In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-98)*, San Jose, California, pages 686–692, November 8–12, 1998.

Symbolic Model Checking of Process Networks Using Interval Diagram Techniques

Karsten Strehl, Lothar Thiele

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

Abstract

In this paper, an approach to symbolic model checking of process networks is introduced. It is based on *interval decision diagrams* (IDDs), a representation of multi-valued functions. Compared to other model checking strategies, IDDs show some important properties that enable the verification of process networks more adequately than with conventional approaches. Additionally, applications concerning scheduling will be shown. A new form of transition relation representation called *interval mapping diagrams* (IMDs)—and their less general version *predicate action diagrams* (PADs)—is explained together with the corresponding methods.

1 Introduction

Process network models—consisting in general of concurrent processes communicating through unidirectional FIFO queues—as that of Kahn [7, 8] are commonly used, e.g., for specification and synthesis of distributed systems. They form the basis for applications such as real-time scheduling and allocation. Many other models of computation, e.g., *dataflow process networks* [11], *computation graphs* [9], and *synchronous dataflow* (SDF) [10], turn out to be special cases of Kahn's process networks.

As the SDF model is restricted enough, tasks as determining a static schedule 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 balance equation methods, they fail for more powerful descriptions due to their complex internal state.

Typical questions to be answered by formal verification of process networks are about the absence of deadlocks, the boundedness of the required memory, or rather "may process P_1 and P_2 block each other?" or "may P_1 and P_2 collide while accessing to a shared resource?". Especially the memory boundedness is important as process networks in general may not be scheduled statically. Thus, dynamic schedulers have to be deployed which cannot always guarantee to comply with memory limitations without restricting the system model [13].

In addition, the process models may be extended to describe one or several dynamic or hybrid scheduling policies, too, of which the behavior is verified together with the system model. Thus, common properties as the correctness of the schedule may be affirmed or artificial deadlocks [13] may be detected.

A simple example process network from [13] is shown in Figure 1. A, B, C, and D represent processes, while a, b, c, d, and e are unbounded FIFO queues. The network follows a blocking read semantics, i.e., a process is suspended when attempting to get data from an empty input channel.

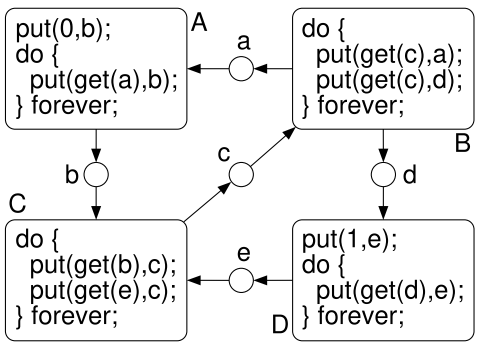

Figure 1: Simple process network.

Formal verification is able to prove, e.g., that there are never more than two tokens buffered on any communication channel—i.e., that the process network is *strictly bounded*—or that the network is non-terminating. Both properties are essential during 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 now is whether B could be blocked forever because A may always be enabled.

During the last years, a promising approach named *symbolic model checking*, see, e.g., [3, 4], was applied to many areas of system verification and even has been able to enter the area of industrial applications. It makes use of *binary decision diagrams* (BDDs) [2] which are an efficient representation of Boolean functions and allow for this very fast manipulation.

Concerning process networks, the traditional BDD-based method of automated verification suffers from the drawback that a binary representation of the process network and its state is required. One severe problem is that the necessary capacity of the queues in general is unknown before the analysis process. But to perform model checking, the queue contents represented by an integer number have to be coded binary, requiring an upper bound. This deficiency could be avoided partially using *multi-valued decision diagrams* (MDDs) [14] with unbounded variable domains instead of BDDs. But problems occur, e.g., when complementing a finite set described by an unbounded MDD as this results in an infinite set taking an infinite number of MDD edges which is not possible. One strategy to avoid this again is to bound the variable domain to a finite range such that complementary sets are finite, too.

Another difficulty emerges from the partially very regular behavior of network processes that in general consists of consuming or producing one or a few constant numbers of tokens at a time. Consider a simple dataflow node. Its firing behavior with respect to an outgoing arc representing a queue 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 queue's state and its successor state after the firing, e.g., $\{(x, x')\} = \{(0, 1), (1, 2), (2, 3), \ldots, (n-1, n)\}$.

An approach with aims similar to our's is the one of Godefroid and Long [5]. They verify system models—especially for lossy communication protocols—based on FIFO queues by coding the queue contents binary 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 have to 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.

