Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. 36

Karsten Strehl

Symbolic Methods Applied to Formal Verification and Synthesis in Embedded Systems Design

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A dissertation submitted to the
Swiss Federal Institute of Technology Zurich
for the degree of Doctor of Technical Sciences (Dr. sc. techn.)

Diss. ETH No. 13572

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Rolf Ernst, co-examiner

Examination date: February 17, 2000

Abstract

In many of today's technical appliances, embedded systems are deployed on an increasing scale. This trend is expected to massively gain in importance, especially for applications in consumer and automotive electronics, telecommunications, medical engineering, and avionics. Embedded systems can be characterized as follows:

- They mostly consist of both hardware and software and perform control or communication tasks within an overall system, to which they are connected via networks, sensors, and actors.
- In general, they are heterogeneous, often distributed systems with real-time requirements which may work on both transformative and reactive domains.
- They typically are not directly visible to the user who interacts unconciously with them.

The design of embedded systems poses significant challenges since system complexity grows rapidly. In particular, very elaborate design methods are necessary, and quality requirements are high because embedded systems often realize safety-critical functions. Hence the use of design automation tools has become indispensable. These can—based on a formal representation—analyze system properties and prove the correctness of the design. Then the implementation can be *synthesized* by automatically generating hardware and software descriptions. This enables a more flexible, faster, and less expensive design process than without automation.

Within this context, the following topics are subject of this monograph:

- A formal representation model called *FunState* is introduced which is especially suited to support embedded systems design. *FunState* serves as an internal design representation and can be used for modeling mixed hardware/software systems for analysis and synthesis purposes.
- Based on *FunState*, a *formal verification* approach is presented which makes it possible to automatically check whether desired system properties are satisfied. In contrast to conventional verification techniques, formal verification can prove the correctness of a system's model mathematically.

- In order to generate an implementation, the execution order of functional parts within the system has to be determined, often in compliance with additional constraints. A novel, *FunState*-based approach to this so-called *scheduling* is explained.

For the above-mentioned applications, mainly *symbolic methods* based on *interval diagram techniques* are suggested in this work. These new techniques can be used for enhancing the efficiency of existing approaches. One of the main challenges facing design methods is to cope with extraordinarily large state spaces of the system which have to be traversed. During the last years, symbolic methods have attracted great attention in this regard. By means of implicit enumeration, they can achieve significant savings in terms of the computational resources required during the design process.

Kurzfassung

In einer Vielzahl technischer Geräte werden bereits heute Eingebettete Systeme eingesetzt, mit weiterhin steigender Tendenz. Diese Entwicklung wird sich aller Voraussicht nach insbesondere im Hinblick auf Anwendungen der Bereiche Konsum- und Fahrzeugelektronik, Telekommunikation und Medizintechnik fortsetzen. Eingebettete Systeme sind wie folgt gekennzeichnet:

- Sie bestehen zumeist sowohl aus Hardware- als auch aus Software-Komponenten und erfüllen Steuerungs-, Regelungs- oder Kommunikationsaufgaben innerhalb eines Gesamtsystems, mit dem sie über Netze, Sensoren und Aktoren verbunden sind.
- Im allgemeinen sind sie heterogene, häufig verteilte Systeme mit Echtzeitanforderungen, die transformativ und reaktiv arbeiten können.
- Sie bleiben dem Benutzer typischerweise verborgen, so daß dieser unbewußt mit ihnen interagiert.

Infolge des rasanten Anwachsens der Systemkomplexität sind die Herausforderungen beim Entwurf außerordentlich groß. Insbesondere sind die Entwurfsprobleme nur sehr aufwendig zu lösen und die Qualitätsansprüche hoch, denn Eingebettete Systeme setzen oft sicherheitskritische Funktionen um. Daher ist die Verwendung von Entwurfsautomatisierungswerkzeugen unverzichtbar geworden. Diese können – aufbauend auf einer formalen Darstellung – Systemeigenschaften überprüfen und die Korrektheit des Entwurfs zeigen. Danach läßt sich die Implementierung *synthetisieren*, indem automatisch Hardware- und Software-Beschreibungen generiert werden. Somit ist ein flexiblerer, kürzerer und kostengünstigerer Entwurfsprozeß möglich als ohne Automatisierung.

Die vorliegende Arbeit behandelt im aufgeführten Umfeld die folgenden Themenbereiche:

- Ein formales Darstellungsmodell namens *FunState* wird eingeführt, das insbesondere den Entwurf Eingebetteter Systeme unterstützt. *FunState* dient als interne Entwurfsdarstellung und läßt sich im Hinblick auf Analyse und Synthese zur Modellierung gemischter Hardware/Software-Systeme nutzen.

- Unter Verwendung von *FunState* wird ein Ansatz zur *Formalen Verifikation* vorgestellt, der die automatische Überprüfung gewünschter Systemeigenschaften ermöglicht. Im Gegensatz zu konventionellen Verifikationstechniken kann die Formale Verifikation die Korrektheit eines Systemmodells mathematisch beweisen.

- Um eine Implementierung zu generieren, muß die Ausführungsreihenfolge von funktionalen Bestandteilen des Systems festgelegt werden, häufig unter Berücksichtigung zusätzlicher Einschränkungen. Ein neuartiger, auf *FunState* basierender Ansatz zu diesem sogenannten *Scheduling* wird erläutert.

Für die aufgeführten Anwendungen werden in dieser Dissertation im wesentlichen *symbolische Methoden* in Form von *Intervalldiagramm-Techniken* vorgeschlagen. Dieses neue Verfahren läßt sich zur Effizienzsteigerung bestehender Ansätze verwenden. Denn eine der wesentlichen Herausforderungen an Entwurfsmethoden stellen die außerordentlich großen Zustandsräume des Systems dar, die traversiert werden müssen. Während der letzten Jahre haben diesbezüglich symbolische Methoden großes Aufsehen erregt. Mit Hilfe impliziter Aufzählung ermöglichen sie bedeutende Einsparungen bezüglich der während des Entwurfs benötigten Rechenleistung.

I would like to express my sincere gratitude to

- Prof. Dr. Lothar Thiele, for advising and inspiring me regarding research work in general and my thesis in particular, as well as for the support during my whole stay in Zurich, and
- Prof. Dr. Rolf Ernst, for agreeing to be the co-examiner of my thesis and for numerous valuable discussions.

Contents

1

Introduction

EMBEDDED systems are likely to emerge as one of the most promising branches of microelectronics during the early twenty-first century. They are expected to more or less visibly penetrate everyone's daily life. One the one hand, this may be in form of ubiquitous portable information appliances—such as cellular phones combined with personal digital assistants—as well as feature-rich automotive or household applications. Many of them will be augmented with high-speed wireless communications, Internet, multimedia, video, and cryptography capabilities. On the other hand, more conventional communication devices such as VoIP, MPEG, and ATM[1] appliances or industrial and traffic real-time control systems will steadily gain in importance.

The engineering challenges associated with the design of complex embedded systems continue to grow [FB99]. Due to the low-cost and consumer-oriented nature of this mass market, development and production of next-generation embedded systems will become more and more cost-driven. This is accompanied by shorter time-to-market constraints in a competitive environment. Moreover, because of their portability embedded systems often are highly integrated small-scale devices (frequently system-on-chip solutions) with very low power consumption—due to the lack of space for both cooling and power supply.

To summarize, features of growing complexity have to be implemented efficiently with regard to cost, size, and power consumption, while some overall system performance has to be ensured. Along with this, the design process is limited heavily in its length. Hence, the use of suitable tools for design automation is of crucial importance to faster de-

[1]VoIP: *voice over IP*, IP: *Internet protocol*, MPEG: *moving picture experts group*, ATM: *asynchronous transfer mode*

velopment time, higher product quality, and lower overall design costs. The aim of such tools is to support the design process by automating several tedious and error-prone tasks that could not or only very poorly be performed by hand. This often is carried out in an iterative process where starting with a first specification of the desired system and its requirements, numerous design steps are performed which, for instance, consist of one or more of the following:

- explore possible design alternatives by estimating and comparing their properties and apply design decision-making methods to ascertain feasible candidates,
- apply transformations to parts of the design which, e. g., decide on details of the final implementation, and hence refine the original specification,
- analyze the design with regard to the above objectives like cost, size, power consumption, and not least performance,
- check if the design meets originally specified constraints such as functional correctness, timing properties, or safety-critical characteristics, or
- generate source code or similar descriptions which can be used directly in order to obtain a prototype or an implementation.

These steps are mostly performed sequentially, in a varying order, in a mixed top-down bottom-up process. For instance, after analysis has shown that some performance requirement cannot be satisfied due to inaccurately predicted execution times, another design alternative is chosen for the evoking part or the entire design. The altered design transformed this way has to be analyzed (and possibly modified) again until it meets the required constraints. Some stages of this iterative process can be performed completely automatically. For example, in design exploration one or several feasible design alternatives are provided after having considered the whole design search space.

In the first phase, the designer ideally converts the system's requirements and constraints into a formal specification. The initial specification defines the interface between the system and its environment as well as the system's functionality [SRMB98]. Capturing the system requirements and describing them as well as the current design using a suitable design specification and representation model is crucially important for the entire dependent design process.

The description has to be unambiguous, complete, and precise enough to enable sufficiently exact results. At the same time, too low and detailed abstraction and granularity levels often are awkward to handle and lead to significant overhead and inefficient methods because large numbers of details have to be considered and the context cannot be grasped entirely. As a trade-off, details often are hidden by abstraction in early design phases and then disclosed or fixed step by step during the design cycle.

1.1 Different Views

Conventionally, specification domains have been divided into signal processing and control. Signal processing concentrates on the functional manipulation of data, while control focuses on influencing the behavior of a system in reaction to internal or external circumstances. In the following, both domains and their characteristics are described briefly.

1.1.1 Data Oriented Aspects

A natural paradigm for describing data oriented systems such as signal processing applications are dataflow related descriptions [Den80, WG82, Ack82]. Here, a system is divided into functional blocks which can execute whenever input data are available. Signal data are divided into *data items* or *tokens* which are treated as atomic and indivisible units corresponding to samples. Dataflow is represented by directed graphs where each node represents a block or function and each arc represents a signal or data path. The nodes or *actors* communicate via the arcs by receiving and sending data items in the direction specified. This is performed by consuming and producing tokens which in between are buffered in storage units represented by the arcs. These storage units are often conceptionally unbounded FIFO[2] queues. Functions perform conventional calculations when executed and thereby mapping input data into output data. This is done within discrete, atomic units of computation called *firings*. Dataflow allows for a natural representation of signal processing applications for concurrent implementation on parallel hardware.

