Symbolic Model Checking 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
Web: www.tik.ee.ethz.ch

TIK Report
No. 40, February 1998

Abstract

In this report, a representation of multi-valued functions called *interval decision diagrams* (IDDs) is introduced. It is related to similar representations as binary decision diagrams. Compared to other model checking strategies, IDDs show some important properties that enable us to verify especially Petri nets, process networks, and related models of computation more adequately than with conventional approaches. Therefore, a new form of transition relation representation called *interval mapping diagrams* (IMDs)—and their less general version *predicate action diagrams* (PADs)—is explained.

A novel approach to symbolic model checking of Petri nets and process networks is presented. Several drawbacks of traditional strategies are avoided using IDDs and IMDs. Especially the resulting transition relation IMD is very compact, allowing for fast image computations. Furthermore, no artificial limitations concerning place 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.

Contents

1 Introduction **1**
- 1.1 The Problem . . . 1
- 1.2 Petri Nets . . . 2
- 1.3 Process Networks . . . 2
- 1.4 Proposed Approach . . . 4

2 Interval Decision Diagrams **6**
- 2.1 Notation . . . 6
- 2.2 Structure . . . 8
- 2.3 Representation . . . 8
- 2.4 Reducing a Function Graph . . . 11

3 IDDs Representing Boolean Functions **13**
- 3.1 Interval Cover Disjunction . . . 13
- 3.2 If-Then-Else Operator . . . 14
- 3.3 Graph Unification . . . 16
- 3.4 MDDs and BDDs . . . 19

4 Interval Mapping Diagrams **20**
- 4.1 Notation . . . 20
- 4.2 Representation . . . 20
- 4.3 Reducing a Mapping Graph . . . 21
- 4.4 Interpretation . . . 21
- 4.5 Predicate Action Diagrams . . . 22

5 Image Computation **23**
- 5.1 Definition . . . 23
- 5.2 Using IDDs and IMDs . . . 24
- 5.3 Computation . . . 25
  - 5.3.1 Forward mapping . . . 25
  - 5.3.2 Backward mapping . . . 26
- 5.4 Image Computation With PADs . . . 26

6 Symbolic Model Checking **29**
- 6.1 Petri Nets . . . 29
  - 6.1.1 Modeling Petri Nets . . . 30

6.1.2 Model Checking 32
6.2 Process Networks 32
6.2.1 Modeling Process Networks 32
6.2.2 Model Checking 32
6.3 Experimental Results 33
6.3.1 Example Petri Net 33
6.3.2 System Models 34

7 Summary and Conclusion **39**

Chapter 1

Introduction

During the last years, a promising approach named *symbolic model checking* [BCL+94, BCM+92] was 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. It makes use of *binary decision diagrams* (BDDs) [Ake78, Bry86] that are an efficient representation of Boolean functions and allow for this very fast manipulation. But when applying this technique to Petri nets, process networks, or related models of computation—i.e., models with unbounded system states as many high-level and data-flow oriented models—, several difficulties occur that question its usefulness in this area.

1.1 The Problem

The traditional BDD-based method of automated verification suffers from the drawback that a binary representation of, e.g., the Petri net and its state is required. One severe problem is that the necessary capacity of the places in general is unknown before the analysis process or even may be its desired result. But to perform model checking, the place contents represented by an integer number have to be coded binary. To save memory and computing power, the coding should be selected such that it covers no more than the necessary integer range—which is not known yet. This deficiency could be avoided partially using *multi-valued decision diagrams* (MDDs) [SKMB90] with unbounded variable domains instead of BDDs. The decision variables of those diagrams do not only have binary, but integer values and thus are able to represent the state of a place without binary coding. 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 very regular behavior of Petri net transitions that in general consists only of consuming or producing one or a few constant numbers of tokens at a time. Consider a simple Petri net transition connecting two places. Its firing behavior with respect to its outgoing place 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 place's state and its successor state after the firing, e.g., $\{(x, x')\} = \{(0,1),(1,2),(2,3),\ldots,(n-1,n)\}$. The upper bound n in turn just has to be introduced not to make the transition relation DD too complex and thus untractable. Each of the state pairs results in at least one DD node, so neglecting all other transitions, yet the described part of the complete transition relation needs at least n nodes.

1.2 Petri Nets

Related work concerning symbolic model checking of Petri nets has been performed by Pastor, Roig, Cortadella et al. [PRCB94, RCP95]. Their focus lies on safe, i.e., 1-bounded Petri nets used to verify speed-independent circuits modeled by signal transition graphs. For this Petri net class, the binary BDD representation is adequate and allows for direct conversions as no binary coding is necessary. Although an indication to the possible extension to k-bounded Petri nets is included in [PRCB94] introducing a direct binary and a one-hot form of state variable coding, this means no fundamental advance due to the reasons stated above.

Jensen [Jen96] introduced the use of symmetry relations in state graphs of Coloured Petri nets (CP-nets) to permit succinct reasoning about certain behavioral properties of CP-nets as, e.g., reachability, liveness, or boundedness. Symmetries of the modeled system induce equivalence classes of states and of state changes. Based on these, a condensed state space may be built consisting of nodes and arcs each representing equivalence classes of states or state changes, respectively, without affecting the analytical power. Using a similar concept concerning inherent symmetries of modeled systems, Clarke et al. [CEFJ96] describe a correspondence between a global state graph and its symmetry reduced quotient graph, preserving the correctness of all CTL* formulae—a generalization of CTL formulae mostly used in model checking. Furthermore, the determination of the symmetry relation is investigated. This paper proves that symmetry relations are a valuable technique for efficient symbolic model checking. Jørgensen and Kristensen [JK97] broaden Jensen's approach to a more general notion of equivalence relations merging similar system states and actions. As practice has shown, to find such equivalence relations is not an easy task and likely the major obstacle inhibiting their widespread use to improve formal analysis of state transition systems. In some sense, our approach is related to this one as it may be considered as a structured method to determine certain kinds of equivalence classes and using them in symbolic model checking.

1.3 Process Networks

The drawback of symbolic model checking of process networks is very similar to that of Petri nets. Process network models—consisting in general of concurrent processes communicating through unidirectional FIFO queues—as that of Kahn

[Kah74, KM77] are commonly used, e.g., for the 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* [LP95], *computation graphs* [KM66], and *synchronous dataflow* (SDF) [LM87], 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 [Par95].

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 [Par95] may be detected.

A simple example process network out of [Par95] is shown in Figure 1.1. 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.

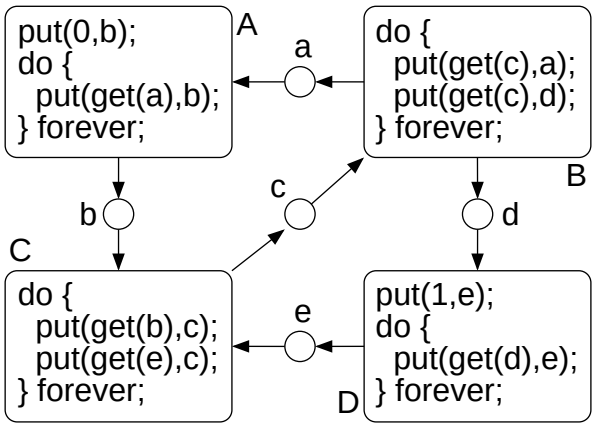

Figure 1.1: Simple process network.