To overcome the above-mentioned limitations of conventional symbolic model checking of process networks, we present an approach that uses *interval decision diagrams* (IDDs) combined with *interval mapping diagrams* (IMDs)—especially their restricted form *predicate action diagrams* (PADs)—and thus is able to remedy the described lacks of traditional approaches. Fundamentally, it is based on a more reasonable way of describing the above-mentioned form of transition relations. It affords the possibility to represent them as the "distance" between a state and its successor after the transition, which means the difference of the numbers of included tokens before and after the execution. 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 compact—especially for models like process networks—as only *state distances* are stored instead of combinations of state and successor. Accordingly, an innovative technique for efficient 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 to apply this new form of function description to symbolic model checking. In this paper, process networks are regarded exemplarily without meaning further restrictions.

2 Interval Decision Diagrams

2.1 Notation

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 $P_i \subseteq \mathbb{Z}$ 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 a set $I \subseteq \mathbb{Z}$, 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 $I = \{b\}$ containing only one single value $b \in \mathbb{Z}$, the literal of variable x with respect to I is denoted $x^b = x^I$.

In the following, we will deal with intervals on $\mathbb{Z}$, i.e., integer intervals. Two intervals are named *neighbored* if they may be joined by union into a larger interval, where overlapping intervals are called neighbored, too.

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

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

In this case, I is called an *independence interval* of f with respect to x_i. From now on, all domain sets P_i are supposed to be intervals.

Definition 2.1 (Interval cover) *The set $I(P_i) = \{I_1, I_2, \ldots, I_{p_i}\}$ of p_i split intervals I_j depicts 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 2.2 (Interval partition) *A cover is* disjoint *if*

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

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

Definition 2.3 (Independence interval partition) *An* independence interval partition *is a partition consisting of independence intervals only.*

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

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

The operations + and $\cdot$ in this equation have a "Boolean-like" meaning, hence shadowing all but one function value of f that corresponds to the respective value of x_i.

Definition 2.4 (Reduced interval partition) *An independence interval partition is named* minimal *if it contains no neighbored 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 2.1 *A reduced independence interval partition of a domain P_i is* unique.

The proof is by contradiction.

2.2 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

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

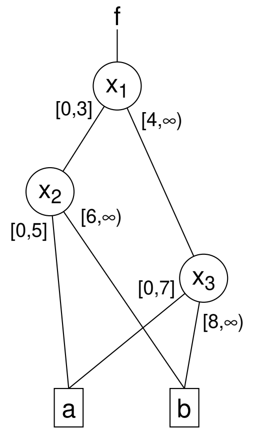

Figure 2: Example interval decision diagram.

The IDD edges are labeled with real or integer intervals. In the scope of this paper, we concentrate on intervals of integer numbers. Extensions for real numbers can easily be derived [15]. IDDs can be regarded as a generalization of BDDs, MDDs, and MTBDDs (*multi-terminal BDDs*).

2.3 Representation

IDDs are represented by canonical *function graphs*, similar to those of [14] and [2].

Definition 2.5 (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 $i = index(v)$, an independence interval partition $part(v) = I(P_i) = \{I_1, I_2, \ldots, I_{p_i}\}$ and $p_i = |part(v)|$ children $child_k(v) \in V, 1 \le k \le p_i$. The split intervals $int_k(v) = I_k \in I(P_i)$ of the partition are assigned to the* corresponding *graph edges $\big(v, child_k(v)\big) \in E$. A* terminal *node v has as attribute a value $value(v) \in Q_f$.*

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

Definition 2.6 *The function f_v associated to a node $v \in V$ of a function graph G is defined recursively as:*

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

$$f_v = \sum_{I_j \in part(v)} x_i^{I_j} \cdot f_{child_j(v)}. \qquad (2)$$

The function denoted by the graph G is associated to 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 to 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 $I_k = int_k(v)$ of the interval partition $part(v)$ and the function F_k associated to the respective corresponding child node $child_k(v)$. This description is directly associated to the Boole-Shannon expansion mentioned above.

Definition 2.7 (Ordered function graph) *A function graph is* ordered *if for any adjacent pair of non-terminal nodes $\big(v, child_k(v)\big)$ we have $index(v) < index\big(child_k(v)\big)$.*

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

Definition 2.8 (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 [2]), and*
3. *the independence interval partitions $part(v)$ of all non-terminal nodes v are reduced.*

Now, one of the major results of this paper is described.

