams* (IDDs) combined with *interval mapping diagrams* (IMDs) and thus is able to remedy the shortcomings of traditional approaches. Formal verification of process networks using these *interval diagram techniques* has been introduced in [16]. Fundamentally, the approach is based on a more reasonable way of describing the above-mentioned form of transition relations. It affords the possibility of representing them as the "distance" between a state and its successor after the transition, which means the difference between the numbers of included tokens before and after the firing. In this way, the partial behavior mentioned above may be described by one single graph node denoting a *state distance* of 1, and no artificial limitation with an upper state bound is necessary.

Interval diagram techniques are well-suited to handling the combination of internal process states and buffer contents, both representing the total system state. In addition to the symbolic formal verification of process networks, as described in Section VII-A.1 interval diagrams have applications in the fields of timing analysis and real-time symbolic model checking [19], [7], where similar problems have to be dealt with. The major enhancements of symbolic model checking with IDDs and IMDs are:

- No state variable bounds due to binary coding or complementation are necessary as with conventional symbolic model checking.
- The transition relation representation is quite compact—especially for the considered kind of models—since only state distances are stored instead of combinations of state and successor. Accordingly, an innovative technique for image computation is introduced.
- Due to the enhanced merging capabilities of IDDs and the abandonment of binary coding, state set descriptions are more compact than using BDDs.

We introduce the formalism of interval decision diagrams, an efficient representation of discrete-valued functions, and the methods and techniques necessary for applying this new form of function description to symbolic model checking by means of interval diagram techniques. IDDs and IMDs are dedicated to state set and transition relation descriptions of many, especially dataflow oriented models of computation. In this paper, interval diagram techniques are applied to symbolic model checking of process networks. We present interval diagrams and the verification techniques used and compare their run-time behavior with that of the conventional BDD approach. IDDs and IMDs are introduced in Section II, III, and IV. Section V explains how image computation is performed using interval diagram techniques, while Section VI describes the application to symbolic model checking of process networks and presents experimental results. Finally, Section VII discusses related work and gives a brief summary.

II. Interval Decision Diagrams

A. Notation

The set of integer intervals $\mathbb{I}$ denotes the set of all intervals on $\mathbb{Z}$. Intervals can be of the forms $[a,b]$, $[a,\infty)$, $(-\infty,b]$, or $(-\infty,\infty)$, where a or $-\infty$ represent the lower and b or ∞ the upper bound. Squared brackets [and] indicate that the corresponding bound is included in the interval set, whereas round brackets (and) indicate that it is not[2]. [] represents the empty interval containing no elements. Two intervals are called *neighboring* iff they may be joined by union into a larger interval. Overlapping intervals as well are called neighboring.

Let $f(x_1, x_2, \ldots, x_n)$ be a multi-valued function with signature

$$f : P_1 \times P_2 \times \ldots \times P_n \to Q_f,$$

where $P_i \in \mathbb{I}$ are the domain sets of the variables x_i, and Q_f is the discrete and finite range set of f.

The term x^I represents a *literal* of a variable x with respect to an interval $I \in \mathbb{I}$, that is the Boolean function

$$x^I = \begin{cases} 0 & \text{if } x \notin I \\ 1 & \text{if } x \in I \end{cases}.$$

For the sake of brevity, for $I = [b,b]$ containing only one single value $b \in \mathbb{Z}$, the literal of variable x with respect to I is denoted $x^b = x^I$.

The function resulting when some argument x_i of function f is replaced by a constant value b is called a *restriction* or *cofactor* of f and is denoted $f|_{x_i=b}$ or, for the sake of brevity, $f_{x_i^b}$. That is, for any arguments $x_1, \ldots, x_n$,

$$f_{x_i^b}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n).$$

If for all possible values of x_i in some interval $I \subseteq P_i$, f does not depend on x_i, i.e.,

$$\forall b, c \in I : f_{x_i^b} = f_{x_i^c},$$

then f is *independent* of x_i in I, and the cofactor of f with respect to the literal x_i^I is defined by

$$f_{x_i^I}(x_1, \ldots, x_n) = f_{x_i^b}(x_1, \ldots, x_n) \text{ for all } b \in I.$$

[2]Note that in the integer domain open or half-open intervals such as $[a,b)$ can be expressed by a closed interval, namely $[a, b-1]$.

In this case, I is called an *independence interval* of f with respect to x_i.

Definition II.1 (Interval cover) The set $I(P_i) = \{I_1, I_2, \ldots, I_{p_i}\}$ of p_i *split intervals* I_j represents an *interval cover* of P_i if each I_j is a subset of P_i, i.e., $I_j \subseteq P_i$, and $I(P_i)$ is *complete*, i.e.,

$$P_i = \bigcup_{I \in I(P_i)} I.$$

Definition II.2 (Interval partition) A cover is *disjoint* if

$$\forall j, k \text{ with } 1 \le j, k \le p_i, j \ne k : I_j \cap I_k = \emptyset$$

holds, i.e., no element of P_i is included in more than one split interval. A disjoint cover is called an *interval partition*.

Definition II.3 (Independence interval partition) An *independence interval partition* is a partition consisting of independence intervals only.

Based on independence interval partitions, most multi-valued functions of interest may be decomposed with regard to a variable x_i in several partial functions describable by cofactors. Each cofactor contributes to f only in an independence interval with respect to x_i. From now on, only functions are considered that are decomposable over an interval partition with a finite number of independence intervals. Their partial functions may be composed by the Boole-Shannon expansion for a multi-valued function with respect to a variable x_i and an independence interval partition $I(P_i)$, given by

$$f = \bigvee_{I \in I(P_i)} x_i^I \wedge f_{x_i^I}. \tag{1}$$

The operations $\vee$ and $\wedge$ in this equation suppress all but one function value of f that corresponds to the respective value of x_i. The operations are extended to multi-valued functions in the natural way.

Definition II.4 (Reduced interval partition) An independence interval partition is called *minimal* if it contains no neighboring split intervals that may be joined into an independence interval. An interval partition $I(P_i) = \{I_1, I_2, \ldots, I_{p_i}\}$ is *ordered* if the higher bounds of all split intervals build an increasing sequence with respect to their indices. An independence interval partition which is minimal and ordered is called *reduced*.

Theorem 1: A reduced independence interval partition of a domain P_i is *unique*.

Proof: The proof of this is by contradiction. Assume that two different independence interval partitions $I(P_i)$ and $I'(P_i)$ exist which both are reduced. In the following, each possible case of being different is regarded.

- Assume that $I(P_i)$ contains more intervals than $I'(P_i)$: $|I(P_i)| > |I'(P_i)|$. Then $I(P_i)$ cannot be minimal since at least two of its intervals can be merged into one. With $|I(P_i)| < |I'(P_i)|$, the above holds in the general sense. Hence, either $I(P_i)$ or $I'(P_i)$ or both cannot be reduced.
- Assume that $|I(P_i)| = |I'(P_i)|$. If there exists a j with $1 \le j \le p_i$ such that $I_j \in I(P_i)$ and $I'_j \in I'(P_i)$ are

different, then $I(P_i)$ or $I'(P_i)$ is not ordered or both $I(P_i)$ and $I'(P_i)$ are not minimal. In both cases either $I(P_i)$ or $I'(P_i)$ or both are not reduced.

Consequently, both partitions $I(P_i)$ or $I'(P_i)$ cannot be reduced and different at the same time. Hence, a reduced independence interval partition must be unique. ■

B. Structure

An example interval decision diagram is shown in Figure 2. It represents a function $f(x_1, x_2, x_3)$ with the variable domains $P_i = [0, \infty)$ and the range $Q_f = \{a, b\}$ denoted as

$$f(x_1, x_2, x_3) = \begin{cases} a & \text{if } x_1^{[0,3]} \cdot x_2^{[0,5]} \vee x_1^{[4,\infty)} \cdot x_3^{[0,7]} \\ b & \text{otherwise} \end{cases} .$$

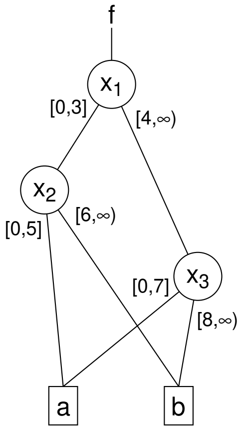


Fig. 2. Example interval decision diagram.

Though the edges are not depicted by arrows, the graph is directed such that all edges are oriented from top to bottom. To determine the function value of f, the function graph has to be traversed beginning at the top such that an edge is taken with an associated interval including the value of the variable the actual node is labeled with. When one of the terminal nodes is reached, its value represents the searched function value. The IDD edges are labeled with integer intervals. [20] gives an extension of the IDD concept described here where edge labels can consist of real intervals instead of integer ones.

Many kinds of discrete-valued functions with a finite range set can be described using IDDs. This can be achieved by employing one terminal node for each possible function value, similar to *multi-terminal BDDs* (MTBDDs) [21], [22]. IDDs can be regarded as a generalization of BDDs, MDDs, and MTBDDs as neither the values of the terminal nodes nor the function parameters are restricted to binary or integer values. MDDs' integer edge weights may be described by rudimentary one-element integer intervals with equal upper and lower bounds, as can BDDs' binary values.

C. Representation

IDDs are represented by canonical *function graphs*, similar to those of [18] and [4].

Definition II.5 (Function graph) A function graph G is a rooted, directed acyclic graph with a node set V containing two types of nodes. A *non-terminal* node $v \in V$ has as attributes an argument index $i = \mathit{index}(v)$, an independence interval partition $\mathit{part}(v) = I(P_i) = \{I_1, I_2, \ldots, I_{p_i}\}$, and $p_i = |\mathit{part}(v)|$ children $\mathit{child}_k(v) \in V, 1 \le k \le p_i$. The split intervals $\mathit{int}_k(v) = I_k \in I(P_i)$ of the partition are assigned to the *corresponding* graph edges $\big(v, \mathit{child}_k(v)\big) \in E$. A *terminal* node v has a value $\mathit{value}(v) \in Q_f$ as an attribute.

We define the correspondence between function graphs and multi-valued functions as follows.

Definition II.6: The function f_v associated with a node $v \in V$ of a function graph G is defined recursively as:

- If v is a terminal node, then $f_v = \mathit{value}(v)$,
- if v is a non-terminal node with $\mathit{index}(v) = i$, then f_v is the function described by the Boole-Shannon expansion as described in equation (1), thus

$$f_v = \bigvee_{I_j \in \mathit{part}(v)} x_i^{I_j} \wedge f_{\mathit{child}_j(v)}. \tag{2}$$

The function denoted by the graph G is associated with its root node. A subgraph G_v of G *induced* by a node v contains all edges and nodes reachable from v.

In the context of decision diagrams, functions are considered to be equivalent to their associated nodes. Hence, a function f_v associated with a node v with variable index i may be represented by a $(p{+}1)$-tuple $f_v = \big(x_i, (I_1, F_1), \ldots, (I_p, F_p)\big)$, where (I_k, F_k) denote pairs each consisting of split interval $I_k = \mathit{int}_k(v)$ of the interval partition $\mathit{part}(v)$ and the function F_k associated with the respective corresponding child node $\mathit{child}_k(v)$. This description is directly associated with the Boole-Shannon expansion mentioned above.

Definition II.7 (Ordered function graph) A function graph is *ordered* if for any adjacent pair of non-terminal nodes $\big(v, \mathit{child}_k(v)\big)$ we have $\mathit{index}(v) < \mathit{index}\big(\mathit{child}_k(v)\big)$.

The term *layer* describes either all non-terminal nodes having the same index or all terminal nodes. In the following, only ordered function graphs are considered.

Definition II.8 (Reduced function graph) A function graph G is *reduced* if

1. each non-terminal node v has at least two different children,
2. it does not contain two distinct nodes v and v' such that the subgraphs rooted by v and v' are *isomorphic* (as defined in [4]), and
3. the independence interval partitions $\mathit{part}(v)$ of all non-terminal nodes v are reduced.