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.

An approach with aims similar to our's is the one of Godefroid and Long [GL96]. 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. 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.

1.4 Proposed Approach

To overcome the above-mentioned limitations of conventional symbolic model checking of Petri nets, process networks, and related models, 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 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 to an upper state bound is necessary. 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 models like Petri nets or process networks—as 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 result in less nodes than using BDDs.

We introduce the formalism of interval decision diagrams, an efficient representation for discrete-valued functions, and the methods and techniques necessary to apply this new form of function description to symbolic model checking. IDDs and IMDs

are dedicated for state set and transition relation descriptions of many, especially data-flow oriented models of computation as Petri nets, process networks, or similar models describing FIFO-based systems with generally unbounded system states. In this report, mostly Petri nets or process networks are regarded exemplarily without meaning further restrictions.

Chapter 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 $P_i \subseteq \mathbb{R}$ 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 $I \subseteq \mathbb{R}$, 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 $b \in \mathbb{R}$, the literal of variable x with respect to I is denoted $x^b = x^I$.

In the following, we will mainly deal with intervals on $\mathbb{R}$ or $\mathbb{Z}$ denoted as real or integer intervals, respectively. Intervals can be, e.g., of the forms $[a, b]$, $(a, b]$, or $[a, \infty)$, where a, b, and ∞ represent the upper and lower bounds. Squared brackets [and] indicate that the corresponding bound is included in the interval set, whereas round brackets (and) indicate that it is not. An interval $[a, b]$ is named *closed*, while (a, b) is an *open* interval and $[a, b)$ a *half-open* one. [ ] represents the empty interval containing no elements. 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.*

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

$$f = \sum_{I \in I(P_i)} x_i^I \cdot f_{x_i^I}. \tag{2.1}$$

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 of this is by contradiction.

2.2 Structure

An example interval decision diagram is shown in Figure 2.1. 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

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

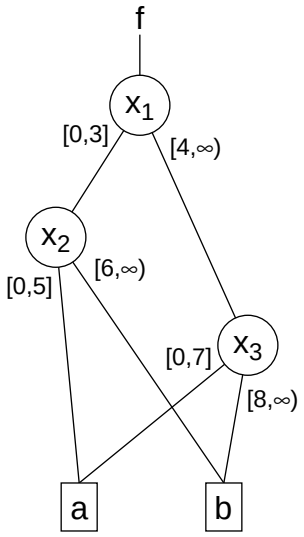

Figure 2.1: Example interval decision diagram.

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 real or integer intervals.

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) [CFZ96, RCP95]. 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. MDD's integer edge weights may be described by rudimentary one-element integer intervals with equal upper and lower bounds, likewise BDD's binary values.

2.3 Representation

IDDs are represented by canonical *function graphs*, similar to those of [SKMB90] and [Bry86].

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 $child_k(v) \in V, 1 \leq k \leq p_i$. 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* (2.1)*, thus*

$$f_v = \sum_{I_j \in part(v)} x_i^{I_j} \cdot f_{child_j(v)}. \tag{2.2}$$

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 [Bry86]), and*

3. *the independence interval partitions $part(v)$ of all non-terminal nodes v are reduced.*

Now, one of the major results of this report 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 [Bry86] and [SKMB90], 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, too. 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 show 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 $|D_g| < d$ holds: If the function graph H of g is reduced, it is unique. Then for each function f with $|D_f| = d$, if its function graph G is reduced, then it is unique. The proof of this is subdivided into two parts. First, we claim that as all subfunctions g associated to 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

functions graphs. This would violence the fact that G is reduced. In particular, all children of the root node v represent different functions depicted 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 to 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 it 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 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 that G has the minimum number of nodes. Hence, G must be the unique reduced function graph. ■

In general, the functions used during a computation are depicted not in several IDDs, each for a specific function, but in a common IDD that represents all functions at once. Thus, identical part IDDs that would have to be stored separately may be shared among the others resulting in a significant reduction of the total number of IDD nodes. 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.

2.4 Reducing a Function Graph

Similar 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 to it any more. Two distinct nodes with the same index and same interval partitions are named *equal-childed* if all children of one of the nodes equal the corresponding children of the other node. *Neighbored edges* are edges starting in the same node, labeled with neighbored 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 have to be joined into one by eliminating the other 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. Neighbored edges are joined into one by uniting their split intervals.

A node which has to be removed due to rule 1 is named *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 2.8. 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 2.2 shows an example for the reduction of a function graph. For the examples shown, the integer interval $[0, \infty)$ is used as variable domain if not otherwise mentioned. Interval covers must be complete with respect to this domain set.

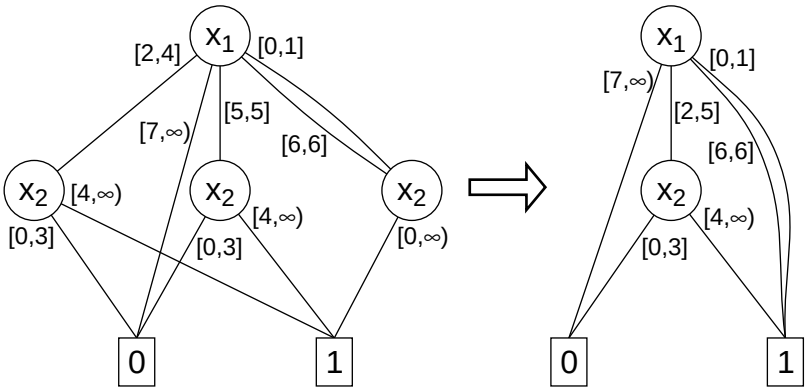

Figure 2.2: Graph reduction.

Chapter 3

IDDs Representing Boolean Functions

For conventional symbolic model checking of the described class of models of computation, basically Boolean functions over integer variables are of importance. Hence, in this section, 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., $f(x_1, x_2) = (x_1 \leq 7) \wedge (x_2 = 3) \vee (x_2 \geq 6) = x_1^{[0,7]} \cdot x_2^{[3,3]} \vee x_2^{[6,\infty)}$ are describable. Nevertheless, part of the described techniques are also applicable to IDDs over real intervals or non-Boolean functions.

The most important operations on Boolean IDDs are presented in this section. Some fundamental operations are equivalent for real and integer IDDs. For the sake of brevity, only integer intervals are treated, as for real intervals, many exceptions arise resulting from the fact that an interval bound may either belong to the interval itself or not as the bound may be closed or open, respectively. From now on, we assume that all integer intervals are depicted by closed intervals, where the only exceptions are positive or negative infinite bounds that have to be open.