A special well-known dataflow class is *synchronous dataflow* (SDF) [LM87a, LM87b] in which the number of tokens being consumed or produced by each node on each invocation is specified a priori, as shown in Figure 1 on the following page. Hence, when an actor is fired, it consumes a fixed number of input samples—depicted as small, solid circles—on each input arc, performs some computation on these data, and finally produces a fixed number of output tokens on each output arc. A node can be fired only if the given numbers of input tokens are available to be consumed. For instance, node v_1 in Figure 1 a) on the next page can be fired any time and then puts two tokens in queue q_1 and three tokens in q_2. Node v_3 can only be executed if q_1 contains at least one token and at the same time q_3 contains at least two tokens, and when fired, v_3 consumes the given numbers of tokens from the respective queues.

A restricted subclass of SDF graphs are *marked graphs* [CH71] where the above, so-called *consumption* and *production rates* are equal to 1 which heavily restricts the sample rates in the system. *Computation graphs* [KM66], in which the numbers of tokens required for function execution are independent of the consumption rates, are slightly more elaborate than SDF graphs due to this notion of threshold.

[2]FIFO: *first-in, first-out*

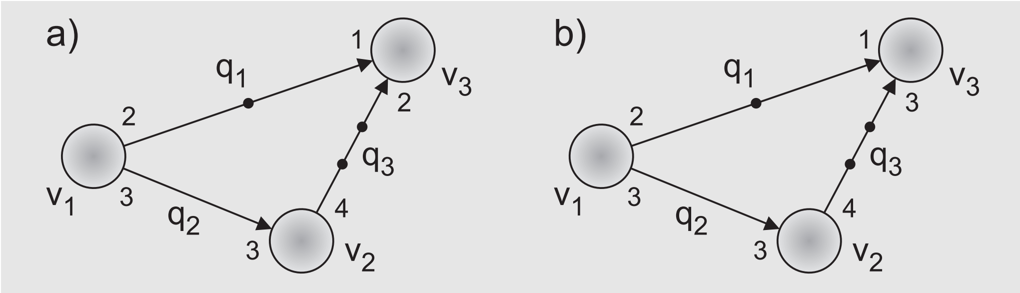

Figure 1. Schedulable and non-schedulable synchronous dataflow graph.

When an SDF graph is to be implemented on a signal processing system (consisting of one or several processors), the firing of functions has to be controlled such that they are executed only when data is available. This process called *scheduling* can be done either *dynamically*, *statically*, or using hybrid techniques. Dynamic scheduling means that during run time the readiness of actors for being fired is checked. Then, enabled functions are assigned to processors and run there. However, fully dynamic schedulers often exhibit significant run-time overhead due to many dynamic tasks to be executed.

In contrast, SDF graphs, for instance, can be scheduled statically at compile time, avoiding this dynamic control overhead. The order in which nodes are executed can be determined a priori by a compiler that then can construct sequential code for each processor. The scheduling process hence consists of sequentializing function executions. The resulting firing sequence represents the *schedule* which also has significant impact on the FIFO queues buffering data items during execution. Namely, these buffers are required to be large enough to store any number of tokens occuring during execution of the schedule, and this buffer size heavily depends on the schedule chosen. As the memory used in a physical implementation for implementing buffers is limited and expensive, buffer sizes should be kept small.

Two necessary requirements are imposed on a schedule to be *feasible*: The schedule has to be *deadlock-free*, and it has to guarantee bounded *queue contents*—the numbers of tokens in queues. *Deadlock* in this case refers to an undesirable situation in which a node intended by the schedule for being fired cannot be executed at that time due to the lack of input tokens. The queue contents have to be bounded regarding the memory required for the above-mentioned reason. A schedule providing bounded queue contents is called a *bounded* schedule. Among all feasible schedules the preferred ones are those which optimize some criterion, for example, those that have the minimum sum of all the queue lengths.

Regarding Figure 1 a), a feasible schedule is described by the firing sequence $v_3\, v_1\, v_2\, v_3 \;\; v_3\, v_1\, v_2\, v_3 \ldots$ which is repeated periodically. This schedule is bounded and deadlock-free at the same time as the reader can easily confirm.

In Figure 1 b), the situation is different. For instance, the schedule $v_1\,v_2\,v_3\ \ v_1\,v_2\,v_3\ldots$ is deadlock-free but not bounded as both q_1 and q_3 overflow. On the other hand, $v_3\,v_1\,v_2\,v_3\ \ v_3\,v_1\,v_2\,v_3\ldots$ runs into a deadlock for the fourth invocation of v_3 with regard to q_3. Indeed, for this SDF graph no feasible schedule exists which can easily be verified using *balance equation* methods [LM87a, LM87b]. The reason for this is a sample rate inconsistency which makes any schedule for this graph result either in deadlock or unbounded buffer sizes.

1.1.2 Control Oriented Aspects

Finite state machines (FSMs) have long been used for describing and analyzing aspects of complex control oriented systems. FSMs are often described using a directed graph called *state transition diagram*, as shown in Figure 2. At any time, the state machine is in exactly one of its two *states* represented by nodes of the state transition diagram. The arcs between nodes represent *transitions* from one state to another (or itself) which can be taken only if some annotated condition is satisfied. For instance, the transition from the left-hand to the right-hand state is allowed only if some guard b is satisfied. If the transition is taken, then some action v is performed. Transition conditions can, for instance, consist of predicates on input signal values or events used for communication, while actions can represent similar output to be emitted. The left-hand state in Figure 2 is marked as the initial state of the FSM by a small transition arc.

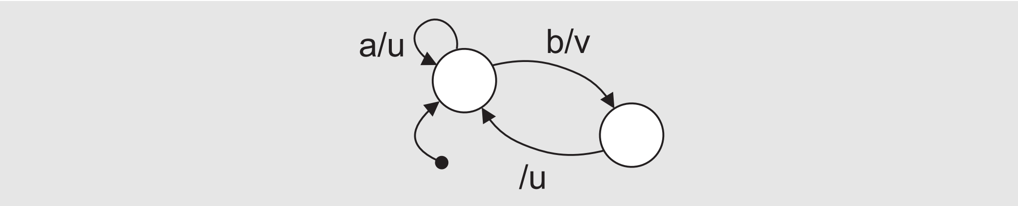

Figure 2. Finite state machine.

In order to cope with complex control oriented systems and for more expressive power, FSMs are often enhanced with communication as well as hierarchical and concurrent extensions, as described in later chapters. Then, conditions and actions even can be used for communication between concurrent state machines. Due to their finite nature, FSMs and their numerous variants often provide better analysis and synthesis means than comparable models such as conventional sequential programming. For instance, simply enumerating the set of states reachable from the initial state can guarantee that some dangerous system state cannot occur. Nevertheless, for complex real-world systems often more elaborate techniques for formal verification have to be employed for such tasks due to the extent of the problem.

1.2 Mixed Control and Data Oriented Systems

Embedded systems typically consist of both control and data oriented parts, interacting with each other. In general, the control portion influences the operation of the computationally intensive dataflow part. In turn, control often depends on feedback from the data oriented part as well as on external circumstances. Hence, both the system aspects are closely interwoven with each other. During the last years, this has been recognized by many researchers [GSM97, GLL99, TSZ$^+$99]. For instance, Grötker *et al.* [GSM97] state that

> "For a smooth design flow with a maximum degree of modularity it is of crucial importance to be able to model the complete functionality of the system, containing both control and data flow, within one single design environment."

Hence, various *models of computation* arose, integrating the specification of control and data oriented system aspects. The term "model of computation" comprises the behavior of system components (rules of execution of components) combined with the semantics of the composition (rules of interaction between components). In this work, the term "*process networks*" refers to both simple and more elaborate models of computation consisting of processes, which may have a complex internal state, and buffers for communication. In this sense, most of the abovementioned integrating models are process networks. Process networks combine data and control oriented system aspects as explained above. More precisely, they merge dataflow and finite state machines.

Since the synchronous dataflow model is restricted enough, tasks such as determining a feasible static schedule or the verification of properties are well investigated and may be handled efficiently. The situation is akin regarding computation graphs. While many dataflow models are sufficiently analyzable by means of balance equation methods, such methods fail in more powerful process network models due to their complex internal state. Dedicated methods required to remedy this drawback are the principal focus of this monograph.

Essentially, two topics related to embedded systems design are addressed. On the one hand, the formal verification of several process network models of computation is investigated which leads to promising results with regard to the computational efficiency. On the other hand, an approach to the scheduling of process networks is described which is able to deal with mixed data/control flow specifications and takes into account different mechanisms of non-determinism as occurring in the design of embedded systems. In the following sections both fields are briefly introduced. They have the fact in common that the use of symbolic techniques for state space traversal is a feasible solution to certain problems concerning the computational complexity—which is sketched later on.

1.2.1 Formal Verification Issues

Typical questions to be answered by means of formal verification of process networks are concerned with the absence of deadlocks or the boundedness of the required memory. Questions like "may process P_1 and P_2 block each other?" or "may P_1 and P_2 collide while accessing a shared resource?" are likewise of interest. 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 compliance with memory limitations without restricting the system model [Par95]. The process models in addition may be extended to describe one or several dynamic or hybrid scheduling policies. Their behavior then is verified together with the system model. Thus, common properties such as the correctness of the schedule may be affirmed or artificial deadlocks [Par95] may be detected.

A simple example process network from [Par95] is shown in Figure 3. A, B, C, and D represent processes, while a, b, c, d, and e are unbounded FIFO queues. The internal process states are represented by program code in this example. The command $get(a)$ reads a data item from queue a and returns the item value, while $put(x, a)$ writes a token of value x into queue a. The network follows a blocking read semantics, i. e., a process is suspended when attempting to get data from an empty input channel. A process having sufficient data on its input channels is called enabled.