One of the major results of this paper is described in the following.

Theorem 2: For any describable multi-valued function f, there is a unique reduced function graph denoting f. Any other function graph denoting f with the same variable ordering contains more nodes.

Proof: The proof proceeds along the same lines as those in [4] and [18], by induction on the size of the dependence set D_f. The dependence set of a function f is the set of arguments that f depends upon, i.e.,

$$D_f = \{x_i | \exists b, c \in P_i, b \neq c, \text{ such that } f|_{x_i=b} \neq f|_{x_i=c}\}.$$

The following statements are assumed without further explanation.

- As the partitions of a reduced function graph are reduced, the evaluation path from root to a terminal node is unique for given variable values.
- Each node in a function graph is reachable from its root node.
- If a function graph G is reduced, then any subgraph G_v induced by a node v is reduced.

For $|D_f| = d > 0$, if x_i is the argument with the lowest index i that f depends upon, then the root node v of the reduced function graph G of f has $index(v) = i$. This follows directly from the fact that G is reduced and thus all its interval partitions are reduced as well. If the root node had $index(v) < i$, then it would have exactly one leaving edge with the reduced trivial interval partition $I(P_i) = \{P_i\}$ consisting of only one split interval. Therefore, it would have just one child. Thus, G would not have been reduced. If the root node had $index(v) > i$, then x_i would not be element of the dependence set D_f which contradicts the assumption.

For $|D_f| = 0$, the root node and also the only node of the reduced function graph G of the constant function f is the terminal node v with $value(v) = f(x_1, x_2, \ldots, x_n) = t$, hence G is unique. To demonstrate this, suppose that the root node v is non-terminal with $index(v) = i$, then it has only one leaving edge with the trivial partition as f is independent of x_i. Therefore, it has just one child. Hence, G was not reduced. The graph cannot contain terminal nodes with value unequal to t, since every node is reachable, and that would imply a path which evaluates to a value other than t.

Suppose that for all functions f with $|D_g| < d$ the following holds: If the function graph H of g is reduced, the graph is unique. Then for each function f with $|D_f| = d$, if its function graph G is reduced, then the graph is unique. The proof of this is subdivided into two parts. First, we claim that as all subfunctions g associated with nodes w with $index(w) > i$ satisfy $|D_g| < d$, all those subfunctions are different. Otherwise, the subgraphs of two equal subfunctions would be isomorphic due to the uniqueness of function graphs. This would violate the premise that G is reduced. In particular, all children of the root node v represent different functions represented by non-isomorphic subgraphs. Now, consider the Boole-Shannon expansion of f with respect to x_i as described above. Let v be the root of G with $index(v) = i$. Its independence interval partition $part(v)$ is unique. Consequently, the number of outgoing edges from v is fixed, and their labels—associated intervals—are unique. Moreover, the edge with label I_j ends in the node $child_j(v)$ which is unique. Otherwise, there would exist two identical subfunctions associated with nodes with $index(w) > i$.

Finally, we can prove that of all graphs representing f, only the reduced function graph has a minimum number of nodes. Let G be a function graph with the minimum number of nodes. Since the reduced function graph is unique, if G is not the reduced graph this would imply that G either has a node v such that for all children $child_j(v)$ and $child_k(v)$, $child_j(v) = child_k(v)$, or that it contains two distinct nodes v and v' such that the subgraphs rooted by v and v' are isomorphic. In either case, we can reduce the number of nodes in G, contradicting the premise that G has the minimum number of nodes. Hence, G must be the unique reduced function graph. ∎

C.1 Implementation

The canonicity property described in Theorem 2 is of fundamental importance with regard to the frequent equality tests performed during fixpoint computations for most symbolic verification methods. In general, the functions used in a computation are stored not in several IDDs, each with a specific function, but in a common IDD that represents all functions at once, analogously to efficient BDD implementations. Thus, equal cofactors, represented by identical partial IDDs that would have to be stored separately, are shared among the IDDs resulting in a significant reduction of the total number of IDD nodes.

Further, the canonical IDD form is supported which results in only constant-time equality tests. This is because only IDD nodes—more precisely, pointers to nodes or unique node ids, respectively—have to be compared with each other instead of whole graphs. Hence, two functions are equal iff the IDD nodes representing them are identical.

Similarly to BDDs, the common IDD structure is stored using a hash table called *unique table* [23] for efficiency as well as for canonicity. The canonical form is maintained by looking up in the unique table each time before inserting a node into the IDD. A new node is created only if no equivalent node exists in the unique table. Otherwise, this existing node is used instead of inserting a new one. The unique table allows a single multi-rooted directed acyclic graph to represent any number of Boolean functions simultaneously.

We assume the functions to be represented all have the same n arguments $x_1, \ldots, x_n$. The variable ordering must be the same for all functions.

D. Reducing a Function Graph

Similarly to BDDs and related decision diagrams, several reduction rules exist to transform any IDD into a reduced form. An IDD is reduced if none of these rules can be applied. Two distinct nodes with the same index and same interval partitions are called *equal-childed* if all children of one of the nodes equal the corresponding children of the other node. *Neighboring edges* are edges starting in the same node, labeled with neighboring intervals, and ending in the same child node. The reduction rules are:

1. If all outgoing edges of a non-terminal node end in the same child, this node must be eliminated and its entering edges must be redirected into the child node.
2. If two equal-childed nodes exist or two distinct terminal nodes with the same value, in either case these

two nodes must be joined into one node by eliminating the other node and redirecting its incoming edges into the resting node.

3. If the interval partition of a non-terminal node is not reduced, it must be transformed into a reduced form as follows. The split intervals and their corresponding children are ordered. Neighboring edges are joined into one edge by uniting their split intervals.

A node which has to be removed due to rule 1 is called *obsolete*. The rule holds particularly if the interval partition consists of one single split interval only. The three reduction rules correspond directly to the respective properties of a reduced function graph introduced in Definition II.8. If none of the rules can be applied any more, all necessary conditions of a function graph to be reduced are satisfied. Thus, the result always is a reduced function graph. Figure 3 shows an example for the reduction of a function graph. For the examples shown, the interval $[0, \infty)$ is used as variable domain if not otherwise mentioned. Interval covers must be complete with respect to this domain set.

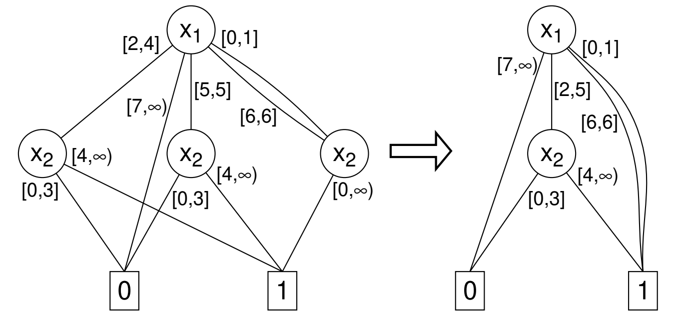


Fig. 3. Graph reduction.

III. IDDs Representing Boolean Functions

For conventional symbolic model checking of the models described above, mainly Boolean functions[3] over integer variables are of importance. Hence, in this section, only IDDs are considered that represent Boolean functions over integer intervals. Their terminal nodes may have only the values 0 or 1 and thus are called *0-* or *1-terminal nodes*, respectively. With the use of such kind of IDDs, truth functions and propositions such as $f(x_1, x_2) = (x_1 \leq 7) \wedge (x_2 = 3) \vee (x_2 \geq 6) = x_1^{[0,7]} \cdot x_2^{[3,3]} \vee x_2^{[6,\infty)}$ are describable. As mentioned above, IDDs have been extended toward real domains in [20].

A. Interval Cover Disjunction

Some operations described below make use of the transformation of a non-disjoint interval cover $I(P_i) = \{I_1, I_2, \ldots, I_{p_i}\}$ into a partition $\tilde{I}(P_i) = \{\tilde{I}_1, \tilde{I}_2, \ldots, \tilde{I}_{\tilde{p}_i}\}$ consisting of *greatest common intervals* (GCIs). By uniting a certain number of those split intervals, each original interval may be expressed. Interval cover disjunction is performed by splitting the $I_1, I_2, \ldots, I_{p_i}$ and then removing duplicates such that the whole variable domain P_i remains covered but none of the elements of P_i is covered by more than one interval in $\tilde{I}(P_i)$.

Figure 4 shows an example interval cover disjunction transforming $I(P_i) = \{[-1, 2], [1, 3], [3, 4]\}$ in the upper part into $\tilde{I}(P_i) = \{[-1, 0], [1, 2], [3, 3], [4, 4]\}$ in the lower part. The intermediate steps shown in the figure are described below.

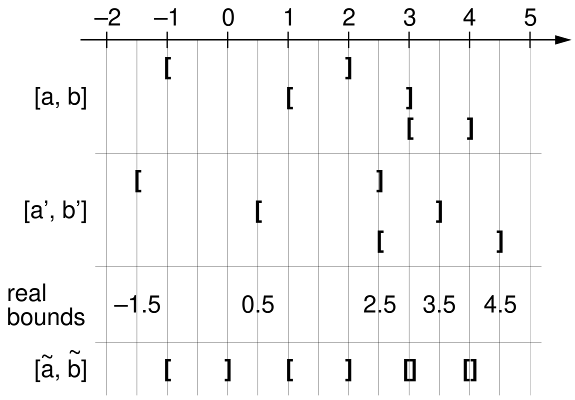


Fig. 4. Example of interval cover disjunction.

Formally, the GCI partition $\tilde{I}(P_i)$ of a cover $I(P_i)$ is defined as the unique minimal interval partition with the property

$$I_j = \bigcup_{k=1}^{\tilde{p}_i} (B_{jk} \cap \tilde{I}_k), \forall j \text{ with } 1 \leq j \leq p_i, \tag{3}$$

where $B_{jk} \in \{[\,], (-\infty, \infty)\}$. Here, each B_{jk} behaves like a switch which is $(-\infty, \infty)$ for "on" iff the GCI $\tilde{I}_k$ is included in the original interval I_j, otherwise B_{jk} is $[\,]$ for "off". The postulated property of $\tilde{I}(P_i)$ being minimal must hold with regard to the number $\tilde{p}_i$ of GCIs. Thus, it is supposed that all $\tilde{I}_k$ are as large as possible and no neighboring GCIs $\tilde{I}_k$ may be joined by union without violating the above equation.

An efficient procedure to determine the GCIs of an interval cover is described in Table I. First, each original integer interval $[a, b]$ is transformed into an intermediate real interval $[a', b']$, including the original interval as shown in Figure 4. This is achieved by adding the value $-\frac{1}{2}$ to the lower and $\frac{1}{2}$ to the upper bound of the original interval. The resulting real bounds can be considered as borders separating the GCIs. All real bounds are stored using the list L of real numbers. After sorting the list elements in increasing order, the disjoint intervals are determined by back-transforming successive real bounds into integer GCIs $[\tilde{a}, \tilde{b}]$. The described interval cover disjunction algorithm can be implemented with worst-case complexity $O(p_i \cdot \log p_i)$ where $p_i = |I(P_i)|$.

[3]The term "Boolean function" in this work denotes functions of Boolean range, while their domain is arbitrary. This class of functions sometimes is referred to as pseudo-Boolean functions in literature, whereas both the domain and the range of a Boolean function then must be Boolean.

TABLE I
INTERVAL COVER DISJUNCTION.