3.1 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. The GCI partition $\tilde{I}(P_i)$ of a cover $I(P_i)$ is the unique minimal interval partition with the property

$$I_j = \bigcup_{k=1}^{\tilde{p}_i} (B_{jk} \cap \tilde{I}_k), \forall j \in \{1, 2, \ldots, p_i\}, B_{jk} \in \{[\,], (-\infty, \infty)\}, \tag{3.1}$$

where B_{jk} indicate whether a GCI $\tilde{I}_k$ is included in the original interval I_j. The postulated property of $\tilde{I}(P_i)$ to be minimal must hold with regard to the number $\tilde{p}_i$

of GCIs, thus supposing that all $\tilde{I}_k$ are as large as possible and no neighbored GCIs $\tilde{I}_k$ may be joined by union without violating the equation.

For integer intervals, an efficient procedure to determine the GCIs of an interval cover is described in Table 3.1. First, the original integer intervals $[a, b]$ are transformed into real ones $[a', b']$ including them as shown in Figure 3.1. This is achieved by adding the value $-\frac{1}{2}$ or $\frac{1}{2}$ to the lower or upper bound, respectively, of the original interval. The resulting bounds are sorted. Then the disjoint intervals are determined by back-transforming successive real bounds into integer intervals $[\tilde{a}, \tilde{b}]$.

$determineGCIs\big(I(P_i)\big)$:
  `for each` interval $[a, b] \in I(P_i)$
    $[a', b'] = [a - \frac{1}{2}, b + \frac{1}{2}]$;
    add a' and b' to unique sorted list L;
  `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` $\tilde{I}(P_i)$;

Table 3.1: Interval cover disjunction.

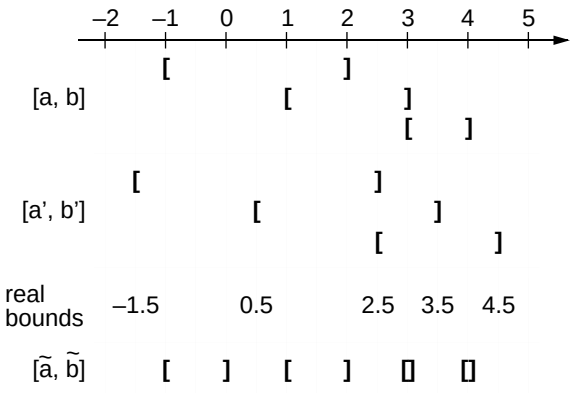

Figure 3.1: Example for interval cover disjunction.

3.2 If-Then-Else Operator

The *If-Then-Else* operator (ITE) [Bry86] 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 [SKMB90]. For the sake of brevity and clearness, 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. E.g., $AND(F,G)$ 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. Analog to [BRB90], 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 $ITE(1,F,G) = ITE(0,G,F) = ITE(F,1,0) = F$.

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 which in general is not disjoint. Then this cover is transformed as described in Section 3.1 into an independence interval partition. The resulting partition $\{I_1, I_2, \ldots, I_p\}$ 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 inserting into the IDD. The ITE algorithm is sketched as pseudo code of the function $ite(F,G,H)$ in Table 3.2.

```
ite(F,G,H) :
    if terminal case reached
        return result of terminal case;
    let x be the top variable of {F,G,H};
    mutually decompose interval partitions of top nodes of {F,G,H} if
      corresponding to x into GCI partition {I_1, I_2, ..., I_p};
    for each resulting split interval I_k:
        set child_k = ite(F_{x^{I_k}}, G_{x^{I_k}}, H_{x^{I_k}});
    merge neighbored I_k if child_k are equal;
    if only I_1 left
        return child_1;
    return findOrAddInIDD(x, (I_1, child_1), ..., (I_k, child_k));
```

Table 3.2: The ITE algorithm.

The function $findOrAddInIDD(v)$ first searches the IDD for the argument node v. If a such node is already included in the IDD, it is returned. Otherwise, a new node with the specified properties is created and inserted in the IDD. As the new node is reduced, a reduced form of the IDD is maintained.

Figure 3.2 shows an example for the application of the If-Then-Else operator where $I = ITE(F, G, H)$. For the sake of clearness, the part IDDs are depicted separately and not in one common IDD representing all used functions. The calculation for the ITE operation is sketched in Table 3.3. The structure of the resulting IDD can directly be read off from the last row of this calculation.

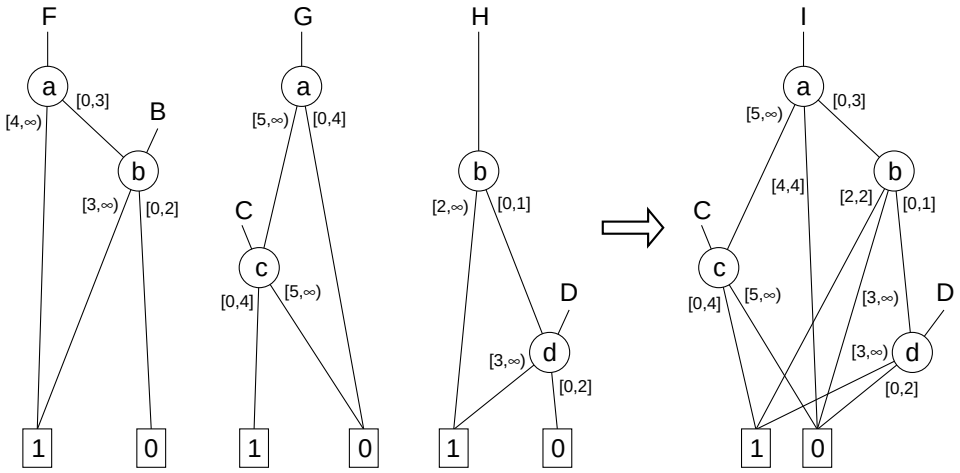

Figure 3.2: If-Then-Else operator.

3.3 Graph Unification

Under certain circumstances, a graph may be of importance which 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 a such 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 certain value for each variable—always end in the same terminal node, the graph represents a unique function even if it has an ambiguous structure. But 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 to a node v in equation (2.2) to interval covers as

$$f_v = \sum_{I_j \in cover(v)} x_i^{I_j} \cdot f_{child_j(v)}, \tag{3.2}$$

where $cover(v)$ replaces the partition of node v. Hence, the *covering rule* predicates that the value of the corresponding function is 1 for all assignments which have

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

Table 3.3: Example for ITE computation.

at least one evaluation path ending in the 1-terminal node, otherwise it is 0. An example is shown in Figure 3.3.

Now, a general method is described which enables the transformation of an ambiguous graph into a function graph. A procedure is applied to the whole graph that transforms all interval covers into partitions, using the cover disjunction of Section 3.1 and the covering rule described above. Beginning at the root and stepping downwards, all nodes v of each non-terminal layer are transformed one after the other by applying the procedure $transformNode(v)$ described in Table 3.4 as pseudo code. TN_0 and TN_1 represent the 0- and 1-terminal node, respectively. Thus, the disjointness of the interval covers propagates from top to bottom.