Theorem 2.2 *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 [2] and [14]—by induction on the size of the dependence set—, but is omitted here for reasons of space [15]. ■

Similar to BDDs and related decision diagrams, several reduction rules exist to transform any IDD into a reduced form [15]. For conventional symbolic model checking of the described class of models of computation, basically Boolean functions over integer variables are of importance. Hence, from now on, only IDDs are considered that represent Boolean functions over integer intervals, i.e., 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 as, e.g., $f(x_1, x_2) = (x_1 \le 7) \wedge (x_2 = 3) \vee (x_2 \ge 6) = x_1^{[0,7]} \cdot x_2^{[3,3]} \vee x_2^{[6,\infty)}$ are describable.

2.4 If-Then-Else Operator

The *If-Then-Else* operator (ITE) [2] constructs the graph for the function obtained by composing two functions. 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. 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. Using the Boole-Shannon decomposition ITE is recursively defined as

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

where the terminal cases of this recursion are $ITE(1, F, G) = ITE(0, G, F) = ITE(F, 1, 0) = F$. The procedure to compute ITE is described in detail in [15].

3 Interval Mapping Diagrams

Interval mapping diagrams are used to represent transition relations, e.g., in symbolic model checking. They map a set described by a Boolean-valued IDD onto a new set—described by such IDD, too—by performing operations like, e.g., shifting or assigning some or all values of the IDD's decision variables. For interval shifting, a simple unbounded interval arithmetic with the operators $+$ for *addition* and $-$ for *substraction*, each over two intervals, is used.

3.1 Representation

IMDs are represented by *mapping graphs*, similar to the function graphs described in Definition 2.5. They contain *interval mapping functions* $f : \mathbb{I} \rightarrow \mathbb{I}$, mapping intervals onto intervals, where $\mathbb{I}$ denotes the set of all integer intervals.

Definition 3.1 (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 $value(v) = 1$.*

Unlike IDDs, IMDs in general have no canonical forms. As, for instance, no equivalence checkings or ITE operations of IMDs have to be performed in contrast to IDDs, this is no general limitation.

3.2 Interpretation

Informally, the functional behavior of IMDs may be described as "set flow", similar to that of data flow. The data consists of sets represented by unions of intersected intervals of state variable values, as described in Section 2.1 and represented by an IDD. Beginning at the root node of an IMD, the set data has the possibility to 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 degenerated 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 sketched in Section 4.

3.3 Predicate Action Diagrams

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

Definition 3.2 (Predicate action diagram) *A predicate action diagram is an interval mapping diagram containing only the following mapping functions:*

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

and

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

where $f+$ is the shift function *and $f_=$ the* assign function*. I_P is the* predicate interval *and I_A the* action interval*.*

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 reverse direction corresponding to interval substraction is achieved by addition of $-I = [-b, -a] = I_A$ and is denoted as $I_P/-I$.

4 Image Computation

4.1 Definition

Let $x = (x_1, x_2, \ldots, x_n)$ be a vector depicting a system state. Then a state set S is represented by its characteristic function $s(x)$ with

$$s(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases}. \quad (3)$$

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

$$\delta(x, x') = \begin{cases} 1 & \text{if } (x, x') \in T \\ 0 & \text{otherwise} \end{cases}. \quad (4)$$

In symbolic model checking, *image computation* is of importance. The image $Im(S, T)$ of a set S of system states with respect to 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 inverse image $PreIm(S, T)$ depicts all states that in one transition *can* reach a state in S. The formal definitions of the image operators are given, e.g., in [6].

4.2 Using IDDs and IMDs

Figure 3 shows an example process network with unspecific consumption and production rates represented by intervals. It is similar to a computation graph [9] where the consumption rate is independent of the threshold—depicted as a condition here.

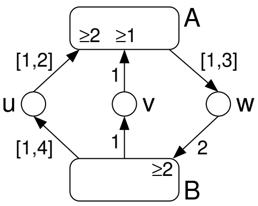

Figure 3: Example process network.

Figure 4 b) shows the corresponding transition relation represented by the PAD T, Figure 4 a) an example state set IDD S. The latter does not include the 0-terminal node and all its incoming edges for clearness.

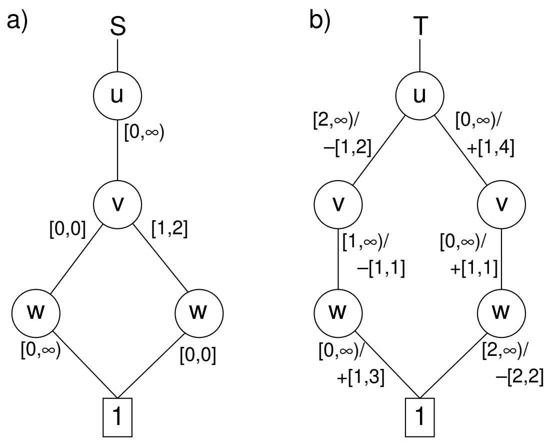

Figure 4: State set IDD and transition relation PAD.