$\mathit{determineGCIs}\big(I(P_i)\big)$:
  for each interval $[a,b] \in I(P_i)$
    $[a',b'] = [a-\frac{1}{2}, b+\frac{1}{2}]$;
    add bounds a' and b' to list L;
  sort list L in increasing order and remove duplicates;
  for each pair (a',b') of successive elements in L
    $[\tilde{a},\tilde{b}] = [a'+\frac{1}{2}, b'-\frac{1}{2}]$;
    add $[\tilde{a},\tilde{b}]$ to $\tilde{I}(P_i)$;
  return $\tilde{I}(P_i)$;

B. If-Then-Else Operator

The *If-Then-Else* operator (ITE) [4] constructs the graph for the function obtained by composing two functions. In the case of non-Boolean IDDs, it has to be replaced by the $CASE$ operator described in [18]. For the sake of clarity, only ITE is considered here. ITE is a ternary Boolean operation directly derived from the Boole-Shannon expansion and is denoted

$$ITE(F,G,H) = F \cdot G \vee \neg F \cdot H.$$

Thus, it means: if F then G else H. ITE can be used to apply all two-variable Boolean operations on IDDs within a single algorithm. For instance, $AND(F,G)$ may be denoted as $ITE(F,G,0)$, $OR(F,G)$ as $ITE(F,1,G)$, or $NOT(F)$ as $ITE(F,0,1)$.

Let $Z = ITE(F,G,H)$ and let x be the top variable of F, G, and H, i.e., the variable at the common highest layer of their IDDs and thus with the lowest index. Analogously to [23], using the Boole-Shannon decomposition ITE is recursively defined as

$$ITE(F,G,H) = \Big(x, \big(I_1, ITE(F_{x^{I_1}}, G_{x^{I_1}}, H_{x^{I_1}})\big), \ldots, \big(I_p, ITE(F_{x^{I_p}}, G_{x^{I_p}}, H_{x^{I_p}})\big)\Big).$$

The terminal cases of this recursion are $ITE(1,F,G) = ITE(0,G,F) = ITE(F,1,0) = F$.

At the beginning of the calculation of $ITE(F,G,H)$, the interval partitions of those top nodes of F, G, and H that are on the common highest layer of the three IDDs are decomposed mutually, while their original partitions remain unchanged. This mutual decomposition is performed by building the union of all concerned partitions, resulting in an interval cover $I'(P_i)$ that in general is not disjoint. Then this cover is transformed as described in Section III-A into an independence interval partition. The resulting partition $I(P_i) = \{I_1, I_2, \ldots, I_p\}$ is used for the recursive ITE calculation as described below. As this is a mutual decomposition, the possibility to restrict F, G, and H to the split intervals I_k by building their cofactors is guaranteed. Hence, the original IDD remains reduced (if it was before), and at most the resulting new nodes possibly have to be reduced before insertion into the IDD. The ITE algorithm is sketched as pseudo code of the function $ite(F,G,H)$ in Table II.

TABLE II
THE *ITE* ALGORITHM.

$ite(F,G,H)$:
  if terminal case reached
    return result of terminal case;
  let x be the top variable of $\{F,G,H\}$;
  $I'(P_i) = \emptyset$;
  for each $E \in \{F,G,H\}$:
    if x is top variable of E
      let v be the top node of E;
      $I'(P_i) = I'(P_i) \cup part(v)$;
  $I(P_i) = \{I_1, I_2, \ldots, I_p\} = \mathit{determineGCIs}\big(I'(P_i)\big)$;
  for each $I_k \in I(P_i)$:
    set $child_k = ite(F_{x^{I_k}}, G_{x^{I_k}}, H_{x^{I_k}})$;
  $\mathit{mergeEqualChildedNeighbors}\big(I(P_i)\big)$;
  if $|I(P_i)| = 1$
    return $child_1$;
  return $\mathit{findOrAddInIDD}\big(x, (I_1, child_1), \ldots, (I_k, child_k)\big)$;

The terminal cases and their results are given above. The function $\mathit{mergeEqualChildedNeighbors}\big(I(P_i)\big)$ iterates through the intervals I_k within $I(P_i)$ and merges each two neighboring I_k of which the corresponding children $child_k$ equal each other. Hence, the resulting interval partition is reduced.

Within function $\mathit{findOrAddInIDD}(v)$, first the unique table of the IDD is searched for the argument node v. If such a node already is included in the unique table, it is returned. Otherwise, a new node with the specified properties is created and inserted into the unique table. Since the new node is reduced, the reduced form of the IDD is maintained.

Analogously to BDDs, a *computed table* [4], [23] consisting of a hash-based cache table of already computed results of $ITE(F,G,H)$ is employed for efficiency. The computed table serves as a memory function and returns previously calculated (and not yet overwritten) results without further computation.

The worst-case complexity of the ITE algorithm is $O(|F| \cdot |G| \cdot |H| \cdot p \cdot \log p)$, where $|F|$, $|G|$, and $|H|$ denote the numbers of IDD nodes in the respective graphs, and p is the maximum number of intervals in any interval cover of the three IDDs[4]. In case bounded variable domains $P_i = [a_i, b_i]$ are used, the maximum over the domain widths $b_i - a_i + 1$ is an upper bound of p. The complexity has been determined on the usual assumption of constant time lookup and insert in the computed table. With regard to the unique table, lookup and insert may depend linearly on p due to the p-dimensional characteristic of the table. The highest complexity of all operations within $ite(F,G,H)$ is that of

[4]In order to determine the complexity, p cannot be bounded by a graph size such as $|F|$ since there exist functions needing an infinite number of edges but only one non-terminal node, e.g., periodically alternating functions.

the interval cover disjunction which is $O(p_i \cdot \log p_i)$. Due to the computed table, $ite(F, G, H)$ is called at most once for each combination of nodes in F, G, and H. Hence, the total complexity is $O(|F| \cdot |G| \cdot |H| \cdot p \cdot \log p)$.

Figure 5 shows an example of the application of the If-Then-Else operator where $I = ITE(F, G, H)$. For the sake of clarity, the partial IDDs are depicted separately and not in one common IDD that represents all used functions. The calculation of the ITE operation is sketched in Table III. The structure of the resulting IDD can directly be read from the last row of this calculation.

TABLE III
Example of *ITE* computation.

$$
\begin{aligned}
I &= ITE(F,G,H) \\
&= \Big(a, \big([0,3], ITE(F_{a[0,3]}, G_{a[0,3]}, H_{a[0,3]})\big), \\
&\quad \big([4,4], ITE(F_{a[4,4]}, G_{a[4,4]}, H_{a[4,4]})\big), \\
&\quad \big([5,\infty), ITE(F_{a[5,\infty)}, G_{a[5,\infty)}, H_{a[5,\infty)})\big)\Big) \\
&= \Big(a, \big([0,3], ITE(B,0,H)\big), \big([4,4], ITE(1,0,H)\big), \\
&\quad \big([5,\infty), ITE(1,C,H)\big)\Big) \\
&= \Bigg(a, \bigg([0,3], \Big(b, \big([0,1], ITE(B_{b[0,1]}, 0, H_{b[0,1]})\big), \\
&\quad \big([2,2], ITE(B_{b[2,2]}, 0, H_{b[2,2]})\big), \\
&\quad \big([3,\infty), ITE(B_{b[3,\infty)}, 0, H_{b[3,\infty)})\big)\Big)\bigg), \\
&\quad \big([4,4], 0\big), \big([5,\infty), C\big)\Bigg) \\
&= \Bigg(a, \bigg([0,3], \Big(b, \big([0,1], ITE(0,0,D)\big), \\
&\quad \big([2,2], ITE(0,0,1)\big), \big([3,\infty), ITE(1,0,1)\big)\Big)\bigg), \\
&\quad \big([4,4], 0\big), \big([5,\infty), C\big)\Bigg) \\
&= \Bigg(a, \bigg([0,3], \Big(b, \big([0,1], D\big), \big([2,2], 1\big), \big([3,\infty), 0\big)\Big)\bigg), \\
&\quad \big([4,4], 0\big), \big([5,\infty), C\big)\Bigg)
\end{aligned}
$$

C. Graph Unification

Under certain circumstances—e.g., when constructing an IDD during initialization—a graph may be of importance that is similar to the function graph of an IDD but contains one or several non-disjoint interval covers instead of interval partitions. As the evaluation paths in such a graph are ambiguous, this cannot be a function graph based on the Boole-Shannon decomposition. Nevertheless, the transformation of an ambiguous graph into a function graph—the *unification*—is possible. If all possible evaluation paths for a given variable *assignment*—a given value for each variable—always end in the same terminal node, the graph represents a unique function even if it has an ambiguous structure. However, for assignments evoking evaluation paths to different terminal nodes an unambiguous behavior must be defined.

In the case of Boolean functions, we define a disjunctive behavior where a path to a terminal node with the Boolean value 1 covers all other paths of the same variable assignment. This behavior may be described by adapting the definition of the function f_v, associated with a node v, in equation (2) to interval covers, denoted as

$$f_v = \bigvee_{I_j \in cover(v)} x_i^{I_j} \wedge f_{child_j(v)}, \tag{4}$$

where $cover(v)$ replaces the partition $part(v)$ of node v. Hence, the value of the corresponding function is 1 for all assignments which have at least one evaluation path ending in the 1-terminal node, otherwise it is 0. In [20], a general method is described which transforms an ambiguous graph into a function graph.

IV. Interval Mapping Diagrams

Interval mapping diagrams are used to represent transition relations, for instance, in symbolic model checking. IMDs map a set described by a Boolean-valued IDD onto a new set—described by such IDD as well—by performing operations such as shifting or assigning some or all values of the IDD's decision variables. For interval shifting, a simple interval arithmetic is used.

A. Notation

The underlying interval mathematics consist of a simple unbounded interval arithmetic with the operators $+$ for *addition* and $-$ for *subtraction*, each over two intervals. Both arguments and the result of each operator are elements of $\mathbb{I}$. In the following, for the sake of brevity, the interval operators are defined for finite intervals only, but their finite bounds may be replaced by infinite ones. The addition or subtraction of empty intervals is not allowed.

Definition IV.1 (Interval addition) The addition of two intervals $[a, b]$ and $[c, d]$ is defined as $[a, b] + [c, d] = [a + c, b + d]$.
Thus, the resulting interval consists of all numbers that could have been the result of the addition of any two elements of the argument intervals. The upper and lower bounds of the resulting interval are the sums of the upper or lower bounds of the argument intervals, respectively.

Definition IV.2 (Interval subtraction) The subtraction of two intervals is defined as $[a, b] - [c, d] = [a, b] + [-d, -c]$. Hence, the result of the subtraction is the set of all numbers that could have been the result of the subtraction of any two elements of the argument intervals.

B. Representation

IMDs are represented by *mapping graphs* similar to the function graphs described in Definition II.5. Their edges

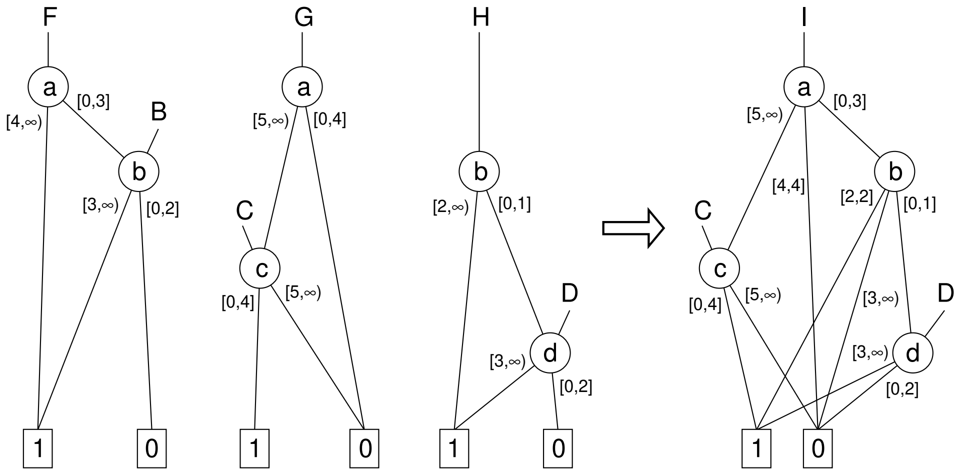


Fig. 5. If-Then-Else operator.

are labeled with *interval mapping functions* $f : \mathbb{I} \to \mathbb{I}$, mapping intervals onto intervals.