This procedure performs the transformation of the node cover into a partition and rearranges the induced child nodes corresponding to the split intervals such that ambiguities in this node are removed preserving the original meaning. Of all equal disjoint intervals, the *1-intervals*—corresponding to edges ending in the 1-terminal node—dominate all others. The covered split intervals and their corresponding edges are ignored. *0-intervals*—ending in the 0-terminal node—are dominated by all other equal split intervals and thus are ignored, too.

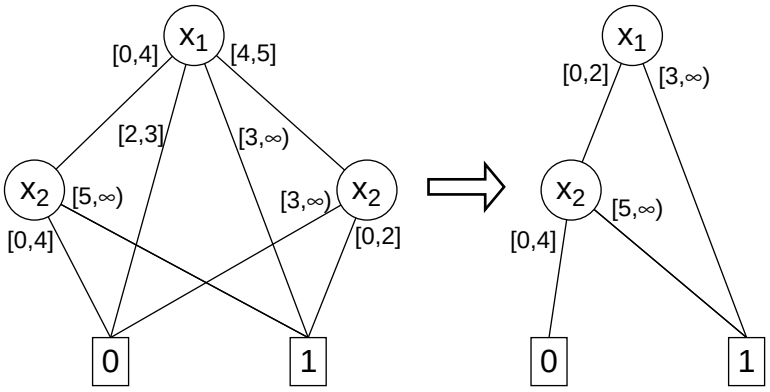

Figure 3.3: Covering rule.

We figured out that the combined application of unification and reduction is recommendable to minimize the extent of computation. First, the three reduction rules of Section 2.4 are applied as long as possible. They may straightforwardly be adopted for non-disjoint interval covers without essential changes. Thus, the number of nodes and edges is minimized while the ambiguity of the graph is not affected. Then, the ambiguous graph is unified as described above. After application of this transformation to all nodes, the three reduction rules are applied again to the resulting function graph until none of them holds any more. Eventually, the ambiguous graph has been transformed into a reduced function graph representing a valid IDD. Figure 3.4 shows the node transformation of the top node of an example graph. The application of the above-mentioned covering rule concerning 0- and 1-edges is obvious.

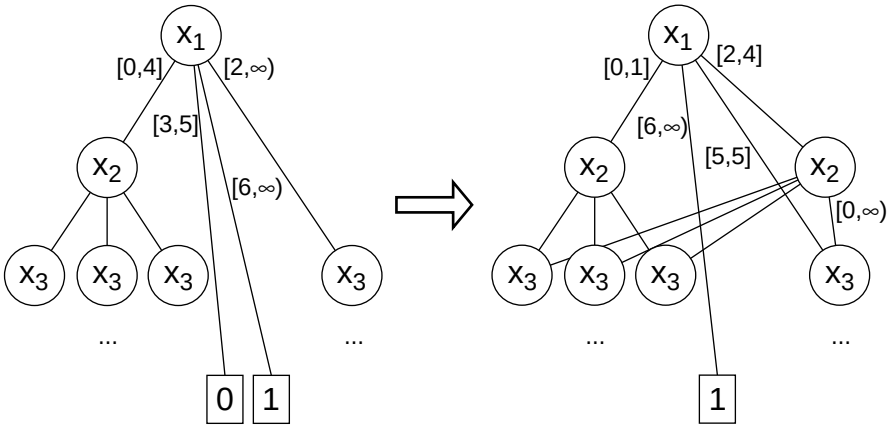

Figure 3.4: Graph unification.

As the unification procedure begins at the top of the graph, and as during the transformation of one node only its induced nodes are affected, but later transformed themselves, the procedure guarantees to transform all interval covers into partitions. Those are reduced later by applying the reduction rules preserving the disjointness

$transformNode(v)$:
  join neighbored edges of v;
  `if` cover of v is disjoint
    `return;`
  create new node $\tilde{v}$ with $index(\tilde{v}) = index(v)$;
  transform cover $\{I_1, I_2, \ldots, I_{p_i}\}$ of v into partition $\{\tilde{I}_1, \tilde{I}_2, \ldots, \tilde{I}_{\tilde{p}_i}\}$ of $\tilde{v}$;
  `for each` resulting split interval $\tilde{I}_j$:
    determine set $I = \{I_k \supseteq \tilde{I}_j | child_k(v) \neq TN_0\}$;
    `if` exists $I_k \in I$ with $child_k(v) = TN_1$
      set $child_j(\tilde{v}) = TN_1$;
    `else if` I is empty
      set $child_j(\tilde{v}) = TN_0$;
    `else`
      let i_{min} be the minimum index of all $child_k(v)$ corresponding
        to $I_k \in I$;
      create new node $child_j(\tilde{v})$ with index i_{min};
      `for each` $I_k \in I$
        `if` $index\big(child_k(v)\big) > i_{min}$
          add quasi-obsolete edge ending in $child_k(v)$ to $child_j(\tilde{v})$;
        `else`
          add all edges leaving $child_k(v)$ to $child_j(\tilde{v})$;
  replace v by $\tilde{v}$ and remove unused $child_k(v)$;

Table 3.4: Node transformation during unification.

of the interval partitions. Hence, the resulting graph is guaranteed to be a reduced function graph.

3.4 MDDs and BDDs

As IDDs are a generalization of MDDs and those in turn of BDDs, the latter both may be described using IDDs. For the sake of brevity, only the transformation of a BDD into an IDD is explained here. Moreover, Boolean-valued MDDs may be mapped directly onto BDDs as stated in [SKMB90] thus allowing to describe those MDDs as IDDs over their BDD form.

To depict states and transitions of binary coded systems by IDDs, bounded state spaces are used that limit the values of the state variables to the Boolean values 0 and 1. Hence, the variable domain is $[0, 1]$. In the IDD, the Boolean value 0 is depicted by the interval $[0, 0]$, value 1 by $[1, 1]$, and a "don't care" by $[0, 1]$ which results in an obsolete node that is removed in a reduced IDD.

Chapter 4

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 the interval shifting, a simple interval arithmetic is used.

4.1 Notation

$\mathbb{I}$ denotes the set of all intervals, depending on the integer or real range, respectively. The underlying interval mathematics consist of a simple unbounded interval arithmetic with the operators $+$ for *addition* and $-$ for *substraction*, 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 infite ones. The addition or substraction of empty intervals is not legal.

Definition 4.1 (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 4.2 (Interval substraction) *The substraction of two intervals is defined as* $[a, b] - [c, d] = [a, b] + [-d, -c]$.

Hence, the result of the substraction is the set of all numbers that could have been the result of the substraction of any two elements of the argument intervals.

4.2 Representation

IMDs are represented by *mapping graphs*, similar to the function graphs described in Definition 2.5. They contain *interval mapping functions* $f : \mathbb{I} \to \mathbb{I}$, mapping

intervals onto intervals.