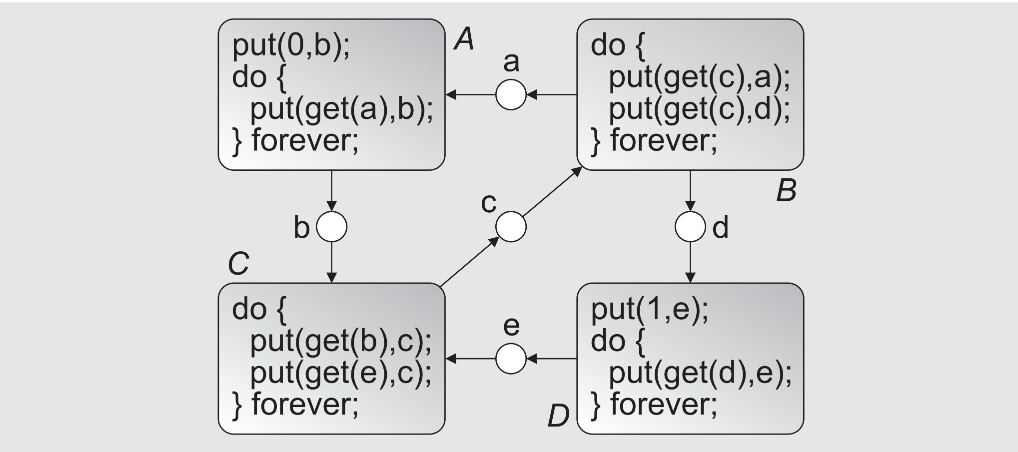

Figure 3. Simple process network.

Formal verification is able to prove that for any execution 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 for the development of scheduling policies. For instance, a plain priority-based scheduling policy can be modeled in combination with the network to control the execution order of the network's processes. Assume that part of this scheduling policy is the rule that process B is executed only if A is not enabled, otherwise A is executed. The consequence of this rule is that B could be blocked forever if A may always be enabled. An important question to be answered by formal verification is whether or not this case may occur during any execution of the process network controlled by the given scheduler.

1.2.2 Scheduling Issues

Besides conventional scheduling constraints such as data dependencies or resource constraints, even more complex constraints may be imposed to influence a schedule being developed. For instance, interface protocols between several components or various external constraints may have to be taken into account. Furthermore, the desired scheduling policy may be given partially, resulting in an incomplete specification which has to be obeyed during schedule development.

Consider a constellation of components mapped onto different implementational units and communicating via queues in a distributed, parallel setting. The components have both data and control flow properties. Non-determinism may exist resulting from incomplete specifications or data dependencies resolved only at run time. The methods presented in this work deal with the problem of finding a feasible schedule, for instance, for the components mapped onto one implementational unit respecting constraints given by other components.

1.2.3 FunState

As mentioned above, common forms of representation of mixed control/data oriented systems have gained in importance, mainly in the fields of embedded systems and communication electronics. Therefore, the *FunState* (*functions driven by state machines*) formalism has been developed which combines dataflow properties with finite state machine behavior [TSZ+99]. *FunState* can be used for modeling mixed hardware/software systems for analysis and synthesis purposes. It serves as an internal representation in the design phase. *FunState* has been designed to support formal verification and the representation of implementation decisions, such as scheduling strategies. Using a mixture of functional programming and state machines, the *FunState* model of computation enables the representation of different types of scheduling mechanisms. It combines the notion of dataflow with controlled, event-driven process activation. The term process networks as introduced above also comprises the *FunState* model.

In Figure 4 on the next page, the *FunState* model corresponding to the process network in Figure 3 on the preceding page is shown. It consists of four components; each of them has two parts: an upper, data oriented part—depicting dataflow by functional units (rectangles) and FIFO queues (circles)—and a lower, control oriented part—described by a finite state machine. Each component corresponds to the respective process of the process network.

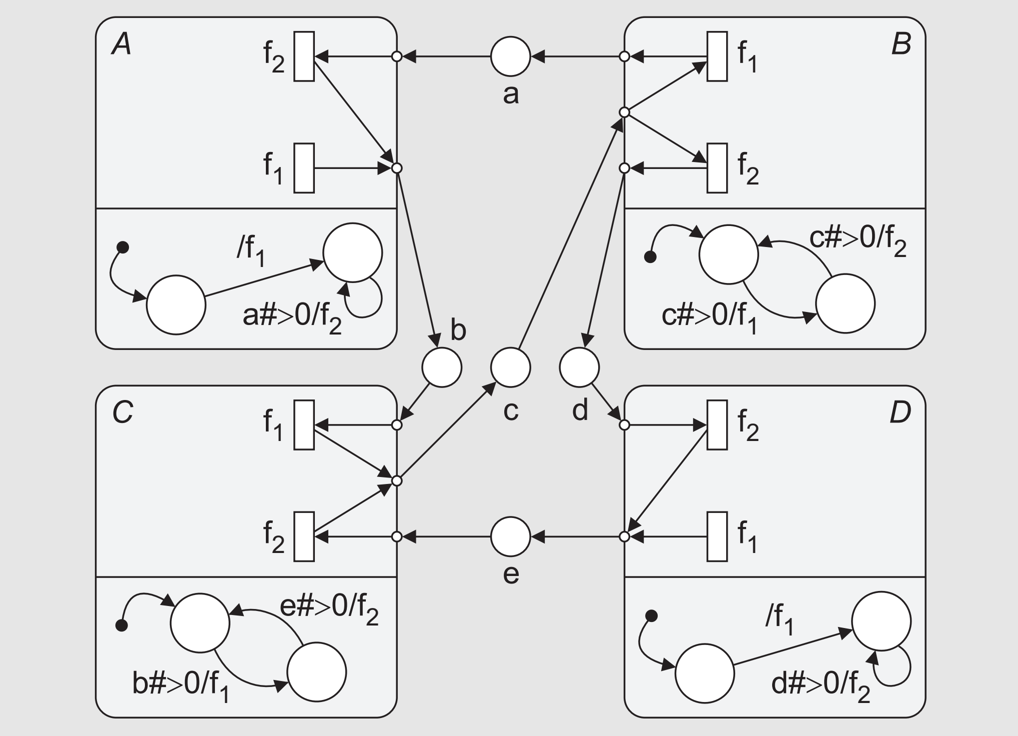

Figure 4. *FunState* model corresponding to process network in Figure 3 on page 7.

The queues in the data oriented part store data items depicted by tokens, while the functional units perform computations on the data. The functions have consumption and production rates of 1 in this example. One major peculiarity of *FunState* compared with related models of computation is that the functions are passive. The execution of the functions of each component is controlled by the corresponding state machine. The concurrent execution of state machines of different components is asynchronous and interleaved in this context.

The labels of the state machine transitions indicate combinations of a condition and an action (e. g., "$c\# > 0/f_1$" in component B), meaning that the respective transition, and thereby the action, may be executed only if the condition ($c\# > 0$) is satisfied (i. e., if the queue labeled with c contains at least one data item). The purpose of such a predicate mostly is to ensure the presence of the input data tokens needed for an execution of the corresponding function. If the above transition is taken, function f_1 is executed, consumes one token from queue c and produces one token for queue a.

Consider process C of the process network in Figure 3 on page 7 and the corresponding component C of the *FunState* model in Figure 4. Function f_1 of the component corresponds to the first combination of get and put in the process, while f_2 imitates the second combination. Each combination consumes one token from either queue b or queue e and then produces one token in queue c. Exactly this is performed by the respective functions. Both combinations of get and put are executed within a never-ending loop which alternately executes one of them. This control structure is represented by the state machine part of the *FunState* component which toggles between its both states. The fact that a state machine transition can only be taken if the respective queue contains enough data items corresponds to the blocking read semantics of the process network.

1.3 The Symbolic Approach

During the last years, the *symbolic model checking* approach, see, e. g., [BCM+92, McM93, BCL+94, CGP00], has been applied to many areas of formal system verification in computer engineering and related research fields and even has been able to enter the area of industrial applications. For real-world applications, the major problem of many formal verification approaches is the huge number of system states to be considered. This fact usually is described aptly by the term "state explosion" problem. In contrast to simulation, formal verification takes into account all possible execution traces without exception. Since enumerative techniques often show not to be feasible for this task, symbolic methods have been developed which perform an implicit enumeration to reduce the required computational efforts.

To this end, the original symbolic model checking approach applied to conventional finite state systems makes use of *binary decision diagrams* (BDDs) [Ake78, Bry86] which are an efficient representation of Boolean functions and enable their very fast manipulation. But when applying this technique to process networks or related models of computation—i. e., models with unbounded system states such as many high-level and dataflow oriented models—, several obstacles prevent its practical usefulness in this area.

In order to overcome some limitations of conventional symbolic model checking of process networks and related models, an approach has been developed that uses *interval decision diagrams* (IDDs) combined with *interval mapping diagrams* (IMDs) and thus is able to remedy the described shortcomings of traditional approaches. Formal verification of process networks using these *interval diagram techniques* has been introduced in [ST98]. Fundamentally, the approach is based on a more reasonable way of describing certain forms of transition relations.

Interval diagram techniques are well suited to handle the combination of internal process states and buffer contents, both representing the entire system state. In addition to the symbolic formal verification of process networks, interval diagrams have applications in the fields of real-time symbolic model checking and symbolic scheduling, where similar problems have to be dealt with.

Besides several applications, this work describes the formalism of interval decision diagrams, an efficient representation of discrete-valued functions, and the methods and techniques necessary for applying this form of function description to symbolic formal verification and symbolic scheduling by means of interval diagram techniques. IDDs and IMDs are dedicated to state set and transition relation descriptions of many, especially data oriented models of computation.

Mainly, interval diagram techniques have the above-mentioned fields of application in the areas of real-time computing, embedded, and communication systems which have been summarized in [ST99b]. These fields are of analytical or constructive nature, respectively.

An approach to symbolic model checking of process networks has been developed based on interval diagram techniques. Several diverse system models have been investigated [ST98, ST99a] which led to promising results concerning the required memory as well as the computation time. For both criteria, IDDs and IMDs turned out to be superior to the conventional approach using BDDs. Further, the formal verification of *timed automata* [AD94] has been performed by reachability analysis on discrete time [Str99]. The interval diagram techniques approach has been compared with another, similar verification technique based on BDDs yielding significant computation time reductions.

Besides formal verification, interval diagram techniques have been applied to symbolic scheduling of process network models in the embedded systems domain. Due to similar transition behaviors, the advantages of symbolic formal verification approaches based on interval diagram techniques may be transferred to the area of symbolic scheduling of certain process network models of computation. A symbolic scheduling method for heterogeneous embedded systems has been developed [STZ$^+$99].