Definition IV.3 (Mapping graph) A mapping graph G is a rooted, directed acyclic graph with a node set V containing two types of nodes. A *non-terminal* node $v \in V$ has as attributes an argument index $i = index(v)$, a set of interval mapping functions $func(v) = \{f_1, f_2, \ldots, f_n\}$, and $n = |func(v)|$ children $child_k(v) \in V, 1 \leq k \leq n$. The mapping functions $f_k(v)$ are assigned to the *corresponding* graph edges $\big(v, child_k(v)\big) \in E$. V contains exactly one *terminal node* v with $value(v) = 1$.

Though the terminal node in the above definition is not absolutely necessary, it is introduced because of the similarity of IMDs and IDDs. The interval mapping function $f(I) = I$, mapping an interval onto itself, is called *neutral*.

C. Interpretation

Informally, the functional behavior of IMDs may be described as "set flow", similar to that of dataflow. The data consists of sets represented by unions of intersected intervals of state variable values, as described in Section II-A and represented by an IDD. Beginning at the root node of an IMD, the set data can flow along each path until reaching the terminal node. Each IMD edge transforms the data according to the respective mapping function. More precisely, the mapping function maps each interval of the corresponding state variable included in the actual set description onto a transformed interval. The effect of this may be, e.g., to shift, shrink, grow, or assign the whole set along exactly one coordinate of the state space. Then the modified set data is transferred to the next IMD node corresponding to another state variable where the transformation continues.

If an interval is mapped onto the empty interval, this is a degenerate case as the set is restricted to an empty set, i.e., the set data effectively does not reach the end of the computation along this path. From a global view, the set data flows through the IMD from its top to the bottom along all paths in parallel and finally is united in the terminal node to the resulting set. The algorithm to achieve this behavior will be described in detail in Section V. Section VI describes how an IMD is derived from a given model and gives two examples.

D. Reducing a Mapping Graph

Unlike IDDs, IMDs in general have no canonical form. Nevertheless, depending on the structure of the contained mapping functions a partial reduction often is possible. As no equality test or *ITE* operations of IMDs must be performed in contrast to IDDs, the lack of canonicity is no general limitation. However, smaller IMDs lead to a reduction of the computational complexity during image computation.

Similarly to IDDs, two distinct nodes with the same index and same mapping function sets are called equal-childed if all children of one of the nodes equal the corresponding children of the other node. IMDs may be reduced using the three following reduction rules:

1. If a non-terminal node has only one leaving edge and this edge is labeled with a neutral mapping function, the node is eliminated, and its entering edges are redirected into the child node.
2. Equal-childed nodes are joined into one by eliminating all but one and redirecting all incoming edges into the resting node.
3. Two edges starting in the same node and labeled with the same mapping function are replaced by such an edge to a new node having the same index as the higher one of the two child nodes. If both child nodes are on the same layer and non-terminal, the new node obtains all their leaving edges. If they are on different layers, the new node obtains all edges of the higher child node and an edge labeled with a neutral mapping function to the lower child node.

E. Predicate Action Diagrams

Predicate action diagrams (PADs) are a restricted form of IMDs dedicated to describe the transition behavior especially of process networks and similar models.

Definition IV.4 (Predicate action diagram) A predicate action diagram is an interval mapping diagram containing only the following mapping functions:

$$f_+(I) = \begin{cases} I \cap I_P + I_A & \text{if } I \cap I_P \neq \emptyset \\ [\,] & \text{otherwise} \end{cases}$$

and

$$f_=(I) = \begin{cases} I_A & \text{if } I \cap I_P \neq \emptyset \\ [\,] & \text{otherwise} \end{cases} ,$$

where f_+ is the *shift function* and $f_=$ the *assign function.* I_P is called *predicate interval* and I_A *action interval.* Each mapping function within a PAD can have different intervals.

Thus, the interval mapping functions have one among two possible structures. Both types first intersect the argument interval with the predicate interval and then—if the result is not an empty interval—either shift the resulting interval up or down or assign a default interval. The shift is performed by interval addition of the action interval. The assignment of a default interval is achieved by returning the action interval as the result of the interval mapping function.

The combination of predicate and action interval parameterizes the mapping function and completely defines its behavior. The syntax $I_P/{+}I_A$ is used for the shift function f_+ and $I_P/=I_A$ for the assign function $f_=$. The shift about $I = [a,b]$ in the reverse direction corresponding to interval subtraction is achieved by addition of $-I = [-b,-a] = I_A$ and is denoted as $I_P/-I$.

For PADs, the neutral interval mapping function is a shift function with the respective variable domain as the predicate interval and $[0,0]$ as the action interval, corresponding to $P_i/+[0,0]$. Assign functions cannot be neutral.

V. Image Computation

A. Definition

In general, symbolic model checking is based on Boolean functions representing characteristic functions of sets containing system states. The characteristic function of a set is a function with a value of 1 for those arguments that are elements of the set and 0 otherwise. Let $x = (x_1, x_2, \ldots, x_n)$ be a vector describing a system state. Then a state set S is represented by its characteristic function $s(x)$ with

$$s(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases} . \tag{5}$$

The transition behavior of the system is described by a transition relation. Let x and x' be system state vectors—x the one before and x' its successor after a transition—and $\delta(x,x')$ be a characteristic function representing the transition relation T, i.e.,

$$\delta(x,x') = \begin{cases} 1 & \text{if } (x,x') \in T \\ 0 & \text{otherwise} \end{cases} . \tag{6}$$

In symbolic formal verification, an important operation is the application of a transition relation to a set of system states such that the set of all successor or predecessor states, respectively, is calculated in one single computation. This process is called *image computation* where the image $\mathit{Im}(S,T)$ of a set S of system states with respect to the transition relation T represents the set of all states that may be reached after exactly one valid transition from a state in set S. The following definitions are taken from [24]. The image operator is defined as $\mathit{Im}(S,T) = \{x' | \exists x \text{ with } s(x) \wedge \delta(x,x')\}$. The inverse image $\mathit{PreIm}(S,T)$ represents all states that *can* reach a state in S after one transition and is defined as $\mathit{PreIm}(S,T) = \{x | \exists x' \text{ with } s(x') \wedge \delta(x,x')\}$. $\mathit{BackIm}(S,T)$ represents the set of states that after one transition *must* end up in S, denoted as $\mathit{BackIm}(S,T) = \{x | \forall x' \text{ holds } \delta(x,x') \implies s(x')\} = \neg\mathit{PreIm}(\neg S,T)$. The last operator behaves in some sense as the inverse of the image operator because $\mathit{BackIm}(S,T)$ is the largest set such that $\mathit{Im}\big(\mathit{BackIm}(S,T),T\big) \subseteq S$.

B. Using IDDs and IMDs

Figure 6 shows an example process network with uncertain consumption and production rates represented by intervals (depicted as edge labels). It is similar to a computation graph [9] where the consumption rate is independent of the threshold—denoted as a condition here. For instance, process A is enabled if queue u contains at least two tokens and at the same time queue v contains at least one token. If A is then executed, it consumes one or two tokens from u and one token from v, and produces one to three tokens in queue w.

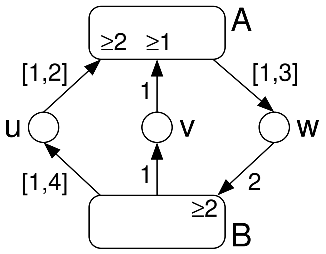


Fig. 6. Example process network.

Figure 7 b) shows the corresponding transition relation represented by the IMD T, Figure 7 a) an example state set IDD S. The state set is described as $s(u,v,w) = (v = 0) \vee (1 \le v \le 2) \wedge (w = 0)$, the transition relation as $\delta(u,v,w,u',v',w') = (u \ge 2) \wedge (u - u' \in [1,2]) \wedge (v \ge 1) \wedge (v' = v - 1) \wedge (w' - w \in [1,3]) \vee (u' - u \in [1,4]) \wedge (v' = v + 1) \wedge (w \ge 2) \wedge (w' = w - 2)$, for $u,v,w,u',v',w' \in [0,\infty)$. Note that $P_i = \mathbb{Z}$ is assumed here with regard to the variable domains of the IDD in Figure 7 a). If $P_i = \mathbb{Z}_{\ge 0}$ were assumed, both nodes having edges labeled with $[0,\infty)$ would be obsolete.

The left-hand path of the IMD in Figure 7 b) corresponds to the transition behavior of process A in Figure 7 a), while the right-hand path corresponds to process B. The proce-

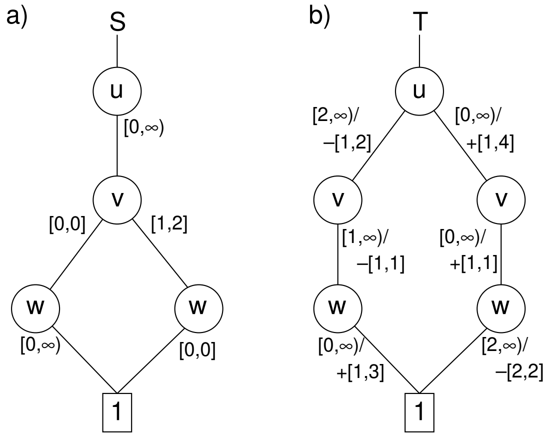


Fig. 7. State set IDD and transition relation IMD.

dure of deriving an IMD from a given process network is described in Section VI-A.1.

For a clearer view, an IDD representation is used in the figures where the 0-terminal node and all its incoming edges are omitted. Thus, for each non-terminal node, all split intervals that miss for a complete interval partition have to be added implicitly and connected with the invisible 0-terminal node. This description has no impact on the real internal representation of an implementation. It should not be confused with the DD class of *zero-suppressed BDDs* (ZBDD) [25] which are a BDD derivative based on modified reduction rules.

C. Computation

In this section, we describe how to perform image computation using IDDs and IMDs. Conventionally, as mentioned in Section I, the transition relation is represented as a BDD that explicitly stores all valid combinations of system state and successor state. Image computations are performed using Boolean operations such as existential and universal quantification $\exists$ and $\forall$, respectively—which internally can be computed using the If-Then-Else operator *ITE*, see [26], for instance. This strategy is possible for IDDs as well.

A more advanced technique is described in the following. A major advantage of IMDs in contrast to BDDs is the possibility of efficiently storing only mapping functions describing partial transition behaviors instead of many state variable value pairs. In this way, image computations cannot any longer be performed as usual. Instead, an IMD representing the transition relation is combined with an IDD storing a set of system states in a way that resembles the piecewise application of an operator to two sets, similarly to Bryant's *Apply* operator [4]. First, we describe a general form of transition relations based on IMDs. Then we concentrate on further restrictions using PADs which allow the efficient representation of state distances combined with the corresponding transition conditions.

C.1 Forward mapping

Using the described mapping functions, the image computation with IDDs and IMDs is performed. It requires an IDD S for the characteristic function $s(x_1, x_2, \ldots, x_n)$ of a state set and an IMD T for the characteristic function of the transition relation. The final result is a reduced IDD S' for the characteristic function $s'(x'_1, x'_2, \ldots, x'_n)$ of the set of successor states.

This image operation is performed recursively by the function *mapForward*(v, w)—over an IDD node v and an IMD node w—as sketched partially in Table IV. The resulting IDD S' is constructed recursively by traversing depth-first both source graphs. Here, new edges and nodes are created in a reduced manner, resulting from the interval mapping application by maintaining the respective graph structures. This computation is always based on reduced IDDs. The operation is similar to the *Apply* operation described in [4].