Definition 4.3 (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 due to the similarity of IMDs and IDDs. The interval mapping function f mapping an interval onto itself, i.e., $f(I) = I$, is named *neutral*.

4.3 Reducing a Mapping Graph

Unlike IDDs, IMDs in general have no canonical forms. Nevertheless, depending on the structure of the contained mapping functions, a partial reduction is often possible. For instance, as no equivalence checkings or ITE operations of IMDs have to be performed in contrast to IDDs, this is no general limitation. But smaller IMDs yet lead to a reduction of the computational complexity during image computation.

Similar to IDDs, two distinct nodes with the same index and same mapping function sets are named 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 one is labeled with a neutral mapping function, this 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 a such 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.

4.4 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 the 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 5.

4.5 Predicate Action Diagrams

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

Definition 4.4 (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 the* predicate interval *and I_A the* action interval.

Thus, the interval mapping functions have one out of two possible structures. Both types first intersect the argument interval with the predicate interval and then—if it is not empty—either shift the resulting interval up or down or assign a default interval. The shift is performed by interval addition of the action interval and returning the result. 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 reverse direction corresponding to interval substraction 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 predicate interval and $[0,0]$ as action interval, corresponding to $P_i/+[0,0]$. Assign functions cannot be neutral.

Chapter 5

Image Computation

5.1 Definition

In general, symbolic model checking is based on Boolean functions representing characteristic functions of sets consisting of 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 depicting a system state. Then a state set S is represented by its characteristic function $s(x)$ with

$$s(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases} . \tag{5.1}$$

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 $\delta(x, x')$ a characteristic function representing the transition relation T, i.e.,

$$\delta(x, x') = \begin{cases} 1 & \text{if } (x, x') \in T \\ 0 & \text{otherwise} \end{cases} . \tag{5.2}$$

In symbolic model checking, 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 $Im(S, T)$ 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 following definitions are taken from [HD93]. The image operator is defined as $Im(S, T) = \{x' | \exists x \text{ with } s(x) \wedge \delta(x, x')\}$. The inverse image $PreIm(S, T)$ depicts all states that in one transition *can* reach a state in S and is defined as $PreIm(S, T) = \{x | \exists x' \text{ with } s(x') \wedge \delta(x, x')\}$. $BackIm(S, T)$ represents the set of states that in one transition *must* end up in S, denoted as $BackIm(S, T) = \{x | \forall x' \text{ holds } \delta(x, x') \implies s(x')\} = \neg PreIm(\neg S, T)$. The last operator behaves in some sense as the inverse of the image operator because $BackIm(S, T)$ is the largest set such that $Im\big(BackIm(S, T), T\big) \subseteq S$.

5.2 Using IDDs and IMDs

Figure 5.1 shows an example process network with unspecific consumption and production rates represented by intervals. It is similar to a computation graph [KM66] where the consumption rate is independent of the threshold—depicted as a condition here.

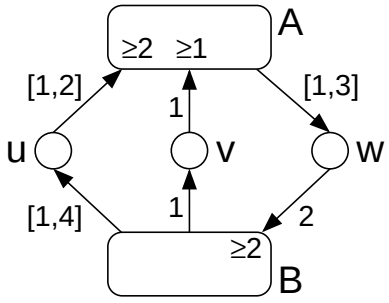

Figure 5.1: Example process network.

Figure 5.2 b) shows the corresponding transition relation represented by the PAD T, Figure 5.2 a) an example state set IDD S. 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 depiction named *lean description* has no impacts on the real internal representation of an implementation. It should not be confused with the DD class of *zero-suppressed BDDs* (ZBDD) [Min93] which are a BDD derivative based on modified reduction rules.

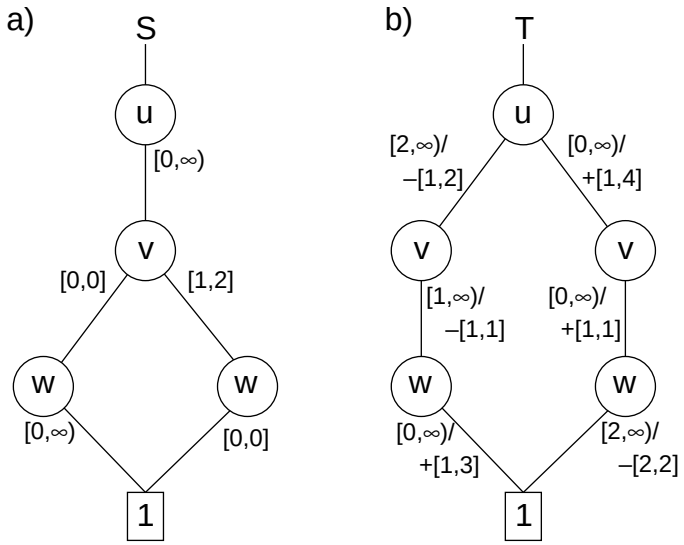

Figure 5.2: State set IDD and transition relation PAD.

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 $u, v, w, u', v', w' \in [0, \infty)$.

5.3 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 [McM93] for instance. This strategy is possible for IDDs, too.

Another technique is described in the following. A major advantage of IMDs in contrast to BDDs is the possibility to efficiently store only mapping functions describing partial transition behaviors instead of many state variable value pairs. In this way, image calculations 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, similar to Bryant's *Apply* operator [Bry86]. First, we describe 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.

5.3.1 Forward mapping

With the use of 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 5.1. 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 [Bry86].

Obviously, whenever the empty interval [ ] is the result of an interval operation, the recursion during the image calculation is stopped at this point, continuing with the next edge. Left-out layers in the transition relation IMD are implicitly treated as "dummy" mapping functions returning their argument interval without changes. As a memory function, a hash table of already computed results for pairs of v and w is maintained—omitted in Table 5.1—such that an included result may be returned

without further computation. Left-out IDD layers implying quasi-obsolete nodes are treated separately. Only those edges have to be concerned not ending in the 0-terminal node.

5.3.2 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 CTL model checking using $PreIm(S,T)$ and $BackIm(S,T)$, 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, PADs have valid inversions which are denoted below.

The applied interval mapping functions depend on the kind of image operation that is performed. To determine the image $Im(S,T)$ of state set S with respect to transition relation T, normal 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 analog 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.

5.4 Image Computation With PADs

Image computation now will be demonstrated with an example. As mentioned above, PADs are dedicated to perform image computation especially for Petri nets, process networks, and related models of computation as the state distance combined with the respective firing condition may be stored efficiently instead of many state pairs. The state distance between two system states x and x' is defined as $\Delta x = x' - x$. Thus, according to equation (5.2), the transition relation T may be described as the characteristic function

$$\hat{\delta}(x,\Delta x) = \begin{cases} 1 & \text{if } (x, x+\Delta x) \in T \\ 0 & \text{otherwise} \end{cases} . \tag{5.3}$$

In Figure 5.2 b), only shift functions f_+ are used as mapping functions. In the case of place or 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 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 Petri nets with safe places, i.e., places with bounded capacity of 1. The transition relation PAD of a binary coded system must replace Boolean operations like 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 5.3 shows the application of the image operator and its result. In the left part, for better understanding an intermediary IDD is depicted that does not really exist during the computation, but is more comprehensive 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 part of Figure 5.3. The broken edges labeled with empty intervals represent such locations where the recursion of $mapForward(v, w)$ is stopped due to a resulting empty interval.