The state set is described as $s(u, v, w) = (v = 0) \vee (1 \leq v \leq 2) \wedge (w = 0)$, the transition relation as $\delta(u, v, w, u', v', w') = (u \geq 2) \wedge (u - u' \in [1, 2]) \wedge (v \geq 1) \wedge (v' = v - 1) \wedge (w' - w \in [1, 3]) \vee (u' - u \in [1, 4]) \wedge (v' = v + 1) \wedge (w \geq 2) \wedge (w' = w - 2)$, for $u, v, w, u', v', w' \in [0, \infty)$.

4.3 Computation

In this section, we describe how to perform image computation using IDDs and IMDs. Conventionally, as mentioned in Section 1, the transition relation is represented as a BDD that explicitly stores all valid combinations of system state and predecessor state. Image calculations are performed using Boolean operators as existential and universal quantifier $\exists$ and $\forall$, respectively—which internally are computed using the If-Then-Else operator ITE, see [12] for instance. This strategy is possible for IDDs, too.

Another technique is described in the following. First, we introduce a general form of transition relations based on IMDs. Then we concentrate on further restrictions using PADs which allow for the efficient representation of state distances combined with the corresponding transition conditions.

Image computation with IDDs and IMDs 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 $mapForward(v, w)$ as sketched partially in Table 1—over an IDD node v and an IMD node w. The resulting IDD S' is constructed recursively by traversing depth-first both source graphs and creating new edges and nodes resulting from the interval mapping application by maintaining the respective graph structures. The operation is similar to the $Apply$ operation described in [2]. TN_0 and TN_1 are the 0- and 1-terminal node, respectively.

```
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)
    [...]
  else if index(v) > index(w)
    [...]
  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 new node ṽ with index(v);
              add new edge with interval I_res
                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;
  return v_res;
```

Table 1: Forward mapping for image computation.

As a memory function, a hash table of already computed results for pairs of v and w is maintained—omitted in Table 1—such that an included result may be returned without further computation.

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 CTL model checking using essentially $PreIm(S, T)$, the reverse direction is necessary, thus some kind of inverse mapping functions has to be used. For IMDs, depending on the structure of the mapping functions, this inversion is not always possible as an interval representation is necessary to display the function result. However, especially PADs have valid inversions [15].

4.4 Image Computation With PADs

Image computation with PADs now will be described briefly. The state distance between two system states x and x' is defined as $\Delta x = x' - x$. Thus, according to equation (4), the transition relation T may be described as the characteristic function

$$\hat{\delta}(x, \Delta x) = \begin{cases} 1 & \text{if } (x, x + \Delta x) \in T \\ 0 & \text{otherwise} \end{cases} . \quad (5)$$

In Figure 4 b), only shift functions f_+ are used as mapping functions. In the case of queue contents limited to non-negative numbers, the predicate intervals must ensure that the resulting state variables x'_i after a transition may not become negative, i.e., the 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 assign functions $f_=$ are dedicated to model finite state systems.

5 Symbolic Model Checking

Symbolic model checking allows for the verification of certain temporal properties of state transition systems, where the explicit construction of an entire reachability graph is avoided by implicitly depicting it using symbolic representations. Often, the propositional branching-time temporal logic CTL (*Computation Tree Logic*) is used to specify the desired system properties [12]. To verify such CTL formulae, ITE operator and image computation are used.

The most frequently employed form of symbolic representation are BDDs and their derivatives, e.g., see [4, 12]. The interval representations as introduced in this paper provide the following advantages.

- 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 dedicated 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 depicted as one IDD or IMD node.

5.1 Modeling Process Networks

For symbolic model checking, only the quantitative system behavior is considered, i.e., only the number of tokens in each queue, not their values. The behavior of Kahn process networks may be reproduced by decomposing the transition behavior of each process into atomar transitions, changing the internal state of the process and consuming and producing tokens in dependence on the internal state. For this decomposition, the process behavior has to be describable by a finite state machine. Recursive network structures are not allowed. Non-blocking read or blocking write semantics may be represented, too. Even non-determinate models with multi-reader and multi-writer queues as, e.g., Petri nets are verifiable using IDDs and IMDs.

Each path in the transition relation PAD describes one possible state transition. The mapping functions along the path depict enabling conditions and the corresponding state variable changes. The transition is enabled if all conditions along the path are satisfied. Analog 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 queue 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 FIFO queues—or of finite domain—describing internal process states or bounded queues.

5.2 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 [12].

Apart from this, 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 content of queues. For termination and deadlocks, respective CTL formulae may easily be formed. 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.