Obviously, whenever the empty interval [ ] is the result of an interval operation, the recursion during image computation is terminated, continuing with the next edge. Omitted layers in the transition relation IMD are implicitly treated as "dummy" mapping functions returning their argument interval without changes. Omitted IDD layers implying quasi-obsolete nodes are treated separately. Only those edges have to be concerned that are not ending in the 0-terminal node.

As a memory function, a hash-based cache table of computed results for pairs of v and w is maintained—omitted in Table IV—such that an included result may be returned without further computation, analogously to the *ITE* computation described in Section III-B.

The worst-case complexity of this image computation algorithm is $O(|S|^3 \cdot |T| \cdot f \cdot p^2 \cdot \log p)$, where $|S|$ and $|T|$ denote the numbers of nodes in the respective graphs, and $p = \max_v \{ \, |part(v)| \, \}$ and $f = \max_w \{ \, |func(w)| \, \}$ denote the maximum numbers of edges of any node in IDD S or IMD T, respectively. The complexity of *mapForward*(v, w) is determined by the third block of the algorithm of which the inner part is executed at most $p \cdot f$ times. Its dominating part is the disjunction $v_{res} \vee \tilde{v}$ of which the complexity[5] is $O(|S|^2 \cdot p \cdot \log p)$. Since *mapForward*$(v, w)$ is called at most once for each combination of nodes in S and T due to the use of the computed table, the resulting complexity is $O(|S|^3 \cdot |T| \cdot f \cdot p^2 \cdot \log p)$.

C.2 Backward mapping

As described above, the mapping functions are used for forward traversal during image computation with $Im(S, T)$, e.g., for reachability analysis where time proceeds in forward direction. To perform symbolic model checking using $PreIm(S, T)$ and $BackIm(S, T)$, the reverse direction is necessary, thus some kind of inverse mapping functions must be used. For IMDs, depending on the structure of the mapping functions, this inversion is not always pos-

[5] $OR(F, G)$ can be calculated as $ITE(F, 1, G)$, and the complexity of $ITE(F, G, H)$ is $O(|F| \cdot |G| \cdot |H| \cdot p \cdot \log p)$ as shown in Section III-B.

TABLE IV
FORWARD MAPPING FOR IMAGE COMPUTATION.

```
mapForward(v, w) :
  if v = TN_0
     return TN_0;
  if (v = TN_1) ∧ (w = TN_1)
     return TN_1;
  v_res = TN_0;
  if index(v) < index(w)
     for each I_j ∈ part(v)
        if (child_j(v) ≠ TN_0)
           v_c = mapForward(child_j(v), w);
           if v_c ≠ TN_0
              create new node ṽ with index(v);
              add new edge with interval I_j
                ending in v_c to ṽ;
              complement ṽ with edges to TN_0;
              if ṽ is obsolete
                 ṽ = child_1(ṽ);
              v_res = v_res ∨ ṽ;
  else if index(v) > index(w)
     for each f_k ∈ func(w)
        I_res = f_k(P_index(w));
        if I_res ≠ []
           v_c = mapForward(v, child_k(w));
           if v_c ≠ TN_0
              create new node ṽ with index(w);
              add new edge with interval I_res
                ending in v_c to ṽ;
              [...]
  else
     for each I_j ∈ part(v)
        for each f_k ∈ func(w)
           if (child_j(v) ≠ TN_0)
              I_res = f_k(I_j);
              if I_res ≠ []
                 v_c = mapForward(child_j(v),
                   child_k(w));
                 if v_c ≠ TN_0
                    create node ṽ with index(v);
                    add new edge with interval
                      I_res ending in v_c to ṽ;
                    [...]
  return v_res;
```

sible because the function result has to be representable by an interval. However, PADs have valid inversions as described below.

The kind of applied interval mapping functions depends on the direction of the image operation that is performed. To determine the image $Im(S,T)$ of state set S with respect to transition relation T, regular mapping functions are used as described for $mapForward(v,w)$ to determine all values a state variable could have after a transition. Accordingly, for the calculation of the inverse image $PreIm(S,T)$, the inverse mapping functions are used to determine all values a state variable could have had before a transition. Therefore, a function $mapBackward(v,w)$ exists which is analogous to $mapForward(v,w)$, except for the mapping functions. For the calculation of $BackIm(S,T)$, its equivalence to $\neg PreIm(\neg S,T)$ as specified above is used, thus no additional operation is necessary.

D. Image Computation With PADs

Image computation here will be demonstrated with an example. As mentioned above, PADs are dedicated to perform image computation especially for process networks and related models of computation because the state distance combined with the respective firing condition may be stored more efficiently than many state pairs. The state distance between two system states x and x' is defined as $\Delta x = x' - x$ (note that the variables involved are vectors). Thus, according to equation (6), the transition relation T may be described as the characteristic function

$$\hat{\delta}(x,\Delta x) = \begin{cases} 1 & \text{if } (x, x+\Delta x) \in T \\ 0 & \text{otherwise} \end{cases} . \qquad (7)$$

In Figure 7 b), only shift functions f_+ are used as mapping functions. In the case of buffer contents limited to non-negative numbers, the predicate intervals must ensure that the resulting state variables x_i' cannot become negative after a transition, i.e., the firing or enabling condition has to be satisfied. The action intervals perform the consumption and production of tokens by shifting intervals of state set variable values. Hence, the action intervals represent the state distance Δx.

The mapping functions could have any structure describing valid system states and transitions. This includes the use of assign functions $f_=$ as mapping functions, e.g., for most finite state systems or process networks with *safe* buffers—buffers with bounded capacity of 1. The transition relation PAD of a binary coded system must replace Boolean operations such as *AND* and *OR* used in a BDD by their corresponding mapping functions. Thus, valid combinations of state variable values before and after a transition are substituted by the corresponding predicate and action intervals which are subsets of $[0,1]$.

Figure 8 shows the application of the image operator and its result. In the left-hand part, for a better understanding an intermediate IDD is depicted that is not really built during the computation but is more comprehensible than the final result. It may be transformed by unification into the reduced IDD returned by $mapForward(v,w)$ which is shown in the right-hand part of Figure 8—note that for efficiency $mapForward(v,w)$ is implemented without the need for unification. The dashed edges labeled with empty intervals represent such locations where the recursion of $mapForward(v,w)$ is terminated due to an empty interval resulting.

For backward traversal, the mapping functions of PADs are inverted as follows.

Definition V.1 (Inverse PAD mapping functions) The inverse shift function is denoted as

$$f_+^{-1}(I) = (I - I_A) \cap I_P$$

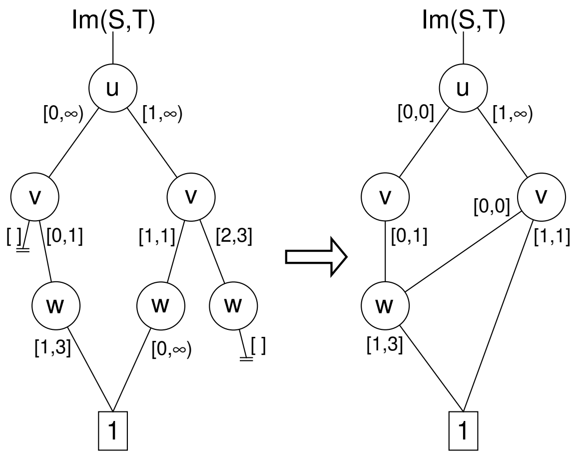


Fig. 8. Computation of $Im(S,T)$: IDD before and after unification.

and the inverse assign function as

$$f_{=}^{-1}(I) = \begin{cases} I_P & \text{if } I \cap I_A \neq \emptyset \\ [\,] & \text{otherwise} \end{cases} .$$

VI. Symbolic Model Checking

Symbolic model checking enables the verification of certain temporal properties of state transition systems, where the explicit construction of an entire reachability graph is avoided by implicitly describing it using symbolic representations. Often, the propositional branching-time temporal logic CTL (*computation tree logic*) is used to specify the desired system properties [26]. To verify such CTL formulae, operations such as *ITE* and image computation are used.

The most frequently employed form of symbolic representation are BDDs and their derivatives, see, e.g., [2], [26], [27], [28]. To perform conventional symbolic model checking, the investigated system is described using BDDs to represent its transition relation and certain state sets. As mentioned in Section I, for process networks this is not a convenient form of representation for several reasons. Advantages of interval diagram techniques are:

- Using IMDs and IDDs for the representation of transition relation and state sets, respectively, avoids some undesirable limitations of BDDs and binary coding.
- The introduced image computation is well suited to process networks as only state distances are stored.
- The description is more compact as sets of state variable or state distance values are combined and represented by one IDD or IMD node.

A. Process Networks

A.1 Modeling Process Networks

For symbolic model checking, only the quantitative system behavior is considered, i.e., only the number of tokens in each buffer, not their values. The behavior of many process networks may be described by decomposing the transition behavior of each process into atomic transitions, changing the internal state of the process and consuming and producing tokens, depending on the internal state. For this decomposition, it is necessary that the process behavior can be described by means of a finite state machine. Recursive network structures are not allowed. Non-blocking read or blocking write semantics may also be represented. Even non-determinate models with multi-reader and multi-writer buffers such as, e.g., Petri nets are verifiable using IDDs and IMDs.

Each path in the transition relation IMD describes one possible state transition. The mapping functions along the path represent enabling conditions and the corresponding state variable changes. The transition is enabled if all conditions along the path are satisfied. Analogously to computation graphs [9], a threshold different from the consumption rate may be specified. Non-determinate consumption rates can easily be considered as intervals—introducing an additional degree of non-determinism. While changes of buffer contents are described using shift functions f_+, assign functions $f_=$ are used for internal state changes. The state variables are of either infinite domain—representing contents of unbounded buffers—or of finite domain—describing internal process states or bounded buffers.

A.2 Model Checking

Symbolic model checking of process networks comprises the whole well-known area of model checking concerning the detection of errors in specification or implementation. Examples are the mutual exclusion of processes or the guaranteed acknowledgement of requests. Properties may be described as CTL formulae and verified as usual [26].

Additionally, applications assisting in scheduling are possible. Boundedness can be determined either by computing the set of reachable states or by checking CTL formulae on the contents of buffers. For termination and deadlocks, respective CTL formulae may be formed easily. Additionally, the effect of certain scheduling policies on these measures may be investigated or improved. Deadlocks in artificially bounded process networks or inherent bounds may be detected. In this way, optimal schedules may be confirmed or even developed by determining least bounds and thus optimal static capacity limits for scheduling, constraining the necessary memory.

B. Petri Nets

In our context, a Petri net is a 6-tuple $G = (P, T, F, K, W, M_0)$ with $P \cap T = \emptyset$ and $F \subseteq (P \times T) \cup (T \times P)$. P and T are the sets of places and transitions, respectively. F is the flow relation including the graph arcs connecting places and transitions. K denotes the place capacities which may be positive integer numbers or infinite. W describes the weights of the arcs, i.e., the consumption and production rates. M_0 is the initial marking of G. The marking M of a Petri net denotes the actual number of tokens in each place. The behavior of a Petri net is defined as usual. A transition $t \in T$ is enabled for firing if all its incoming places contain at least as many tokens as the respective arc weights and all its outgoing places are able to absorb at least as many tokens as the respective arc weights without exceeding their place capacity.

B.1 Modeling Petri Nets

In order to describe the use of IDDs and IMDs for symbolic model checking of Petri nets, Figure 9 shows an example Petri net [29]. The Petri net places are of infinite capacity. Arc weights of 1 are omitted in the figure.