1.4 Overview

This monograph is organized as follows. Chapter 2 presents several models of computation on which the considered formal verification and scheduling techniques are based. This comprises the *FunState* and timed automata models. Chapter 3 deals with symbolic methods for state space traversal. After motivating and explaining their use, interval diagram techniques are introduced. In Chapter 4, the symbolic verification of process networks and timed automata based on interval diagrams is presented and justified with some experimental results. The symbolic scheduling approach of process networks is subject of Chapter 5, with its suitability shown with some examples. Finally, Chapter 6 concludes this dissertation and gives some prospects for the future.

2

Models of Computation

ONE of the major sources of complexity in the design of embedded systems is related to their heterogeneity. On the one hand, the specification of the functional and timing behavior necessitates a mixture of different basic models of computation and communication which come from transformative or reactive domains. On the other hand, the heterogeneity in the implementation is increasing steadily. This not only concerns the functional units which may be implemented in form of dedicated or programmable hardware, microcontrollers, domain specific or even general purpose processors. In addition, these units communicate with each other via different media, e. g., busses, memories, networks, and by using many different synchronization mechanisms.

This heterogeneity caused a broad range of models of computation to be developed with specific application domains. Only few examples—on which formal verification and scheduling as described in following chapters are based—are presented in this chapter. Especially *FunState* and timed automata are introduced in detail in Sections 2.2 and 2.3.

2.1 Process Networks

In this work, the term process networks is used for describing such models of computation characterized as containing concurrent processes—which may have complex internal states—and buffers for communication. By means of tokens, data items can be written into and read from buffers to exchange data between processes. Buffers often are unidirectional and have a FIFO queue behavior, but this is no prerequisite. A maximum number of tokens contained may be assigned to a buffer which is called its *capacity*. Buffers occur in almost every communication setting between processes. For instance, in real-time operating systems they are often called mailboxes.

Many different classes of process networks exist and are employed in diverse application domains. An example is the *FunState* model of computation presented later on. Many other models of computation, e. g., *dataflow process networks* [LP95], *computation graphs* [KM66], and *synchronous dataflow* (SDF) [LM87a, LM87b], turn out to be special cases of Kahn's process networks [Kah74, KM77] and of the *FunState* model of computation.

2.1.1 Petri Nets

In the sense of the above understanding of process networks, Petri nets are representatives of this class as well. In the context of this monograph, a Petri net is defined similarly to [Mur89] as a 6-tuple $G = (\mathbf{P}, \mathbf{T}, \mathbf{F}, K, W, M_0)$ with $\mathbf{P} \cap \mathbf{T} = \emptyset$ and $\mathbf{F} \subseteq (\mathbf{P} \times \mathbf{T}) \cup (\mathbf{T} \times \mathbf{P})$, where

- $\mathbf{P}$ and $\mathbf{T}$ are the sets of places and transitions, respectively,
- $\mathbf{F}$ is the flow relation including the graph arcs connecting places and transitions,
- $K : \mathbf{P} \to \mathbb{Z}_{>0} \cup \{\infty\}$ denotes the place capacities which may be positive integer numbers or infinite,
- $W : \mathbf{F} \to \mathbb{Z}_{>0}$ describes the weights of the arcs, i. e., the consumption and production rates, and
- $M_0 : \mathbf{P} \to \mathbb{Z}_{\geq 0}$ is the initial marking of G.

The marking M of a Petri net denotes the actual number of tokens in each place. The behavior of a Petri net is defined as usual. A transition $t \in \mathbf{T}$ is sensitive for firing if all its incoming places contain at least as much tokens as the respective arc weights and all its outgoing places are able to absorb at least as much tokens as the respective arc weights without exceeding their place capacity. According to the above description, only uncolored Petri nets are considered. Hence, the tokens have no values and thus are not distinguishable ("black" token model).

In Figure 5 on the facing page, an example Petri net is shown. Places are depicted by circles, while squares denote transitions. Some of the arcs are annotated with their weights. Arc weights of 1 are omitted in the figure.

2.2 FunState

FunState (*functions driven by state machines*) [TSZ+99] unifies many different well-known models of computation, supports stepwise refinement

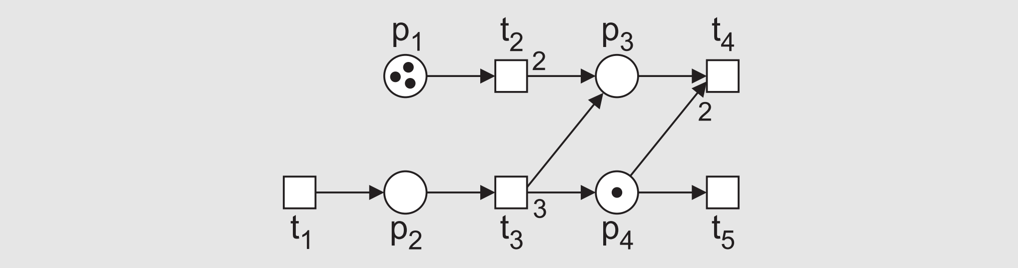

Figure 5. Simple Petri net example.

and hierarchy, and is suited to represent many different synchronization, communication, and scheduling policies. The *FunState* formalism combines dataflow properties with finite state machine behavior and enables the explicit representation of non-determinism and scheduling using a combination of functional units and state machines. Non-determinism is important for high-level models of computation both to model non-determinism in the environment and also to abstract away implementation details. Hence, making low-level decisions can be postponed until high-level design is completed and verified.

FunState has been designed to serve as an internal representation in the design phase of embedded systems. The role of such an internal model in a multi-language setting is shown in Figure 6 on the next page. A specification of a system consists of different input formalisms. These different parts may be modeled and optimized independently. Then the information useful for methods such as allocation of resources, partitioning the design, scheduling, and verification are estimated or extracted and mapped onto internal representations which describe properties of the subsystems and their coordination (synchronization and communication). This approach supports incremental implementation and verification. Another application of a mixed representation is the inclusion of third party, IP[1], or legacy system parts where control information is incomplete.

There may be different internal models for different tasks being performed during system analysis and design. Methods such as scheduling, abstraction, and verification work on these internal representations and eventually refine them by adding components and reducing non-determinism.

This section deals with the following topics.

- The *FunState* representation is defined which serves as an internal representation of heterogeneous embedded systems for the purpose of scheduling and formal verification. Extensions are provided which enable hierarchical representations as well as timed execution behaviors and support abstraction mechanisms.

[1]IP: *intellectual property*

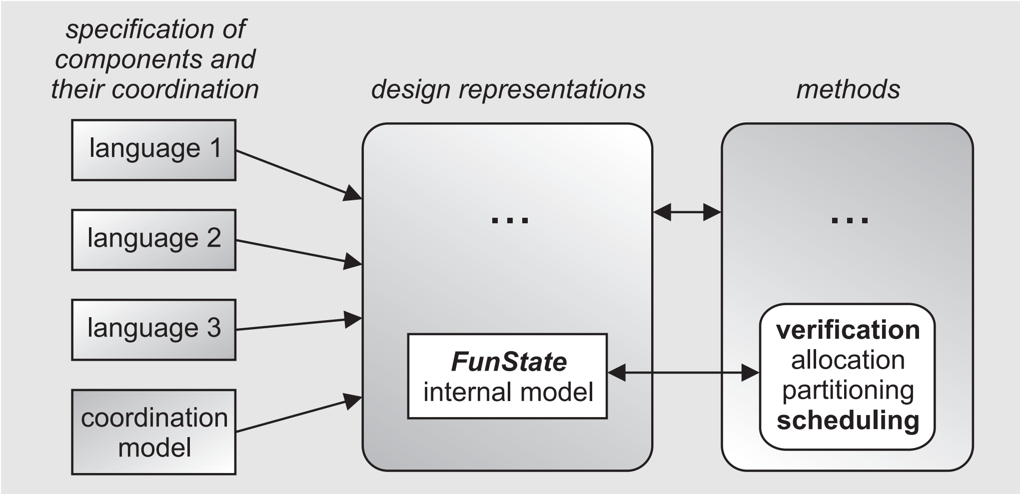

Figure 6. Role of *FunState* in a design process

- Since the *FunState* model explicitly separates data and control flow, properties of many different models of computation can be represented, such as *marked graphs* [CH71], synchronous dataflow [LM87a, LM87b], *cyclo-static dataflow* [BWE+93, EBLP94], *communicating finite state machines* as well as Petri nets. In contrast to other approaches, constraints and refinements as occurring in a typical design process can be represented directly in the model.

- Methods suited to *FunState* such as scheduling and formal verification are based on the representation of the model's state transition behavior in form of a periodic graph. These graphs are theoretically well investigated. The simplicity of the underlying semantics distinguishes the presented representation from other approaches.

2.2.1 Related Work

In many applications such as embedded systems, the transformative domain (data processing, stream processing) and the reactive domain (reaction to discrete events, control flow) are tightly interwoven. Application examples include mode and parameter control of dataflow processing systems, system configuration and initialization, e. g., in packet-based transmission systems [GSM97], wireless modems [CKL95], etc.

It is not possible here to give an overview of all specification models which have been proposed in this area. Many of them are covered in later sections when relating *FunState* to other models of computation. An overview and classification of different models of computation including discrete-event, reactive, and dataflow models is given in [LS98].

The SPI (*system property intervals*) methodology [ZER+98, ZRE+98] has been designed to deal with the modeling problem of complex embedded systems for the purpose of scheduling. Analogously to *FunState*, SPI is a formal design representation internal to a design system. It combines the representation of communicating processes with correlated operation modes, the representation of non-determinate behavior, different communication mechanisms such as queues and registers, and scheduling constraints.

While SPI is intended to capture the semantics of multiple input languages with different semantics, *FunState* has been defined to support verification and the representation of implementation decisions, such as scheduling strategies. While *FunState* allows to explicitly model control, in SPI, control is always linked to data tokens. So, one design scenario would be to start with a SPI notation capturing the design intent and then gradually extend the SPI processes for verification and implementation as shown in Figure 6 on the facing page.

In the SPI model, the control information is communicated using data tokens. Two similar approaches are Huss' codesign model CDM [BHK99] and Eles' *conditional process graph* [EKP+98]. Many other research groups independently proposed models that separate data and control flow. These are, for example, SDL[2] [SSR89], *codesign finite state machines* [BCG+97], *program state machines* [VNG95], and the combination of synchronous dataflow with finite state machines [GSM97] as in the *process coordination calculus* (PCC) [PMRM94]. Most of these approaches have limited composability as data and control flow cannot be mixed arbitrarily in the hierarchical levels.