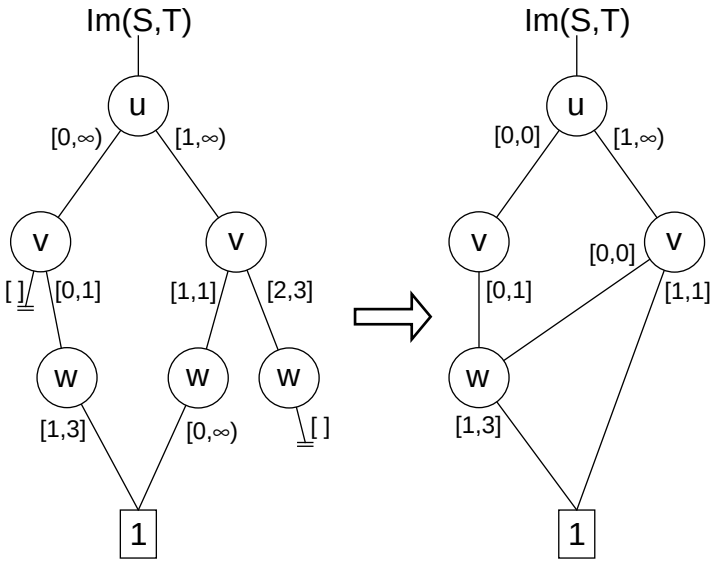

Figure 5.3: Computation of $Im(S, T)$: IDD before and after unification.

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

Definition 5.1 (Inverse PAD mapping functions) *The inverse shift function is denoted as*

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

and the inverse assign function as

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

```
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 new node ṽ with index(v);
                    add new edge with interval I_res ending in v_c to ṽ;
                    [...]
  return v_res;
```

Table 5.1: Forward mapping for image computation.

Chapter 6

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 [McM93]. To verify such CTL formulae, operations like ITE and image computation are used.

The most frequently employed form of symbolic representation are BDDs and their derivatives, e.g., see [BCM+92, McM93, PRCB94]. To perform conventional symbolic model checking, the investigated system is described using BDDs to depict its transition relation and certain state sets. As mentioned in Section 1, for Petri nets and process networks this is no convenient form of representation for several reasons.

- 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 for Petri nets and 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.

6.1 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 depicts 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 depicts the actual number of tokens in each place. The behavior of a Petri net is defined as usual. A transition $t \in T$ is sensitive to fire if all its incoming places contain at least as much tokens as

the respective arc weight and all its outgoing places are able to absorb at least as much tokens as the respective arc weight without exceeding their place capacity.

6.1.1 Modeling Petri Nets

First, we consider symbolic model checking of Petri nets with infinite place capacities. Finally, we shortly describe the necessary adaptions for bounded capacities. Half-bounded state spaces with a lower bound of 0 and without an upper bound are dedicated for symbolic model checking of Petri nets with unbounded place capacities. Such Petri net places have the state variable domain $P = [0, \infty)$.

To describe the principal use of IDDs and IMDs for symbolic model checking of Petri nets, Figure 6.1 shows an example Petri net [Jan83]. The Petri net places are of unbounded capacity. Arc weights of 1 are left out in the figure.

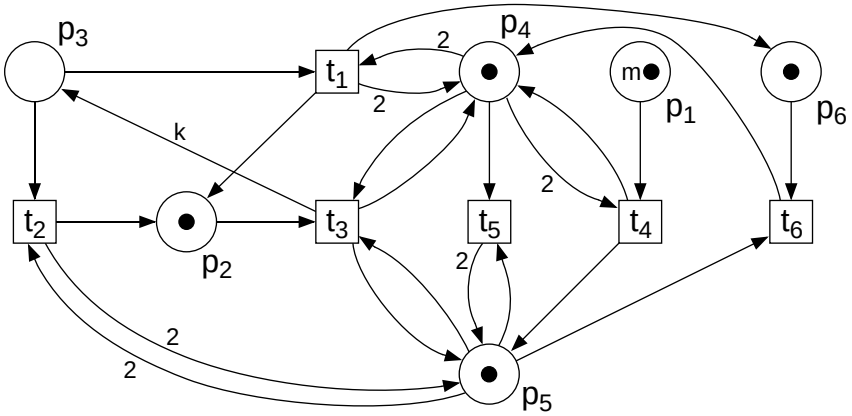

Figure 6.1: Example Petri net.

Figure 6.2 shows the reduced PAD of the corresponding transition relation. Action intervals containing only one element are abbreviated by this only element. Left-out predicate or action intervals stand for non-existing predicates or value changes, respectively.

Each path in the transition relation PAD describes one of the Petri net transitions. The paths in the example PAD correspond to the Petri net transitions t_i ordered from left to right. The mapping functions along a path depict 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. Otherwise, the computation of $mapForward(v, w)$ leads to an empty interval and stops the calculation for this transition. 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.

The lower bound of a predicate interval is different from 0 only for transitions consuming tokens from the respective place. In this case, the lower bound is equal to the consumption rate. For unbounded place capacities, the upper bounds of the predicate intervals are infinite. For loop-transitions, the action interval consists of

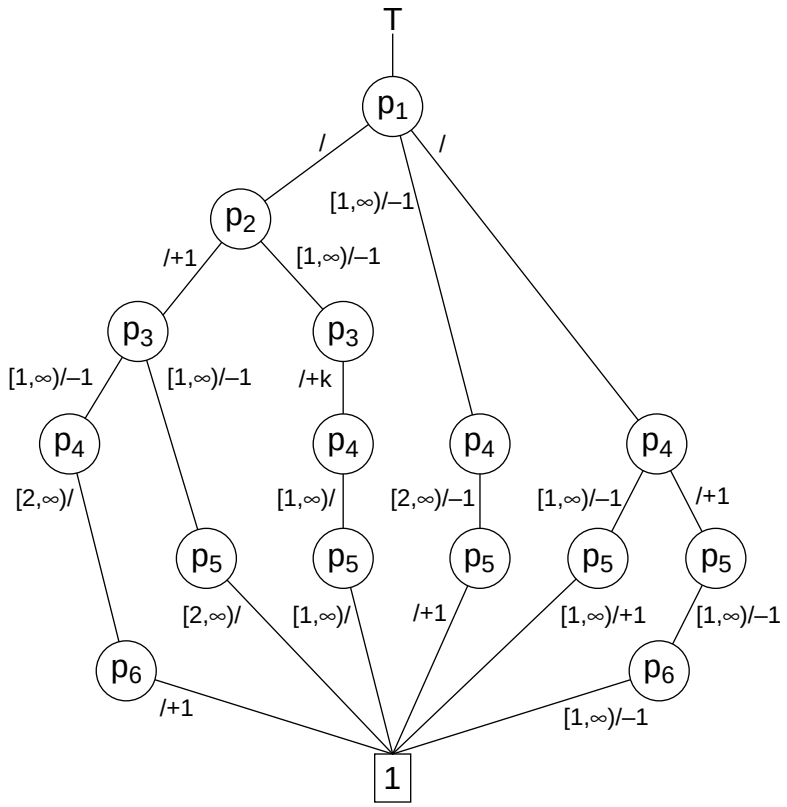

Figure 6.2: Transition relation PAD.

the difference between the production and consumption rate, otherwise it includes only the production or consumption rate with positive or negative sign, respectively. Omitted variable layers along a path result from places not influenced by the respective transition. An inhibitor arc with the weight k may be represented by a shift function with the closed predicate interval $[0, k-1]$ and the action interval $[0, 0]$. Hence, it influences the enabling of a transition, but does not participate in the firing.