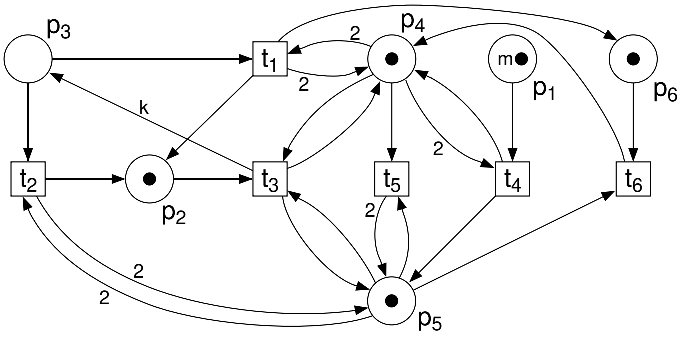


Fig. 9. Example Petri net.

Figure 10 shows the reduced IMD of the corresponding transition relation. Action intervals containing only one element are abbreviated by this element. Omitted predicate or action intervals stand for non-existing predicates or value changes, respectively.

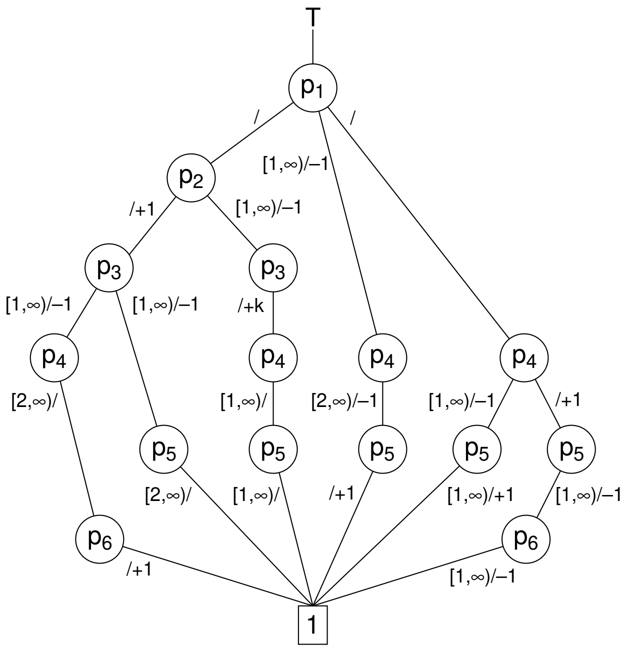


Fig. 10. Transition relation IMD.

Each path in the transition relation IMD describes one of the Petri net transitions. The paths in the example IMD correspond to the Petri net transitions t_i ordered from left to right. The mapping functions along a path describe the state distances and their corresponding firing conditions, i.e., the state variable changes that are the consequence of the execution of the respective transition if all its predicates are satisfied. For instance, if $p_3 \in [1, \infty)$ and $p_4 \in [2, \infty)$, transition t_1 is enabled to fire by decreasing p_3 and increasing p_2 and p_6 by 1 while p_1, p_4, and p_5 remain unchanged.

Since predicate interval and action interval are independent of each other, even self-loop transitions and inhibitor arcs may be modeled. While unbounded state spaces with state variable domains of $[0, \infty]$ are favorable for describing Petri nets with infinite place capacities, an explicit bound may exist, e.g., as an implementational or conceptual limit of the size of the modeled element. In such case, the introduction of bounded state spaces is useful to avoid the necessity of adding restricting places to the original Petri net. For instance, the state variable domain of an element of capacity k is reduced to $[0, k]$, implying the use of the respective modified predicate intervals inhibiting that the number of tokens in a place may exceed its bound. Another important aspect is that IDDs are not able to describe all kinds of Boolean functions in a finite manner, e.g., the relation $u = v$ would result in an IDD of infinite extension as each pair (u, v) of equal variable values has to be stored explicitly. By introduction of lower and upper bounds, the set of pairs becomes finite and so does the IDD.

B.2 Model Checking

Properties to be verified using symbolic model checking of Petri nets may be described as CTL formulae and verified with well-known techniques [26], [28]. A few questions out of the wide variety of system properties to be checked using CTL are, e.g., "may places p_1 and p_2 simultaneously contain no tokens?", "can transitions t_1 and t_2 be fired concurrently?", or "must all firing sequences eventually result in marking M_1?". Additionally, specialized algorithms exist for the verification of many common Petri net properties as described in [27] and [22] and are straightforwardly adaptable to IDDs, e.g., for

- deadlock freeness and diverse levels of liveness,
- boundedness, persistence, and home state property.

C. Experimental Results

C.1 System Models

Several diverse system models based on process networks with FIFO queues have been investigated which led to promising results. Starting with the initial state, the set of reachable states has been calculated by a series of image computations using the operators of Section V until a fixpoint has been reached. Some results for different initial configurations are presented in the following, comparing IDDs and IMDs to BDDs. In the BDD version, the coding of the state variable values was directly binary. Our investigations yielded promising results concerning the number of nodes and edges as well as the computation time.

In the left-hand part of Figure 11, the computation time T to determine the set of reachable states of a model of a flexible manufacturing system with automated guided vehicle [30] is depicted for increasing initial configurations n. The parameter n represents the initial number of tokens in a given queue and hence is a measure of the size or the complexity of the model. The right-hand part of Figure 11

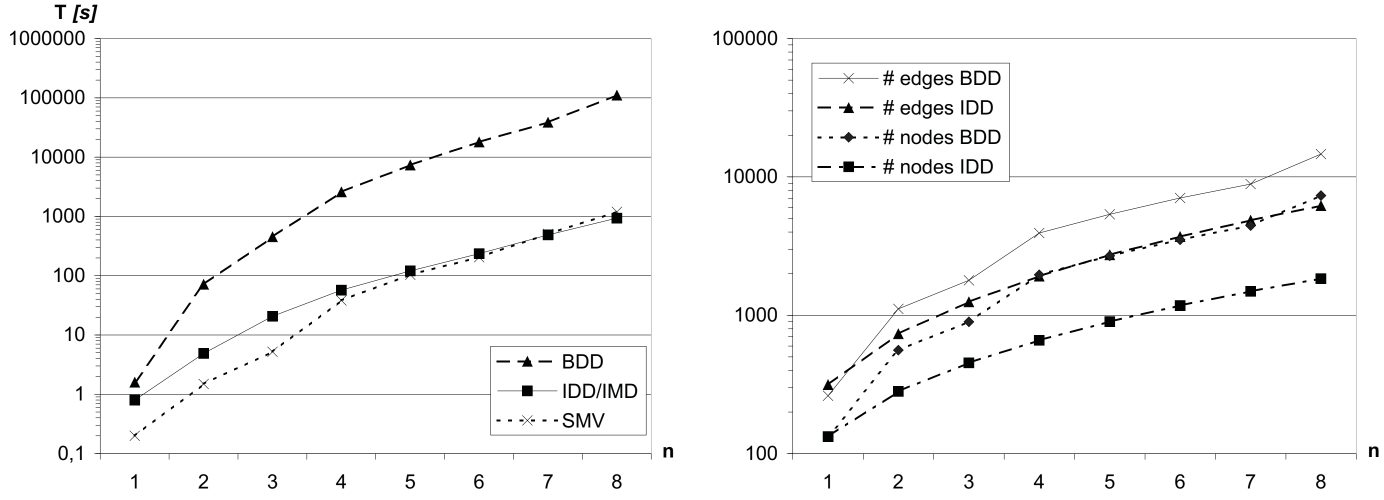


Fig. 11. Computation time and state set diagram size T for manufacturing system.

shows the size of the diagram representing the set of reachable states depending on the initial configuration n. For both criteria, IDDs and IMDs turn out to be significantly superior to the conventional approach using BDDs.

The left-hand part of Figure 11 also shows the computation time using the symbolic model checker Cadence SMV which is based on BDDs. The same variable ordering as for IDDs and the standard optimization parameters have been used.

Our implementation is not tuned for speed and does not contain common optimization techniques such as partitioned transition relations or sifting which are also applicable to interval diagrams. According to the experimental results, the required memory and the complexity of the operations (Boolean operations, image computation) for BDDs and for interval diagrams differ in a constant factor with regard to the model size. Therefore, both implementations would profit in the same way from increasing the efficiency of the implementation.

The BDD implementation of Cadence SMV is up to one or two decades faster than our BDD implementation, while IDDs/IMDs are roughly as fast as SMV for large models. SMV is a highly optimized state-of-the-art model checker. On the other hand, our implementation is experimental and uses the programming language Java. Thus, we expect substantial constant factor improvements based on the implementation.

Obviously, the memory required to store IDD nodes and edges is different compared to BDDs because IDD nodes can have more than only two edges and intervals are assigned to the edges. Figure 12 shows the memory required for the diagram representing the set of reachable states depending on the initial configuration n.

The transition relation diagram size for the manufacturing system is compared for IMDs and BDDs in Table V.

Figure 13 shows the computation time and the size of the state set diagram for the reachability analysis of a readers and writers system accessing a common database.

In Table VI, the size of the transition relation diagram is

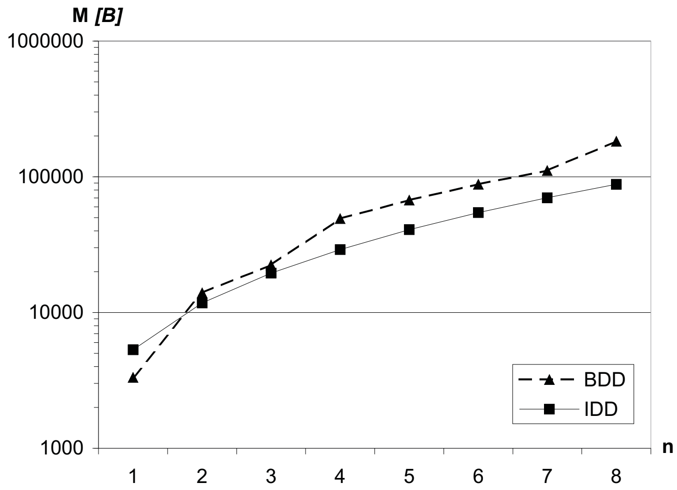


Fig. 12. Memory requirements of state set diagram for manufacturing system.

TABLE V
Size of transition relation diagram for manufacturing system.

	IMD	BDD $n = 7$	BDD $n = 8$
# layers	27	162	216
# nodes	67	1957	2691
# edges	89	3910	5378

compared for the readers and writers system. The significant reduction of the number of nodes in the transition relation IMD and in the state set IDD is obvious compared to the BDD equivalents. The IMD size is independent of the initial configuration, while the BDD size increases heavily.

In Figure 14, the computation time and the state set diagram size for a symmetric multiserver random polling system are shown.

Considering the computation time, BDDs additionally

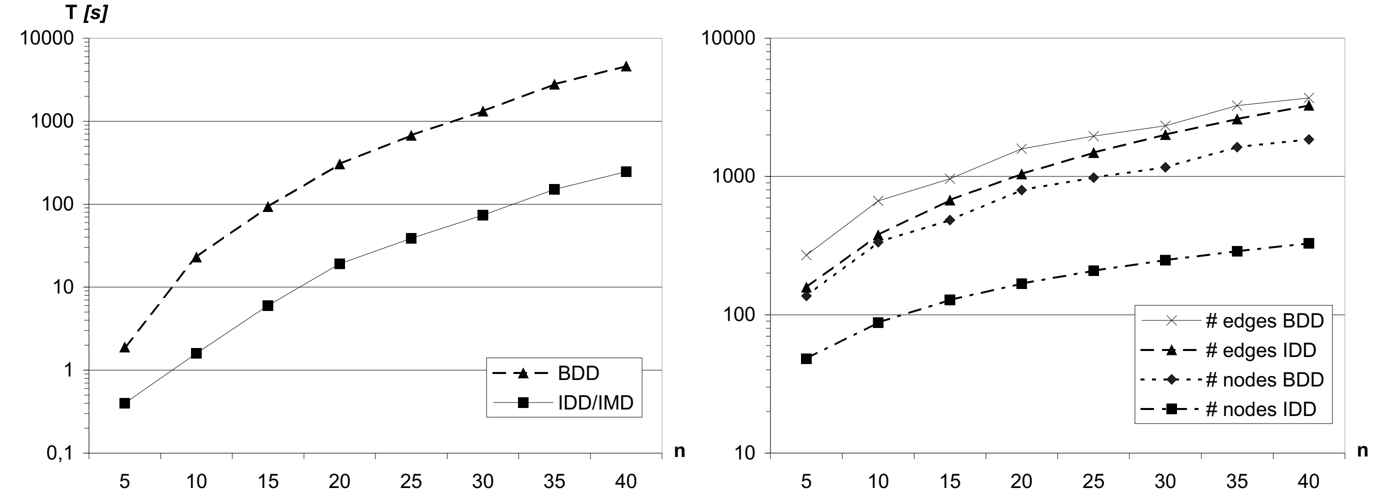


Fig. 13. Computation time T and state set diagram size for readers and writers system.