Also in this area, graphical formalisms based on extensions of classical FSMs like hierarchical and concurrent FSMs such as *statecharts* introduced by Harel [Har87] have been developed, with many variants [Mar89, vdB94]. In the STATEMATE® MAGNUM™ implementation of statecharts by I-Logix, Inc., the dataflow aspect is covered in a separate domain called activity-chart. In *charts [CKL95, GLL99], unlike statecharts and other concurrent hierarchical FSMs, no model of concurrency is defined a priori. Instead, the goal is to show how to embed FSMs within a variety of concurrency models, i. e., dataflow models, discrete-event models, and the synchronous/reactive model.

Whereas these authors favor the systematic combination of disjoint semantics often combined with abstract graphical models (block diagrams), e. g., [CKL95], others seek for a consistent semantics for specification of the complete system, for example, the COSYMA system [EHB93] and the OLYMPUS system [DKMT90]. While in the unified approach, the major problems deal with the challenge of how to extract portions of a design in order to be able to apply efficient analysis and synthesis techniques to portions of the specification, a major problem in the mixed approach lies in finding clean ways to combine diverse models of computation, at various levels of abstraction.

Complementary to the above approaches, the *FunState* internal model attempts to reduce the design complexity by representing only those characteristics of a heterogeneous input specification which are relevant

[2]SDL: *specification and description language*

to certain design methods, in particular scheduling and formal verification. Therefore, the primary purpose is not to provide a unifying algorithm specification.

Besides the usual requirements for specification models such as composability, hierarchical structure, well defined semantics and adaptation to the heterogeneity present in the application domain, some further properties are required.

- The properties of different specification models (computer languages, block diagrams) relevant to certain design methods should be representable in the internal model.
- The internal model should support abstraction mechanisms as necessary for the design of complex systems.
- The internal model should support refinement such that results in the design process can be incorporated into the model, e. g., scheduling decisions reducing the degree of non-determinism.

2.2.2 The Basic FunState Model

At first, the basic non-hierarchical and untimed *FunState* model is explained. The activation of functions in a network is controlled by a finite state machine, similarly to the semantics of activity-charts in statecharts implementations, see [HN96]. In contrast to dataflow models of computation, functions (or actors) are not autonomous.

Def. 2.2.1: *The basic untimed FunState component consists of a network N and a finite state machine M. The network $N = (\mathbf{F}, \mathbf{S}, \mathbf{E})$ itself contains a set of storage units $s \in \mathbf{S}$, a set of functions $f \in \mathbf{F}$, and a set of directed edges $e \in \mathbf{E}$ where $\mathbf{E} \subseteq (\mathbf{F} \times \mathbf{S}) \cup (\mathbf{S} \times \mathbf{F})$.*

Data is represented by *valued tokens*. Storage units and functions form a bipartite graph. In other words, there are no edges connecting two storage units or two functions.

Figure 7 on the next page shows an example of a simple *FunState* model. The upper part represents the network N containing storage units q_1, q_2, q_3, and q_4 with 1, 2, 0, and 3 tokens, respectively, and functions f_1, f_2, and f_3. The lower part contains a finite state machine, in this example with just one state and three transition edges. Details concerning the behavior of the *FunState* model are described below.

2.2.2.1 Elements of the Network

- **Storage units**
  For the sake of simplicity, only two sorts of storage elements are introduced here, namely queues and registers. The actual access functionality and the available query functions on storage unit types can be defined individually for each type. Note that only examples are given here.

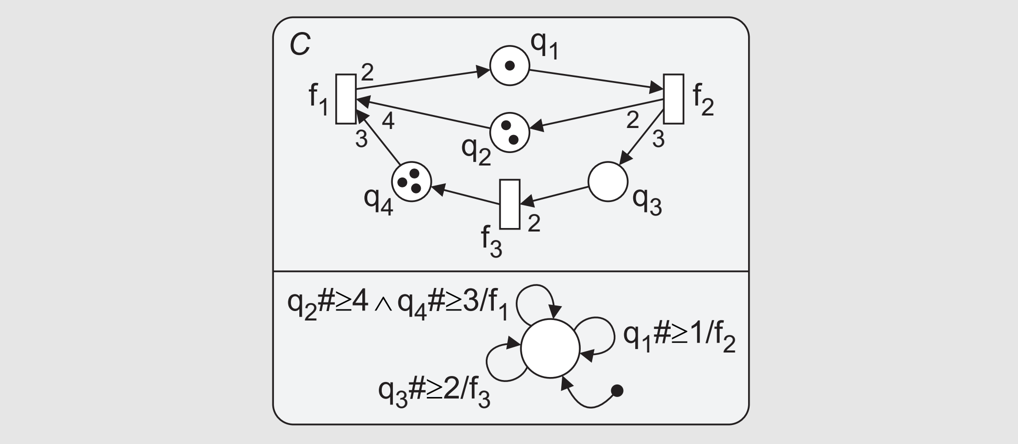

Figure 7. Example of a simple *FunState* model.

Queues Queues have FIFO behavior and unbounded length. They store tokens which are added (removed) via incoming (outgoing) edges. The tokens represent data flowing through the network. The numbers of tokens $q\# \in \mathbb{Z}_{\geq 0}$ in queues $q \in \mathbf{S}$ are part of the system state. $q\#0 \in \mathbb{Z}_{\geq 0}$ denotes the initial number of tokens, see Figure 8. Depending on the abstraction level, colored tokens can be dealt with, i. e., tokens with values associated. In this case, $q\$1$, $q\$2$, …, $q\$k$ with $k = q\#$ denote the values of the first, second, …, kth token in queue q, respectively. The assignment of initial values to tokens is not considered here.

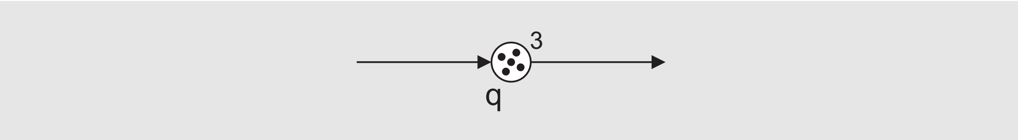

Figure 8. Example of a queue q with $q\#0 = 3$, i. e., 3 initial tokens. The current number of tokens is $q\# = 5$.

Registers Registers are linear arrays of limited length n of pairs of addresses and values. In contrast to tokens in a queue, the number of values in a register is constant. These values $r\$1$, $r\$2$, …, $r\$n$ of a register r can be replaced by tokens via incoming edges or read nondestructively via outgoing edges, see Figure 9 on the next page. In comparison with queues, registers do not impose a partial ordering on function evaluations. Registers are modeled in order to represent the flow of information, e. g., for estimating the necessary communication bandwidth and imposing timing constraints.

- **Functions**

The function objects $f \in \mathbf{F}$ of a *FunState* model are uniquely named and operate on tokens or values when firing. Inputs and outputs of functions have variables $c_i \in \mathbb{Z}_{\geq 0}$ and $p_i \in \mathbb{Z}_{\geq 0}$ associated which denote the num-

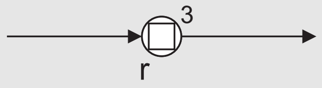

Figure 9. Example of a register r of size $n = 3$.

ber of consumed tokens (read values) and the number of produced tokens (replaced values), respectively. The variables represent expressions which evaluate to constants or random processes. If required, additional constraints, for example, intervals, involving these variables may restrict the numbers of consumed or produced tokens. In case of firing, the function f shown in Figure 10 consumes c tokens from queue q_1 and reads 3 values from register r_1. It adds to q_2 some non-deterministically chosen number of tokens in the interval $[1, 4]$ and replaces p values in r_2.

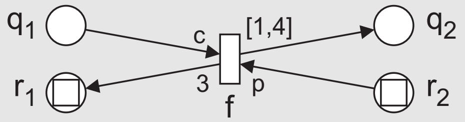

Figure 10. Example of a function f.

2.2.2.2 State Machine

There are many different possibilities of specifying the finite state machine M which controls the activation of embedded components or functions. In order to facilitate analysis, scheduling, and the concept of hierarchy, a synchronous/reactive model is chosen. In particular, the model is akin to the graphical language Argos [Mar89] developed at IMAG (Grenoble). It resembles the statecharts formalism by Harel [Har87, HN96] but resolves circular dependencies using fixed point semantics.

Transitions are labeled with conditions and actions. Conditions are predicates on storage units $s \in \mathbf{S}$ in the network. These predicates very often only concern the number of tokens in a queue, e. g., $q\# \geq v$ for some integer variable v. Again, this variable may represent a determinate value or a random process, possibly constrained. A transition is enabled if the corresponding predicate is *true*. The action consists of a set of the names of those functions which are activated when the transition is taken.

Figure 11 on the next page shows the example of a simple automaton. The transition t is taken if the automaton is in its initial state, if there are at least 3 tokens in queue q, and if the value of the second token is less than 1.5. At the same time, the functions named f_1 and f_2 are activated.

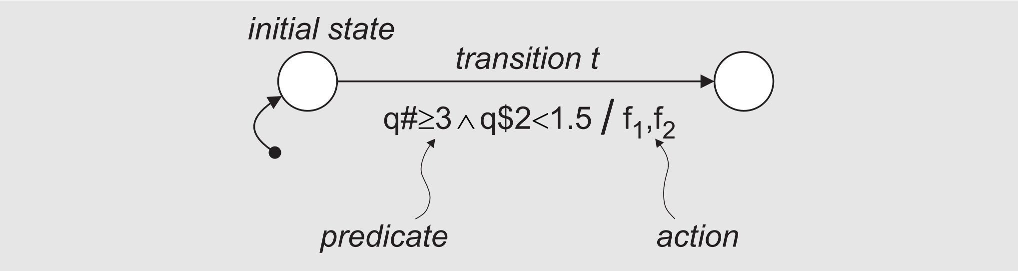

Figure 11. Part of a simple automaton M of some component.

2.2.2.3 Operational Semantics of the Flat Model

Until here, the interaction between the state machine and the network has not been considered. The basic *FunState* component is executed following the steps described below, in the given order.

1. **Initialization**
   The current state of the state machine is set to its initial state. All queues and registers are filled with initial tokens and values according to the initial marking of the network.