5.3 Experimental Results

Among several diverse system models based on FIFO queues producing promising results, the model of a symmetric multiserver random polling system [1] has been investigated. The set of reachable states has been calculated by a series of image computations. Some results for different initial configurations m are presented now, comparing IDDs and PADs to BDDs. In the BDD version, the coding of the state variable values was direct binary. Our investigations yielded promising results concerning the number of nodes and edges as well as the computation time. Figure 5 shows the size of the diagram representing the set of reachable states for increasing initial configurations.

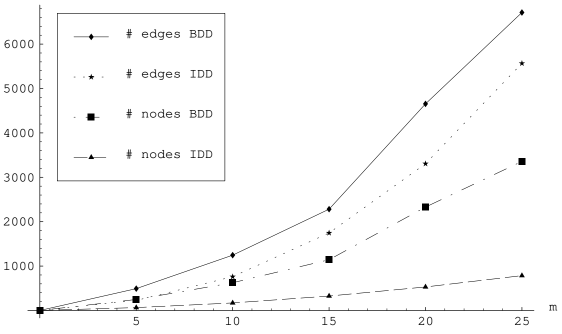

Figure 5: Size of state set diagram.

In Figure 6, the computation time to determine the set of reachable states is depicted depending on the initial configuration.

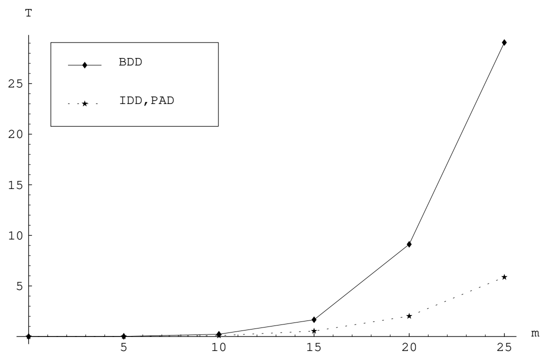

Figure 6: Computation time of reachability analysis.

For both criteria, IDDs and PADs turn out to be superior to the conventional approach using BDDs. The size of the transition relation diagram is compared in Table 2. The significant reduction of the number of nodes in the transition relation PAD and in the state set IDD is obvious compared to the BDD equivalents. The

PAD size is independent of the initial configuration, while the BDD size increases heavily with it.

	PAD	BDD	
		$m = 15$	$m = 20$
# layers	6	48	60
# nodes	16	403	521
# edges	21	802	1038

Table 2: Size of transition relation diagram.

Considering the computation time, BDDs additionally have one major disadvantage in contrast to IDDs and PADs. As mentioned in Section 1, using BDDs requires an upper bound for the state variable values as they are coded binary. But such a priori bound is not known in general. As using too loose upper bounds causes substantial computation overhead—Table 3 shows the dependence of the BDD size on the chosen coding length—, the alternative in most cases would be to increase the bounds incrementally—i.e., to add bits to the coding—until they are high enough, but still tight. Each iteration of this "trial-and-error" method takes time not to be neglected, while no time is wasted using IDDs and PADs as the first run yields the final result.

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

Table 3: BDD size for $m = 15$.

Figure 7 shows the diagram size for another example model 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.

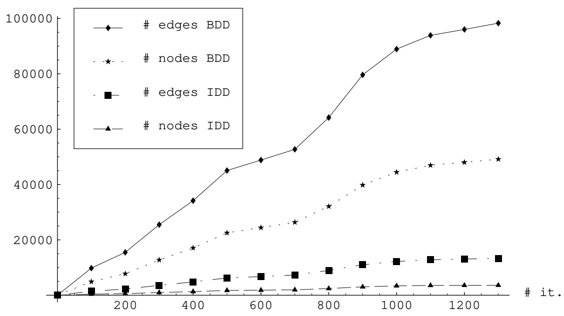

Figure 7: Diagram size during fixpoint computation.

6 Summary

Symbolic model checking tries 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 depicted by large BDDs. Especially for process networks, traditional approaches have shown not to be feasible due to the above-mentioned lacks. A new approach to symbolic model checking 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). Several drawbacks of conventional symbolic model checking of process networks with BDDs are avoided due to the use of IDDs and IMDs.

References

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

[2] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):667–691, August 1986.

[3] 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*, 13(4), 1994.

[4] 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*, 98(2):142–170, June 1992.

[5] 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.

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

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

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

[9] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, and queueing. *SIAM Journal on Applied Mathematics*, 14(6):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*, C-36(1):24–35, 1987.

[11] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–799, 1995.

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

[13] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, Berkeley, 1995.

[14] 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.

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