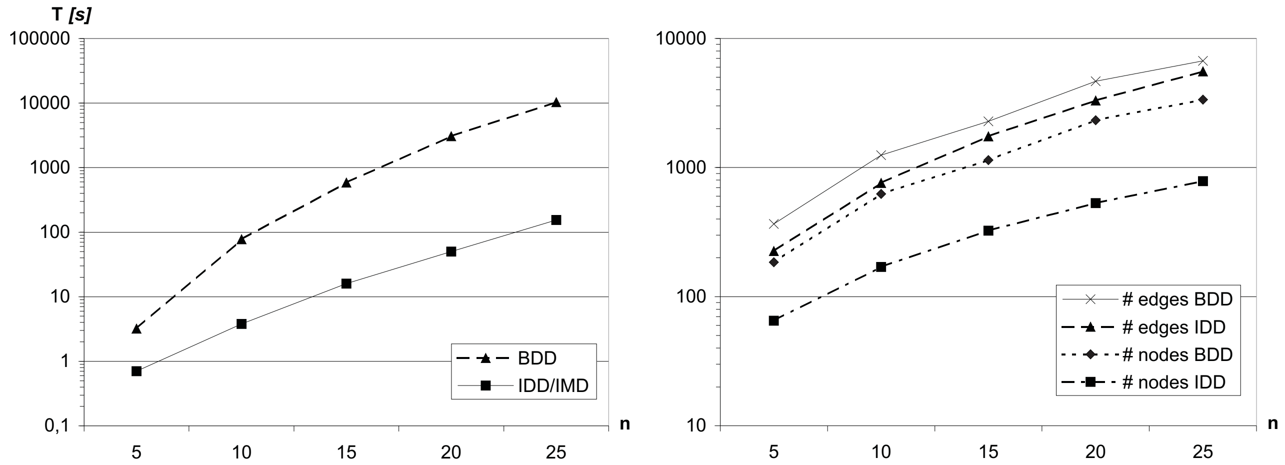


Fig. 14. Computation time T and state set diagram size for polling system.

TABLE VI

Size of transition relation diagram for readers and writers system.

	IMD	BDD	
		$n = 30$	$n = 35$
# layers	7	70	84
# nodes	15	571	696
# edges	20	1138	1388

have one major disadvantage in contrast to IDDs and IMDs. As mentioned in Section I, using BDDs requires an upper bound for the state variable values as they are binary coded. However, such a priori bounds are not known in general. Using loose upper bounds causes substantial computation overhead—Table VII and Table VIII show the dependence of the BDD size for the transition relation T and the set of reachable states S on the chosen coding length. In most cases, the alternative would be to increase the bounds incrementally—i.e., to add bits to the coding—until they are high enough but still tight. The time each iteration of this "trial-and-error" method takes cannot be neglected, while no time is wasted using IDDs and IMDs as the first run yields the final result.

TABLE VII

BDD size for polling system with $n = 15$.

variable domain	$[0, 15]$	$[0, 31]$	$[0, 63]$
# layers of T	48	60	72
# nodes in T	403	521	639
# edges in T	802	1038	1274
# layers of S	24	30	36
# nodes in S	1143	1486	1789
# edges in S	2282	2968	3574

C.2 Example Petri Net

Astonishingly, the Petri net in Figure 9 is inherently bounded in an unobvious but very regular way [29]. The Petri net is parameterized by the weight $k \geq 2$ of the arc between t_3 and p_3 and the initial marking n of place p_1.

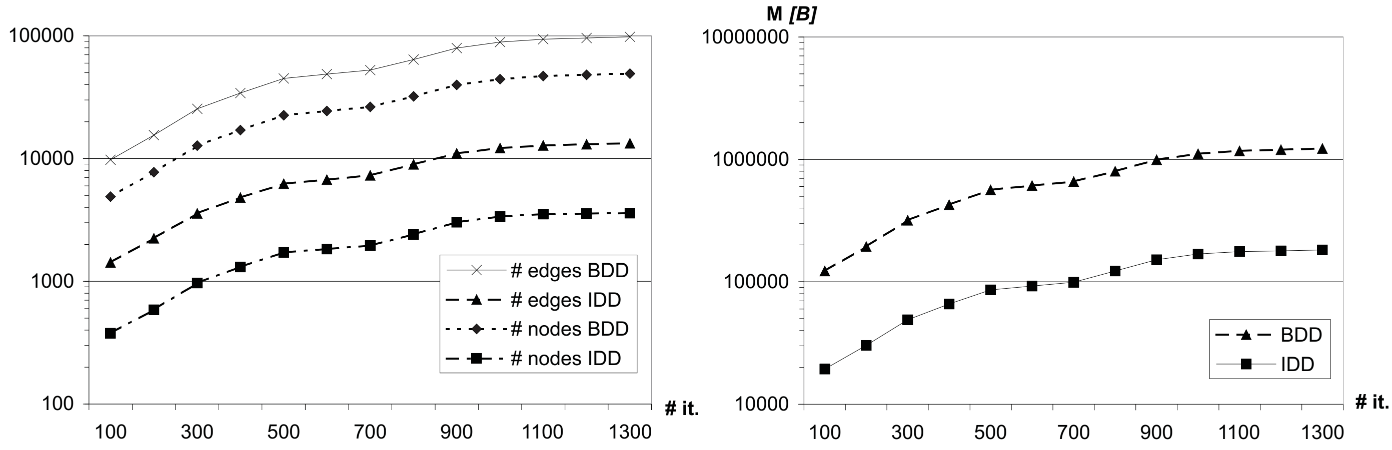


Fig. 15. Diagram size and memory requirements during fixpoint computation for Petri net.

Depending on the values of those two parameters, the total number of tokens in the net is bounded to $max(m, k) = k \cdot f_k(m) + 2$, where f_k is defined inductively by

$$f_k = \begin{cases} k & \text{if } m = 0 \\ f_k(m-1) \cdot k^{f_k(m-1)} & \text{otherwise} \end{cases} .$$

The maximum number of tokens in any single place is $k \cdot f_k(m)$. Some example values of $max(m, k)$ are:

$$\begin{array}{ll} max(1,2) = 18 & max(2,2) = 4098 \\ max(1,3) = 245 & max(2,3) = 3^{86} + 2 \\ max(1,4) = 4098 & max(3,2) = 2^{2060} + 2 \end{array} .$$

The set of reachable states has been calculated by a series of image computations as described in [27]. Some results for $m = 1$ and $k = 3$ are shown in Table VIII.

The maximum total number of tokens is 245, thus any single Petri net place cannot contain more than 243 tokens at the same time. Hence, the length of the coding in the first BDD column of Table VIII suffices to represent all possible state variable values. 494 fixpoint iterations are necessary to determine the set of reachable states. Especially the significant reduction of the number of nodes in the transition relation IMD and in the state set IDD is obvious compared to the BDD equivalents.

For $m = k = 2$, the final result was unable to be determined due to memory limitations. Figure 15 shows the diagram size during fixpoint computation, requiring a variable domain of $[0, 8191]$ and 156 layers for the transition relation BDD. Again, IDDs require greatly less nodes and edges than BDDs.

VII. Concluding Remarks

A. Further Applications

Besides the formal verification approach considered in this paper, interval diagram techniques have several fields of application in the areas of real-time computing, embedded systems, and fuzzy computing, of which some have been summarized in [31], [7].

A.1 Timed Automata

Timed automata [32] are used to model timing behavior of state transition systems. They are derived from conventional finite state automata and are represented by means of state-transition graphs with timing constraints using a finite number of clocks. Discrete-valued clocks may be regarded as particular integer state variables of which the values increase simultaneously when time progresses. This *time forward projection* may be performed by repeated and simultaneous incrementation of all clock values about a *time distance* of 1, depending on the actual system state and thus similarly to state distances.

Using interval diagram techniques, formal verification of timed automata has been performed by reachability analysis on discrete time [19]. Our approach has been compared to another, similar verification technique based on BDDs. For large models, the IDD/IMD approach significantly outperformed the BDD approach with regard to computation time—by a factor of more than 7 and likely even more. Noteworthy is that the weaker slope of the IDD/IMD computation time seems to be an indication that the algorithmic complexity for this kind of application is lower. Furthermore, possible improvements of a factor of up to 5 or 10 are expected because an experimental implementation was compared with an efficient state-of-the-art tool. Besides timed automata, interval diagram techniques may be applied to the analysis of discrete hybrid automata on integer variables.

A.2 Symbolic Scheduling

Besides formal verification, interval diagram techniques have been applied to symbolic scheduling of mixed hardware/software systems in the embedded systems domain. Due to similar transition behaviors, the above advantages of symbolic approaches based on interval diagram techniques may be transferred to the area of symbolic scheduling of, e.g., the FunState model of computation [33], [6]—besides its formal verification by symbolic model checking. A symbolic scheduling method for heterogeneous embedded systems represented by means of FunState models has

TABLE VIII
BDD size for Petri net with $m = 1$ and $k = 3$.

	IDD/IMD	BDD			
variable domain	$[0, \infty)$	$[0, 255]$	$[0, 511]$	$[0, 1023]$	$[0, 2047]$
# layers of T	6	96	108	120	132
# nodes in T	16	700	795	890	985
# edges in T	20	1396	1586	1776	1966
# nodes in S	3796	26791	30585	34379	38173
# edges in S	41070	53578	61166	68754	76342
# layers of S	6	48	54	60	66

been developed [5]. Symbolic scheduling methods often outperform both ILP and heuristic methods while yielding exact results.

The new scheduling method is able to deal with mixed data/control flow specifications and takes into account different mechanisms of non-determinism as occurring in the design of embedded systems. Constraints imposed by other already implemented components are respected. The scheduling approach avoids the explicit enumeration of execution paths by using symbolic techniques based on interval diagrams and guarantees to find a deadlock-free and bounded schedule if one exists. The generated schedule consists of statically scheduled basic blocks which are dynamically called at run time.

A.3 Fuzzy Decision Diagrams

Based on interval decision diagrams, *fuzzy decision diagrams* (FuDDs) [34] have been developed. FuDDs can be used for representing, analyzing, and optimizing fuzzy rule bases. The variables involved are linguistic ones, and their values are fuzzy sets. FuDDs are comparable with multi-terminal IDDs of which the edges are annotated with fuzzy sets instead of crisp interval sets. Hence, in contrast to IDDs several or all FuDD paths can be "active" at the same time but each of them more or less, according to the fuzzy philosophy. Using FuDDs, fuzzy if-then rule bases can be represented graphically.

Similarly to IDDs, fuzzy decision diagrams provide an efficient data structure for adequate representation, analysis, and optimization of rule bases. Moreover, FuDDs enable checking for completeness and consistency of a rule base. Ordered as well as free FuDDs are of importance for rule base optimization. Moreover, hierarchical rule bases can be represented which implies the chaining of rules. For restricted classes of fuzzy rule bases, efficient algorithms for optimization and evaluation or rule bases exist.