2. **Predicate evaluation**
   All predicates on storage units occuring in conditions of transitions originating from the currently active state of the state machine are evaluated.

3. **Check for possible progress**
   If there is no enabled transition, i. e., no one with a satisfied predicate, the execution is stopped.

4. **State machine reaction**
   One non-deterministically chosen enabled state transition, of which the source is the current state, is taken. All functions in the corresponding action set are activated.

5. **Function activation**
   All activated functions are fired in non-determinate order. A fired function removes tokens (reads values) from its input storage units and adds tokens (writes values) to its output storage units[3]. The execution continues at "Predicate evaluation" (step 2).

The above execution stages are exemplified with the *FunState* model in Figure 7 on page 19. Assume that the figure shows the initial state of the model. Hence, for initialization (step 1) the queue contents $(q_1, q_2, q_3, q_4)^T$ are set to $(1, 2, 0, 3)^T$. The current state is set to the state machine's only state. The result of predicate evaluation (step 2) is that

[3]It may happen that a function tries to remove tokens from an empty queue. There are many possibilities to deal with such a case. One may even apply the verification methods described later on in order to statically prove that this cannot happen at run time.

only predicate $q_1\# \geq 1$ is satisfied. Thus, progress is possible (step 3) and execution continues.

Since only one transition starting in the current state is enabled, the state machine performs this transition (resulting in the same current state as before) and activates function f_2 (step 4). The only activated function f_2 is fired (step 5) and changes the queue contents into $(0, 4, 3, 3)^T$. Hence, during the next execution cycle, starting with step 2, either f_1 or f_3 will be executed.

While function executions in this setting are atomic actions, this is not the case for the timed *FunState* extension introduced in Section 2.2.3.4.

2.2.3 Model Extensions

2.2.3.1 Extensions of the State Machine

Several extensions of the above-described simple state machine lead to a model similar to Argos [Mar89]. In particular, the following concepts are used.

- **Hierarchy**
  States can be hierarchical, i. e., may contain other automata. In case of an XOR composition, the outer state is interpreted as being in one of its child states. If the superordinate state is refined using AND composition, it is interpreted as being in all of its child states at the same time (concurrency).

- **Events**
  Events can be part of predicates and action sets. An event has the value *true* if it is in the action set of a taken transition and *false* otherwise. Events are not visible outside a component.

The communication via events is performed synchronously. An activated event possibly influences the predicate evaluation of the very same time instant. A state machine involving AND and XOR composition can easily be flattened, i. e., transformed into a simple state machine consisting of states and transitions annotated with conditions and actions.

These straightforward extensions are exemplified with the following simple exception handler shown in Figure 12 on the next page. If one of the predicates p_1 and p_2 in the left-hand two partial automata is *true*, then state X is left and one of the states Y and Z is entered to manage the exception. The right-hand partial automaton resolves simultaneous exceptions via some priority rule. The communication uses the events u, v, s, and t. In particular, if only p_1 or both predicates are *true*, then state Z is entered. If only p_2 is *true*, then a transition to state Y occurs. All this happens within one time instant.

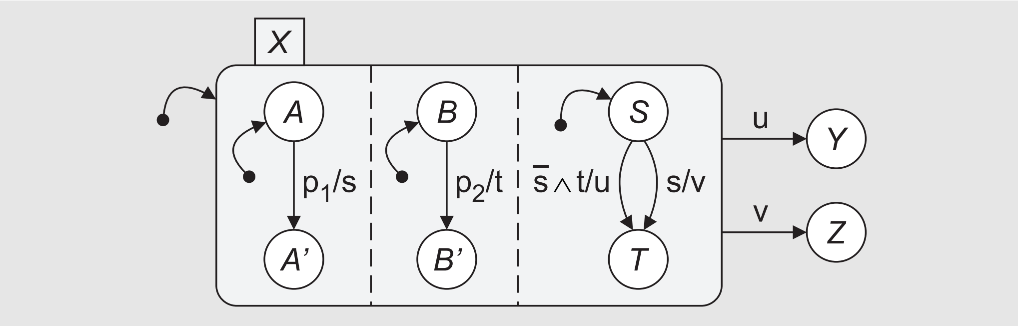

Figure 12. Example of a state automaton with AND and XOR states.

2.2.3.2 Hierarchical *FunState* Network

The basic element of the hierarchical network of the *FunState* formalism is the component. Each component contains a network N and a state machine M as defined for the basic model. In addition to this, N can also contain embedded components and ports representing interfaces through which the component exchanges tokens with the component into which it is embedded. A hierarchical *FunState* model has exactly one top-level component. This top-level component has no interfaces.

Def. 2.2.2: *The network* $N = (\mathbf{F}, \mathbf{S}, \mathbf{C}, \mathbf{I}, \mathbf{O}, \mathbf{E})$ *of a hierarchical FunState component contains a set of functions* $\mathbf{F}$*, a set of storage units* $\mathbf{S}$*, input ports* $\mathbf{I}$*, output ports* $\mathbf{O}$*, embedded components* $\mathbf{C}$ *with input and output ports* $\mathbf{I_C}$ *and* $\mathbf{O_C}$*, respectively, and directed edges* $\mathbf{E} \subseteq \big((\mathbf{F} \cup \mathbf{O_C}) \times (\mathbf{O} \cup \mathbf{S})\big) \cup \big((\mathbf{I} \cup \mathbf{S}) \times (\mathbf{F} \cup \mathbf{I_C})\big)$*. There is at most one edge entering an input port of an embedded component and at most one edge leaving an output port of an embedded component.*

This basically means that a *FunState* component can be regarded as a refinement of a function. A simple example of a hierarchical component is shown in Figure 13.

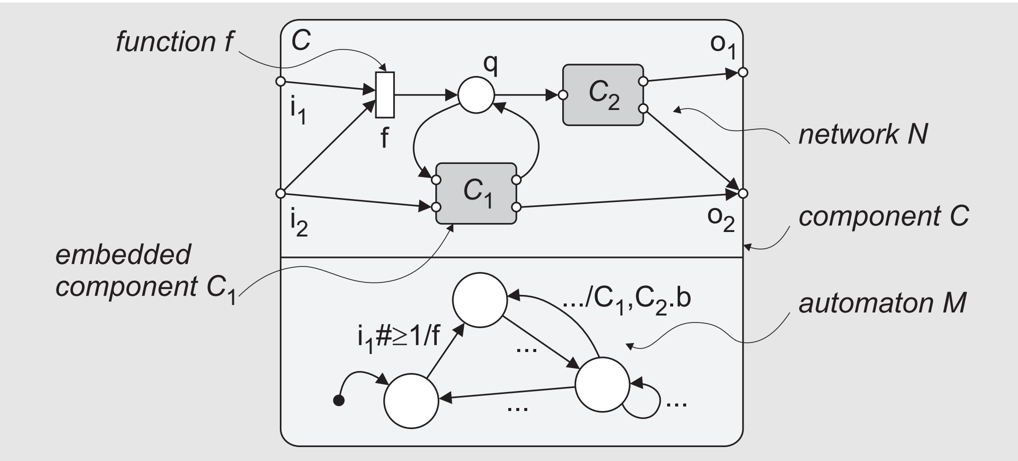

Figure 13. Example of a hierarchical component.

There are two further details concerning the hierarchical *FunState* model which have to be explained.

- The state machine of a component may only access internal storage units or storage units which are directly connected to input ports—via the names of these input ports. For example, the state machine in Figure 13 on the page before may contain a predicate of the form $i_1\# \geq 1$, i. e., the queue connected to input i_1 must contain at least one token.

- The state machine can send events to embedded components. If the action set of a taken transition contains the name of a component, then the state machine of this component can make a transition, i. e., it is activated. If it contains the name of an embedded component augmented with an event, this event is *true* within the embedded component. For example, an action set of the state machine in Figure 13 on the preceding page may look like $C_1, C_2.b$. Then components C_1 and C_2 are activated, i. e., the state machines may perform a reaction. In addition, b is *true* in C_2.

The detailed semantics of the hierarchical model can described best by constructing an equivalent flat model. Before doing this, some characteristics of the hierarchical model are summarized.

- The hierarchy can be nested arbitrarily deep.

- Each component contains a state machine as well as a network of functions, embedded components, and storage units.

- Communication between embedded components is performed via the explicit exchange of tokens, i. e., not synchronous.

- The outer component explicitly activates its children in a synchronous way, i. e., enables them to make a transition.

- Flattening a hierarchical *FunState* model is fairly simple. In particular, hierarchy does not extend the computational model of the basic *FunState* model.

2.2.3.3 Removing Hierarchy

The unfolding process of removing hierarchy is formally described in [TTN+98]. Here, only an informal description is given by means of an example. In Figure 14 on the next page, part of a hierarchical *FunState* model and its equivalent flat model are shown.

If predicate b is *true*, the state machine of C makes a transition. It also activates the components C_1 and C_2. The condition a for the transition of the state machine of C_1 is *true* also, resulting in a reaction of the state machine of C_1 in the same execution cycle if p is satisfied, too.

The unfolding of C_1 is performed in three steps:

1. Functions and embedded components are moved to the outer component.

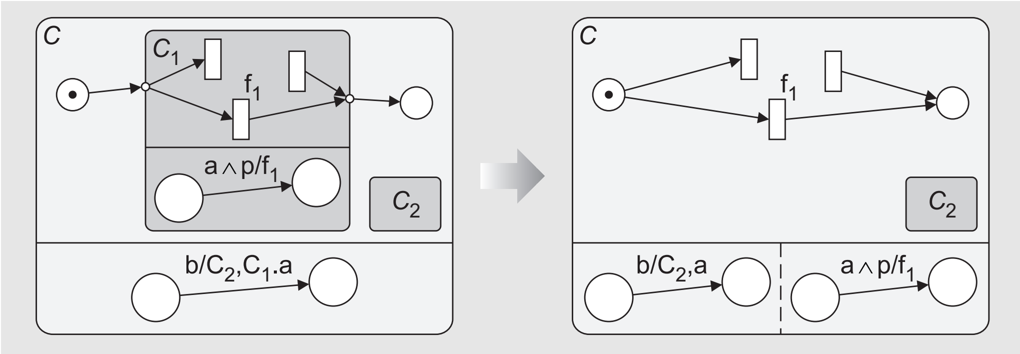

Figure 14. Part of a hierarchical *FunState* model and its unfolding.