Even if half-bounded state spaces are favorable to describe Petri nets with unbounded 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 matter, 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.

6.1.2 Model Checking

Properties to be verified using symbolic model checking of Petri nets may be described as CTL formulae and verified as usual [McM93]. Only 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 [PRCB94] and [RCP95] and are straightforwardly adaptable to IDDs, e.g., for

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

6.2 Process Networks

6.2.1 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 transition. The mapping functions along a path depict enabling conditions and the corresponding state variable changes. Analog to computation graphs [KM66], 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.

6.2.2 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 [McM93].

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.

6.3 Experimental Results

6.3.1 Example Petri Net

Astonishingly, the Petri net in Figure 6.1 is inherently bounded in an unobvious, but very regular way [Jan83]. The Petri net is parameterized by the weight $k \geq 2$ of the arc between t_3 and p_3 and the initial marking m of place p_1. 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:

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

Starting with the initial state, the set of reachable states has been calculated by a series of image computations as described in [PRCB94] using the operators of Section 5 until a fixpoint has been reached. Some results for $m = 1$ and $k = 3$ are shown in Table 6.1.

	IDD/PAD	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

Table 6.1: Results for $m = 1$ and $k = 3$.

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 6.1 suffices to represent all possible state variable values. 494 fixpoint iterations are necessary to determine the set of reachable states. In the BDD version, the coding of the state variable values was direct binary.

First successful investigations yielded promising results concerning the number of nodes and edges as well as the computation speed. Especially 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.

For $m = k = 2$, the final result could not have been determined yet due to memory limitations. Figure 6.3 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.

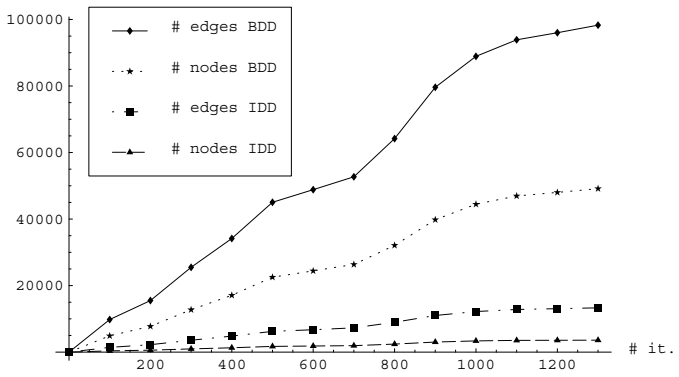

Figure 6.3: Diagram size during fixpoint computation.

In the used implementation, the edges of the non-terminal IDD nodes are stored in conventionally linked, ordered lists, causing considerable effort when adding edges to a node or searching an edge list for a certain interval as during ITE computation. Hence, essential improvements could be achieved by using binary search trees instead of linear lists as more efficient search and ordering algorithms are available. Investigations have shown that about one half of the IDD edges are 0-edges and thus only necessary to preserve the completeness of the interval partitions. By considering 0-edges implicitly, e.g., in the ITE algorithm, they could be omitted to minimize memory requirements and to reduce the length of the above-mentioned edge lists.

6.3.2 System Models

Several diverse system models based on FIFO queues have been investigated, producing promising results. 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 6.4 shows the size of the diagram representing the set of reachable states of a model of a symmetric multiserver random polling system [ABC+95] for increasing initial configurations.

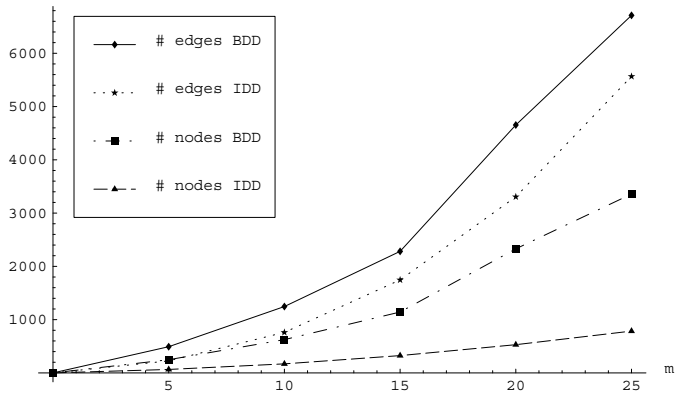

Figure 6.4: Size of state set diagram of polling system.

In Figure 6.5, the duration of the computation to determine the set of reachable states is depicted depending on the initial configuration.

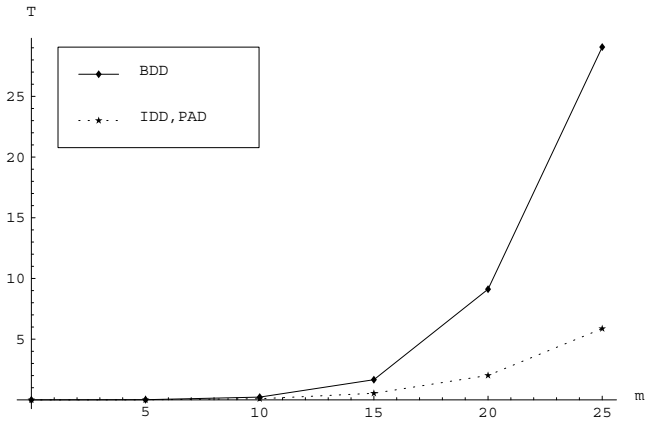

Figure 6.5: Duration of reachability analysis of polling system.

Figure 6.6 shows the size of the state set diagram of a readers and writers system accessing a common database. In Figure 6.7, the duration of the reachability analysis is shown.

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 6.2 for the readers and writers system. 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.