B. Related Work

An approach with aims similar to our's is the one of Godefroid and Long [35], [36]. They verify system models—especially for lossy communication protocols—based on FIFO queues by binary coding the queue contents and representing them in form of *queue BDDs* (QBDDs). QBDDs are an extension of BDDs for dealing with dynamic data structures and infinite state spaces. They must renounce an ordered BDD form as repeated occurrences of variables along a path are necessary. The DD depth is not static, but may increase substantially during the computations depending on the number of elements contained in the queues. Only the enqueuing or dequeuing of single elements is treated. QBDDs are used to describe sets of system states, but not the transition relation. The methods applied for this require specialized enqueuing and dequeuing methods of which the possibility to be combined with conventional BDD techniques is not guaranteed.

Related work concerning symbolic model checking of Petri nets has been performed by Pastor, Roig, Cortadella et al. [27], [22]. Their focus is on safe, i.e., 1-bounded, Petri nets used to verify speed-independent circuits modeled by means of signal transition graphs. For this Petri net class, the binary BDD representation is adequate and allows direct conversions as no binary coding is necessary. Although a hint of possible extensions for k-bounded Petri nets is included in [27], this means no fundamental advance due to the reasons stated above. In [37], zero-suppressed BDDs are used instead of regular BDDs to speed up the computation. General deficiencies of BDDs have been partially mitigated by applying novel encoding schemes to represent place contents [38], [39]. This is achieved by partially or completely transforming the Petri net into a set of finite state automata. [40] presents an MDD-based approach which exploits locality of transition firings to reduce both the number of iterations and the cost of each iteration during reachability analysis.

An approach to the symbolic verification of timed automata inspired by interval diagram techniques is introduced in [41]. It makes use of *clock difference diagrams* (CDDs) which essentially are based on a modification of IDDs. In addition to clock values, CDDs store differences between clock values, but unfortunately by abandoning a directly canonical representation and hence its advantages with regard to fixpoint comparisons. Methods for CDDs such as union, intersection, and complement are straightforwardly adopted from IDDs. For equality tests during fixpoint computations, CDDs first have to be transformed into a quasi-canonical form. This seems to limit run-time reductions significantly which is confirmed by the experimental results presented. Compared with the original Uppaal tool, considerable space savings using a hybrid approach are accompanied even by slight run-time increases.

However, in contrast to the interval diagram techniques as described in [19], CDDs can be used directly in a dense time setting.

C. Summary and Conclusion

Symbolic model checking attempts to avoid the state explosion problem by implicit construction of the state space. The major limiting factor is the size of the symbolic representation mostly described by means of huge BDDs. Especially for process networks, traditional approaches have shown not to be feasible due to the above-mentioned shortcomings. A new approach to the symbolic formal verification of process networks and related models of computation has been presented. It is based on a novel, efficient form of representation of multi-valued functions called interval decision diagram (IDD) and the corresponding image computation technique using interval mapping diagrams (IMDs).

A specialized IMD variant, the predicate action diagram (PAD), is well suited to represent the transition relation of the considered kind of models, describing the transitions by combinations of predicates for the firing conditions and state distances for the production and consumption rates. Several drawbacks of conventional symbolic model checking of process networks with BDDs are avoided due to the use of interval diagram techniques. Especially the resulting transition relation IMD is very compact, enabling fast image computations. Furthermore, no artificial limitations concerning buffer capacities need to be introduced. Our experimental results significantly outperform those of the BDD approach with regard to computation time and memory.

Acknowledgements

We wish to thank the anonymous reviewers for their constructive comments and helpful suggestions.

References

[1] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic model checking for sequential circuit verification.," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 4, 1994.

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang, "Symbolic model checking: 10^{20} states and beyond.," *Information and Computation*, vol. 98, no. 2, pp. 142–170, June 1992.

[3] S. B. Akers, "Binary decision diagrams," *IEEE Transactions on Computers*, vol. C-27, no. 6, pp. 509–516, June 1978.

[4] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[5] K. Strehl, L. Thiele, D. Ziegenbein, R. Ernst, and J. Teich, "Scheduling hardware/software systems using symbolic techniques," in *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES'99)*, Rome, Italy, May 3–5, 1999, pp. 173–177.

[6] L. Thiele, K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich, "*FunState*—an internal design representation for codesign," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-99)*, San Jose, California, November 7–11, 1999.

[7] Karsten Strehl, *Symbolic Methods Applied to Formal Verification and Synthesis in Embedded Systems Design*, Ph.D. thesis, Swiss Federal Institute of Technology (ETH) Zurich, Shaker Verlag, Aachen, Germany, ISBN 3-8265-7193-2, March 2000.

[8] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–799, 1995.

[9] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: Determinacy, termination, and queueing," *SIAM Journal on Applied Mathematics*, vol. 14, no. 6, pp. 1390–1411, 1966.

[10] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24–35, 1987.

[11] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[12] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress Information Processing*, 1974.

[13] G. Kahn and D. B. MacQueen, "Coroutines and networks of parallel processes," in *Proceedings of the IFIP Congress Information Processing*, 1977.

[14] L. Thiele, J. Teich, and K. Strehl, "Regular state machines," *Journal of Parallel Algorithms and Applications*, vol. Special Issue on Advanced Regular Array Design *(to appear)*, 2000.

[15] T. M. Parks, *Bounded Scheduling of Process Networks*, Ph.D. thesis, University of California, Berkeley, 1995.

[16] K. Strehl and L. Thiele, "Symbolic model checking of process networks using interval diagram techniques," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-98)*, San Jose, California, November 8–12, 1998, pp. 686–692.

[17] S. Haynal and F. Brewer, "Efficient encoding for exact symbolic automata-based scheduling," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-98)*, 1998.

[18] A. Srinivasan, T. Kam, S. Malik, and R. K. Brayton, "Algorithms for discrete function manipulation," in *Proceedings of the IEEE International Conference on Computer Aided Design*, 1990.

[19] K. Strehl, "Interval diagrams: Increasing efficiency of symbolic real-time verification," in *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA '99)*, Hong Kong, December 13–15, 1999, pp. 488–491.

[20] K. Strehl and L. Thiele, "Symbolic model checking using interval diagram techniques," Technical Report TIK-40, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, Feb. 1998.

[21] E. M. Clarke, M. Fujita, and X. Zhao, "Multi-terminal binary decision diagrams and hybrid decision diagrams," in *Representations of Discrete Functions*, T. Sasao and M. Fujita, Eds., pp. 93–108. Kluwer Academic Publishers, 1996.

[22] O. Roig, J. Cortadella, and E. Pastor, "Verification of asynchronous circuits by BDD-based model checking of Petri nets," in *16th International Conference on Application and Theory of Petri Nets*, vol. 935 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[23] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package.," in *Proceedings of the 27th IEEE/ACM Design Automation Conference*, 1990.

[24] A. J. Hu and D. L. Dill, "Efficient verification with BDDs using implicitly conjoined invariants," in *Computer Aided Verification*, vol. 697 of *Lecture Notes in Computer Science*, pp. 3–14. Springer-Verlag, 1993.

[25] S. Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems," in *Proceedings of the 30th IEEE/ACM Design Automation Conference*, 1993.

[26] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[27] E. Pastor, O. Roig, J. Cortadella, and R. M. Badia, "Petri net analysis using boolean manipulation," in *15th International Conference on Application and Theory of Petri Nets*, vol. 815 of *Lecture Notes in Computer Science*, pp. 416–435. Springer-Verlag, 1994.

[28] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*, MIT Press, 2000.

[29] M. Jantzen, "The large markings problem," *Petri Net Newsletter*, vol. 14, pp. 24–25, 1983.

[30] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, *Modelling with Generalized Stochastic Petri Nets*, John Wiley & Sons, 1995.

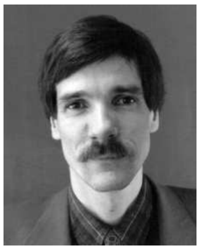
[31] K. Strehl and L. Thiele, "Interval diagram techniques and their

applications," in *Proceedings of the 8th International Workshop on Post-Binary ULSI Systems*, Freiburg im Breisgau, Germany, May 19, 1999, pp. 23–24, invited paper.
[32] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.
[33] L. Thiele, J. Teich, M. Naedele, K. Strehl, and D. Ziegenbein, "*FunState*—functions driven by state machines," Technical Report TIK-33, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, Jan. 1998.
[34] K. Strehl, C. Moraga, K.-H. Temme, and R. Stanković, "Fuzzy decision diagrams for the representation, analysis, and optimization of rule bases," in *Proceedings of the 30th International Symposium on Multiple-Valued Logic (ISMVL 2000) (to appear)*, Portland, Oregon, May 23–25, 2000.
[35] P. Godefroid and D. E. Long, "Symbolic protocol verification with queue BDDs," in *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, 1996.
[36] P. Godefroid and D. E. Long, "Symbolic protocol verification with queue BDDs," *Formal Methods in System Design*, vol. 14, pp. 257–271, 1999.
[37] T. Yoneda, H. Hatori, A. Takahara, and S. Minato, "BDDs vs. zero-suppressed BDDs: for CTL symbolic model checking of Petri nets," in *Formal Methods in Computer-Aided Design*, vol. 1166 of *Lecture Notes in Computer Science*, pp. 435–449. Springer-Verlag, 1996.
[38] E. Pastor and Cortadella, "Efficient encoding schemes for symbolic analysis of Petri nets," in *Proceedings of the Design, Automation and Test in Europe Conference (DATE98)*, 1998.
[39] E. Pastor, J. Cortadella, and M. A. Peña, "Structural methods to improve the symbolic analysis of Petri nets," in *Proceedings of the 20th International Conference on Application and Theory of Petri Nets*, 1999.
[40] A. S. Miner and G. Ciardo, "Efficient reachability set generation and storage using decision diagrams," in *Proceedings of the 20th International Conference on Application and Theory of Petri Nets*, 1999.
[41] G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi, "Efficient timed reachability analysis using clock difference diagrams," in *Proceedings of the International Conference on Computer-Aided Verification*, 1999.

Karsten Strehl's research interests include design automation methods and tools for specification, analysis, and synthesis of embedded hardware/software systems, in particular high-level system design approaches and formal methods.

In 1997, after receiving a diploma degree in electrical engineering from the University of Karlsruhe, Germany, he joined Lothar Thiele's research group at the Computer Engineering and Networks Lab (TIK) at the Swiss Federal Institute of Technology (ETH) Zurich. He received the Doctor of Technical Sciences degree from ETH Zurich in 2000.

Karsten Strehl received the 1997 Award of the Faculty of Electrical Engineering at Karlsruhe University and the 1997 Siemens Information and Communication Award. He is a member of the IEEE.



Lothar Thiele received the Dipl.-Ing. and Dr.-Ing. degrees in electrical engineering from the Technical University of Munich, Germany, in 1981 and 1985, respectively. After 1981, he has been a research associate with Rudolf Saal at the Institute of Network Theory and Circuit Design of the Technical University of Munich. After finishing his habilitation thesis, he joined the group of Thomas Kailath at the Information Systems Laboratory, Stanford University, in 1987.

In 1988, he has taken up the Chair of Microelectronics at the Faculty of Engineering, Saarland University, Saarbrücken, Germany. He joined ETH Zurich, Switzerland, as a full professor in computer engineering at the end of 1994.

His research interests include models, methods, and software tools for the design of embedded systems.

In 1986, he received the Award of the Technical University of Munich for his Ph.D. thesis. He received the 1987 Outstanding Young Author Award of the IEEE Circuits and Systems Society. In 1988, he was the recipient of the 1988 Browder J. Thompson Memorial Prize of the IEEE.