2. Sources (targets) of edges from input (to output) ports of the embedded component are moved to the corresponding queue in the outer component.

3. The state machine of the embedded component is moved to the outer component as an AND composition. Names of events from the outer to the embedded component are made equal (and unique). An activation of an embedded component only is replaced by a new (unique) event that is added to the action set of the activating transition in the outer component and to each of the embedded component's predicates.

A more elaborate concept of hierarchy could be achieved by the following extension. In the hierarchical *FunState* semantics defined above, state transitions within embedded components are totally synchronous to the evoking transition in the superordinate state machine emitting the event. An additional concept of execution proves useful that allows components to independently make more than one step as a consequence of a received event. Hence, subsequent transitions in an embedded component as a whole can be triggered by only one event. This could be achieved by dividing states into two classes of which one behaves as usual (*stable* states). Transitions beginning in the intermediate *stutter* states of the other class are considered as always being enabled even if the component currently is not triggered by an event.

With regard to this, the issue has to be settled if the invoking transition is complete after having emitted the event or only after a stable state has been reached. Connected with this question is the possibility of an embedded component to directly influence the behavior of the outer state machine by delaying its execution. This also has significant impact on the timed *FunState* model introduced below.

Since the flattening process described above would be impeded significantly by this possible extension, it is sacrificed in the scope of this work. Nevertheless, the modification would considerably support hierarchical refinement strategies.

2.2.3.4 The Timed *FunState* Model

In this section, the *FunState* model is extended with information concerning the timing of its execution. Two classes of timing information can be distinguished: timing properties and timing constraints.

- Timing properties specify certain execution durations. A typical example is the latency of a function.
- The purpose of timing constraints is to check whether or not any execution of a system satisfies them. They do not restrict the behavior of the system. Timing constraints can be used for guiding the specification or design process of a system by specifying certain deadlines.

It is possible to define timing constraints and timing properties in an equivalent way. A flag could be attached to some timing annotation which distinguishes between properties and constraints. But it turns out that this generality comes with a high notational overhead. Therefore, the class of possible timing annotations is restricted to a useful subset, namely *latency* in the case of properties and *paths* in the case of constraints. Depending on the particular application domain it may be necessary to include more general annotations.

The definition of the latency of a function enables a timed execution of a model, i. e., with each state change a physical time can be associated.

Def. 2.2.3: *The function $\tau : \mathbf{F} \rightarrow \mathbb{R}_{\geq 0}$ associates a latency with each function of the dataflow network N of a* FunState *component. The function $T : \mathbf{F} \rightarrow \mathbb{R}_{\geq 0}$ associates a remaining processing duration with each active function.*

In the following, τ and T are abbreviated by $\tau(f_i) = \tau_i$ and $T(f_i) = T_i$. The operational semantics of the basic *FunState* model has to be extended to incorporate function latency. The state machine part executes independent of the timing of functions. The respective steps are changed or added as follows.

1. **Initialization**
   The current state of the state machine is set to its initial state. All queues and registers are filled with initial tokens according to the initial marking of the dataflow network. Global time t is set to $t = 0$.
2. **Predicate evaluation**
   Same as before.
3. **Check for possible progress**
   If there are no active functions and no satisfied predicates of state machine transitions, the execution is stopped.
4. **State machine reaction**
   Same as before.

5. **Function activation**
All activated functions become active and are started in non-determinate order by removing tokens (reading values) from their input storage units. It is illegal to reactivate a function that is already active at that instant. The remaining processing duration timers of the newly activated functions f_i are set to $T_i = \tau_i$.

6. **Check for time progress**
If the result of "Predicate evaluation" (step 2) was that at least one state machine transition predicate is satisfied at the current time t or if no functions are active, then the global time is not advanced. Otherwise global time t is progressed to the point in time t' when the functions f_i with the least remaining processing duration T_i will be finishing their processing. The remaining processing durations T_j of all other functions f_j are diminished accordingly.

7. **Function finalization**
All functions f_i that finish processing at the—possibly increased—global time t are finalized. Finalization means to add the tokens (write the values) produced by the computational process of a function to its output storage units. If two functions add tokens to the same queue at the same instant, the resulting token order is non-determinate. If two functions write to the same register at the same instant, a non-determinate decision is made about which write action defines the final state. The execution continues at "Predicate evaluation" (step 2).

The hierarchical interpretation of the operational semantics again is defined via an equivalent flat *FunState* model as described in Section 2.2.3.2. In the case of $\tau_i = 0$ for all $f_i \in \mathbf{F}$, the operational semantics of the *FunState* model with function latencies is equal to the operational semantics of the basic untimed model.

As an example of timing constraints, path constraints can be used for specifying certain deadlines[4]. Rules defining path constraints and a constructive method for checking whether or not a certain execution trace satisfies a given path constraint are described in [TTN+98].

Certain real-time scheduling methods are based on preemption of tasks. This is modeled best by augmenting the set of possible actions with instructions such as suspend f and resume f, interrupting and continuing the execution of function f, respectively. In addition, it is useful to be able to retrieve the current status of a function (its *processing state*). The extension of both concepts to components is conceivable. For instance, it often proves useful to enable the communication via predicates on queue contents and internal states which are exported by a component to its superordinate component. Those predicates then can be used

[4]Note again that it is also possible to define path properties in a similar way. Then the system is built in a way that only executions satisfying these path properties are possible.

within transition conditions of the outer component's state machine. Otherwise a component could emit signals to outer parts of the model only by producing tokens via its output ports.

2.2.4 Regular State Machines

The purpose of this section is to introduce the underlying computational model of *FunState*. It serves as the basis for the methods derived in this monograph, i. e., formal verification and scheduling. Because of the simplicity of this model and its thorough investigation in combinatorial mathematics, many further results can be expected in the future.

The class of *regular state machines* (RSMs) turns out to be a natural description for the state evolution of concurrent, control or data oriented systems. Such systems include *FunState* as well as the models of marked graphs, synchronous dataflow, cyclo-static dataflow, Petri nets, and extensions thereof. Contrary to finite automata, a regular state machine is able to model certain classes of infinite state systems. They extend the class of finite state machines in the same way as recurrence equations [KMW67] or regular iterative algorithms [RK88] extend dependence graphs or directed acyclic graphs. A more detailed comparison with other models is described in [TTS00].

The RSM model is introduced in its simplest form. Nevertheless, it can easily be extended to more general settings, see [TTS00]. In particular, the following class of *FunState* models is treated.

- The conditions in the *FunState* model do not contain data dependencies, i. e., the free variables in predicates denote numbers of tokens in queues only.
- Only queues are considered, no registers.
- Suppose that the hierarchy of components has been unfolded using the techniques described above.
- The functions have constant consumption and production rates c and p, respectively.
- Timing is neglected.

Regular state machines are introduced via the notions of *static* and *dynamic state transition diagrams*.

Def. 2.2.4: (Static state transition diagram) *A static state transition diagram is a directed edge-labeled graph* $G = (\mathbf{V}, \mathbf{A}, D, P, v_0, I_0)$ *with*

- *a set of nodes* $\mathbf{V}$,
- *a set of directed edges* $\mathbf{A}$, *where* $a = (v_1, v_2)$ *denotes an edge with source* $v_1 \in \mathbf{V}$ *and target* $v_2 \in \mathbf{V}$,

- *a function* $D : \mathbf{A} \rightarrow \mathbb{Z}^m$ *which associates with each edge* $a = (v_1, v_2) \in \mathbf{A}$ *an integer* distance vector $d(a) = d(v_1, v_2) \in \mathbb{Z}^m$ *of dimension* m,
- *a* predicate function $P : \mathbf{A} \times \mathbb{Z}^m \rightarrow \{\mathit{true}, \mathit{false}\}$ *which associates with each edge* $a = (v_1, v_2) \in \mathbf{A}$ *a predicate* $P(a, I)$, *and*
- *a node* $v_0 \in V$ *and a vector* $I_0 \in \mathbb{Z}^m$ *which form the initial state.*

The static state transition diagram as defined above is a shorthand notation of the (infinite) state transition diagram of a regular state machine, denoted as a dynamic state transition diagram.

Def. 2.2.5: **(Dynamic state transition diagram)** *The dynamic state transition diagram* $G_d = (\mathbf{X}, \mathbf{T}, x_0)$ *of a given static state transition diagram* $G = (\mathbf{V}, \mathbf{A}, D, P, v_0, I_0)$ *is an infinite directed graph defined as follows.*

- *The nodes* $\mathbf{X}$ *are called the* states *of the regular state machine.* $\mathbf{X} = \mathbf{V} \times \mathbf{I}$ *holds where* $\mathbf{I} = \mathbb{Z}^m$ *denotes the* index set *of the regular state machine.* $x = (v, I) \in \mathbf{X}$ *denotes a state for a node* $v \in \mathbf{V}$ *of the static state transition diagram and an index point* $I \in \mathbf{I}$ *with vector elements* i_k.
- *The state* $x_0 = (v_0, I_0)$ *is the initial state.*
- *The edges* $\mathbf{T}$ *are called* transitions *of the dynamic state transition diagram. There is an edge* $t = (x_1, x_2) \in \mathbf{T}$ *from state* $x_1 = (v_1, I_1) \in \mathbf{X}$ *to state* $x_2 = (v_2, I_2) \in \mathbf{X}$ *iff* $a = (v_1, v_2) \in \mathbf{A}$, $I_2 - I_1 = d(v_1, v_2)$, *and* $P(a, I_1) = \mathit{true}$.

An example of the relationship between a *FunState* model and its computational model is given in Figure 15. The numbers of tokens in queues $q_k\#$ correspond to the respective vector elements i_k as introduced above.

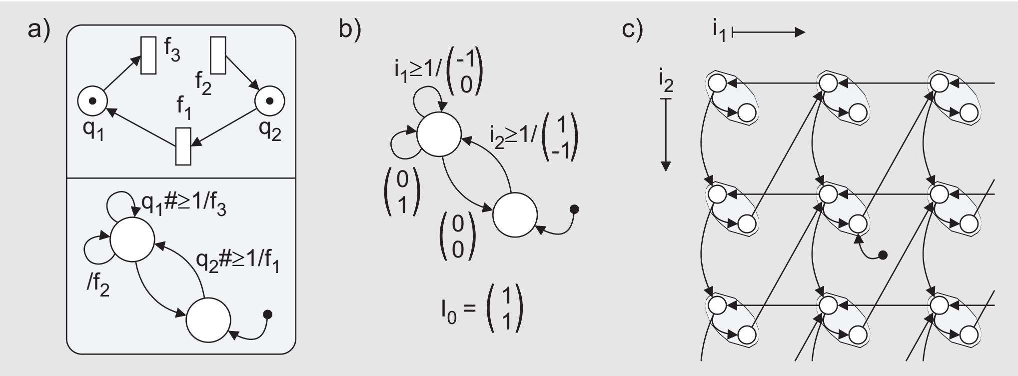

Figure 15. a) A basic *FunState* model, b) its equivalent static state transition diagram, and c) its dynamic state transition diagram.