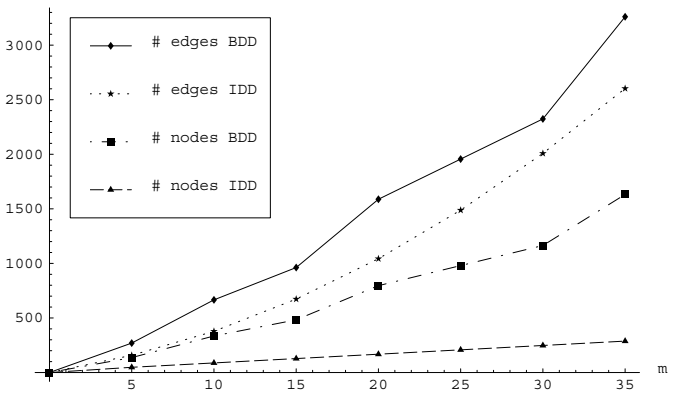

Figure 6.6: Size of state set diagram of readers and writers system.

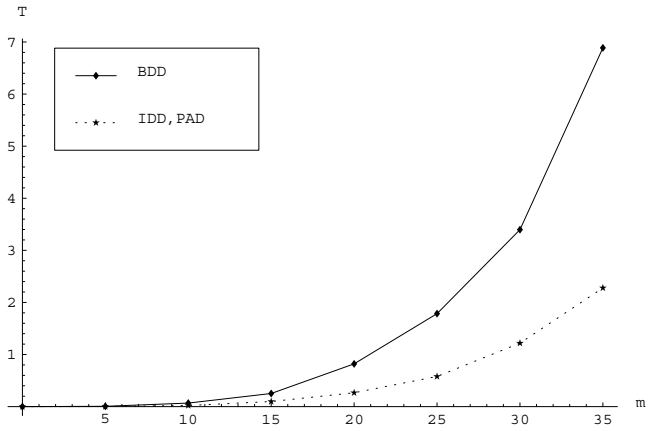

Figure 6.7: Duration of reachability analysis of readers and writers system.

In Figure 6.8, the state set diagram size of a flexible manufacturing system with automated guided vehicle is depicted. Figure 6.9, shows the duration of the reachability analysis.

The transition relation diagram size of the manufacturing system is compared for PADs and BDDs in Table 6.3.

Considering the duration of the computation, 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 6.1 and Table 6.4 show 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

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

Table 6.2: Size of transition relation diagram of readers and writers system.

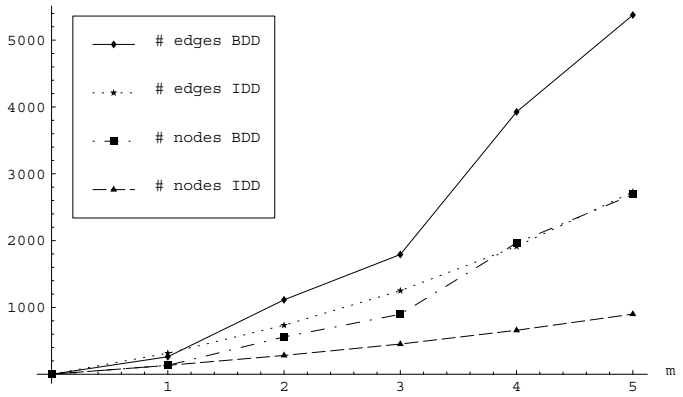

Figure 6.8: Size of state set diagram of manufacturing system.

run yields the final result.

	PAD	BDD	
		$m = 3$	$m = 4$
# layers	27	108	162
# nodes	67	1233	1957
# edges	89	2462	3910

Table 6.3: Size of transition relation diagram of manufacturing system.

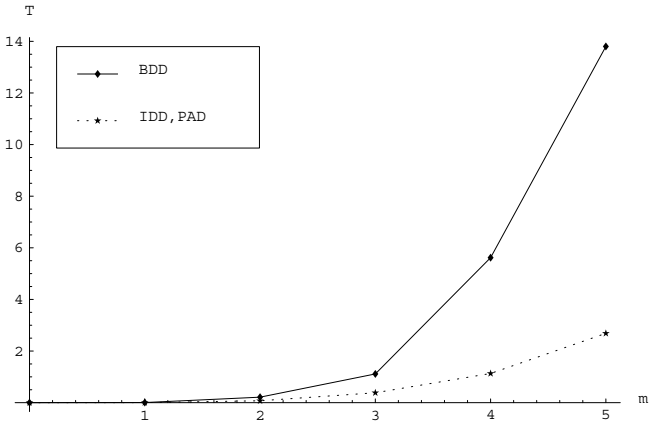

Figure 6.9: Duration of reachability analysis of manufacturing system.

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 6.4: BDD size of polling system for $m = 15$.

Chapter 7

Summary and Conclusion

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 huge BDDs. Especially for unbounded Petri nets and process networks, traditional approaches have shown not to be feasible due to the above-mentioned lacks. A new approach to symbolic model checking of Petri nets, process networks, and related models of computation has been presented. It is based on a novel, efficient form of representation for multi-valued functions called interval decision diagram (IDD) and the corresponding image computation technique using interval mapping diagrams (IMDs).

A specialized variant of the latter, the predicate action diagram (PAD), is dedicated to represent the transition relation of, e.g., Petri nets or process networks, depicting 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 Petri nets and process networks with BDDs are avoided due to the use of IDDs and IMDs. Especially the resulting transition relation IMD is very compact, allowing for fast image computations. Furthermore, no artificial limitations concerning place or queue capacities have to be introduced. Future research concentrates on the above-mentioned improvements of the internal IDD representation of the implementation and, for instance, on efficient heuristics concerning the variable ordering as this in general is one of the most influencing factors in the area of symbolic model checking.

Bibliography

[ABC+95] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, 1995.

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[BCL+94] 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.

[BCM+92] 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.

[BRB90] 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.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):667–691, August 1986.

[CEFJ96] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9:77–104, 1996.

[CFZ96] E. M. Clarke, M. Fujita, and X. Zhao. Multi-terminal binary decision diagrams and hybrid decision diagrams. In T. Sasao and M. Fujita, editors, *Representations of Discrete Functions*, pages 93–108. Kluwer Academic Publishers, 1996.

[GL96] 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.

[HD93] 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.

[Jan83] M. Jantzen. The large markings problem. *Petri Net Newsletter*, 14:24–25, 1983.

[Jen96] K. Jensen. Condensed state spaces for symmetrical coloured Petri nets. *Formal Methods in System Design*, 9:7–40, 1996.

[JK97] J. B. Jørgensen and L. M. Kristensen. Verification of coloured Petri nets using state spaces with equivalence classes. In *Proceedings of the Workshop on Petri Nets in System Engineering*, 1997.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress Information Processing*, 1974.

[KM66] 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.

[KM77] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *Proceedings of the IFIP Congress Information Processing*, 1977.

[LM87] 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.

[LP95] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–799, 1995.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[Min93] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th IEEE/ACM Design Automation Conference*, 1993.

[Par95] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, Berkeley, 1995.

[PRCB94] 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*, volume 815 of *Lecture Notes in Computer Science*, pages 416–435. Springer-Verlag, 1994.

[RCP95] 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*, volume 935 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[SKMB90] 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.