The edges of the static state transition diagram are labeled in the form $P(a, I)/d(a)$. If the predicate is *true* for all index points $I \in \mathbb{Z}^m$ or if $d(a) = 0$, only the label $d(a)$ or $P(a, I)$ is given, respectively. The dynamic state transition diagram in Figure 15 c) shows only part of the index space. At each index point $I \in \mathbb{Z}^m$, there exist two states corresponding to the two states of the static state transition diagram. The index point $I = (0, 0)^T$ is in the upper left corner.

Regular state machines are defined using the usual semantics of state transition graphs.

Def. 2.2.6: (Regular state machine) *The behavior of a regular state machine with a dynamic state transition diagram $G_d = (\mathbf{X}, \mathbf{T}, x_0)$ is defined as follows:*

- *Initially, the regular state machine is in state x_0.*
- *A state transition $x_1 \xrightarrow{t} x_2$ changes the state of a regular state machine from $x_1 \in \mathbf{X}$ to $x_2 \in \mathbf{X}$ where $t = (x_1, x_2) \in \mathbf{T}$ is chosen non-deterministically from all transitions t with source x_1.*

A given *FunState* model can be transformed into a static state transition diagram by a simple syntactic operation. In particular, the nodes of the finite state machine in the *FunState* model are the nodes $\mathbf{V}$, the transitions the edges $\mathbf{A}$, the predicates on the transitions are P, and the initial state is v_0. The dimension m is the number of queues in the *FunState* model, I_0 is a vector containing the numbers of initial tokens, and $d(a)$ denotes the change in the number of tokens caused by the transition corresponding to a.

The state transition diagram of a *FunState* model is given by the dynamic state transition diagram of the corresponding regular state machine. The *FunState* model is in state $x_0 = (v_0, I_0)$ initially. A state transition via some edge $a = (v_1, v_2) \in E$ with source v_1 and target v_2 may happen iff the state machine is in a state $x_1 = (v_1, I_1)$ for some index point I_1 and $P(a, I_1) = true$. After the transition, the *FunState* model is in state $x_2 = (v_2, I_2)$ with $I_2 = I_1 + d(a)$.

In other words, the state $x_1 = (v_1, I_1)$ of a *FunState* model consists of the static state represented in the static state transition diagram and an index vector $I_1 \in \mathbb{Z}^m$. A state change within the static state transition diagram via a static transition a is possible only if the transition is active, i. e., if the corresponding predicate $P(a, I_1)$ is satisfied. Then, the state change via a leads to a new index vector $I_2 = I_1 + d(a)$.

The RSM model is similar to that of vector addition systems or Petri nets. But RSMs can contain several nodes for each index point I. Moreover, many results from combinatorial mathematics are known for the class of periodic graphs considered here, e. g., [BST92, KS88, Orl84].

2.2.5 Relationship to Other Models

Since the *FunState* model serves as an internal representation, properties relevant to scheduling and formal verification of different input specifications should be easily representable.

The modeling power of *FunState* results neither from the concept of hierarchical or parallel automata (as they can be transformed to simple automata without events) nor from the concept of embedded components (as they can be flattened). Instead of this, the partition into a purely reactive part (state machine) without computations and a passive functional part is the main issue.

Nevertheless, it cannot be expected that efficient analysis, code generation, and scheduling techniques exist in general. As shown below, the combination of embedded components with refinement and abstraction mechanisms leads to a feasible approach to solving these complex problems.

The following comparison may lead to useful application or domain specific restrictions of the *FunState* model. This is one of the major capabilities which leads to efficient methods for this internal model. In the following, only the relationship between *FunState* and communicating finite state machines, marked graphs, synchronous dataflow, cyclo-static dataflow, Petri nets, and SPI is dealt with. Further models of computation such as Boolean and dynamic dataflow graphs introducing data dependent dataflow are investigated in [TSZ+99].

2.2.5.1 Communicating Finite State Machines

Basic concepts of statechart-like specifications and synchronous parallel state machines like Argos [Mar89] are directly included as the *FunState* model supports AND and XOR substates. As two further examples, the communication mechanisms of the POLIS [BCG+97] model for specification and design of embedded systems and those of communicating finite state machines are described in some detail.

- **Communicating finite state machines**
  In the case of a communicating finite state machine, e. g., as in SDL process networks [SSR89], asynchronously operating finite state machines communicate via FIFO ordered queues. An FSM M_1 can write a value into a queue q during a transition. An FSM M_2 can guard its transitions with predicates on the value of the first element $q\$1$ in queue q. If the transition is taken, the element is removed from the queue.

  In the *FunState* model, the finite state machines can be embedded into components, e. g., C_1 and C_2 for M_1 and M_2. Writing into and removing from queues can be modeled using functions, e. g., f_1 and f_2, with production and consumption rates of 1. The asynchronous reactions of the finite state machines are implemented using a finite state machine M in the top component with one state and loops for each finite state machine, see Figure 16 on the next page.

- **Polis**
  The POLIS model [BCG+97] has been invented for designing control dominated embedded systems. Here, it is shown how the communication mechanism can be represented in the *FunState* model. All finite state machines operate asynchronously, here M_1 and M_2. They communicate via single element buffers, for instance, r. If an FSM M_1 writes into this buffer,

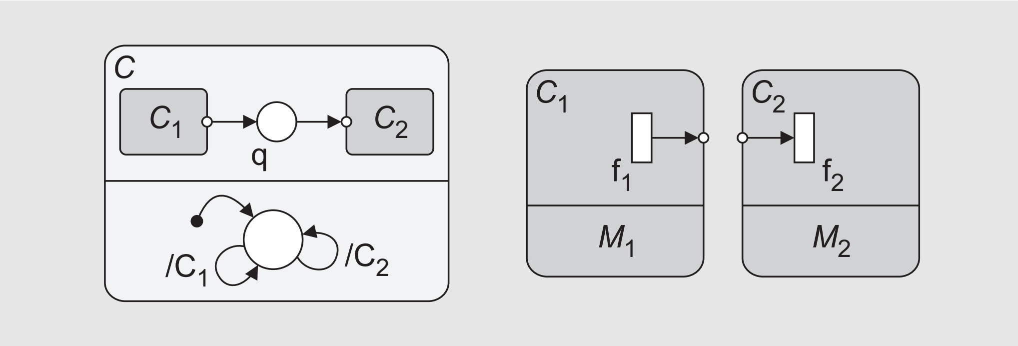

Figure 16. Representation of communicating finite state machines.

the old value is replaced by a new one. Reading from the buffer is non-destructive. This communication model can be represented as shown in Figure 17. It corresponds to the communication via a POLIS data signal. Other communication mechanisms such as general signals and control signals (involving also events) can be modeled easily as well.

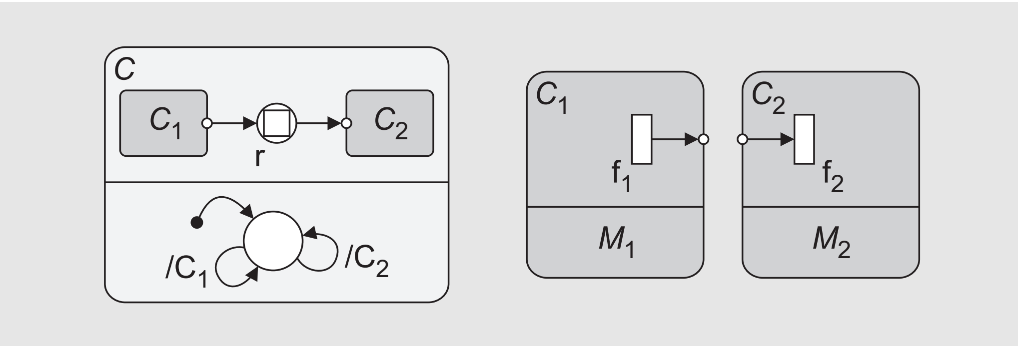

Figure 17. Representation of the POLIS model.

2.2.5.2 Marked Graphs and Synchronous Dataflow

Marked graphs [CH71] and synchronous dataflow graphs [LM87a, LM87b] are labeled directed graphs with nodes $v \in \mathbf{V}$ representing the actors of the system and edges $e \in \mathbf{E}$ denoting the communication and the corresponding queues between the actors. A function $m : \mathbf{E} \to \mathbb{Z}_{\geq 0}$ determines for each edge the number of initial tokens in the corresponding FIFO queue. Two functions $c : \mathbf{E} \to \mathbb{Z}_{>0}$ and $p : \mathbf{E} \to \mathbb{Z}_{>0}$ denote the numbers of tokens removed from the queue if the actor at its target fires and the number of tokens added to the queue if the actor at its source fires, respectively. An actor may fire if in its input queues e there are at least $c(e)$ tokens. For the special class of marked graphs, $c(e) = p(e) = 1$ holds for all edges e.

A *FunState* model which behaves like an SDF graph can be constructed as follows. The whole graph is embedded into one component. Each actor is replaced by a function, each edge is replaced by a concatenation of an edge, a queue, and another edge. The initial numbers of tokens in these queues are determined by the initial numbers of tokens

$m(e)$ on the edges of the SDF graph. The values $c(e)$ and $p(e)$ are written at the corresponding incoming and outgoing edges of the functions. The state machine of the component has one state and one loop transition for each actor. The condition of each loop is the firing condition of the corresponding dataflow actor ($e\# \geq c(e)$ for all input queues e), and the action is an activation of the corresponding function. Thus, although the actors are no longer autonomous, no further constraints on the model execution have been added.

Figure 18 b) (identical to Figure 7 on page 19) shows a *FunState* model corresponding to the SDF graph shown in Figure 18 a). This *FunState* model is constructed as explained above and is an example of a *global control strategy*.

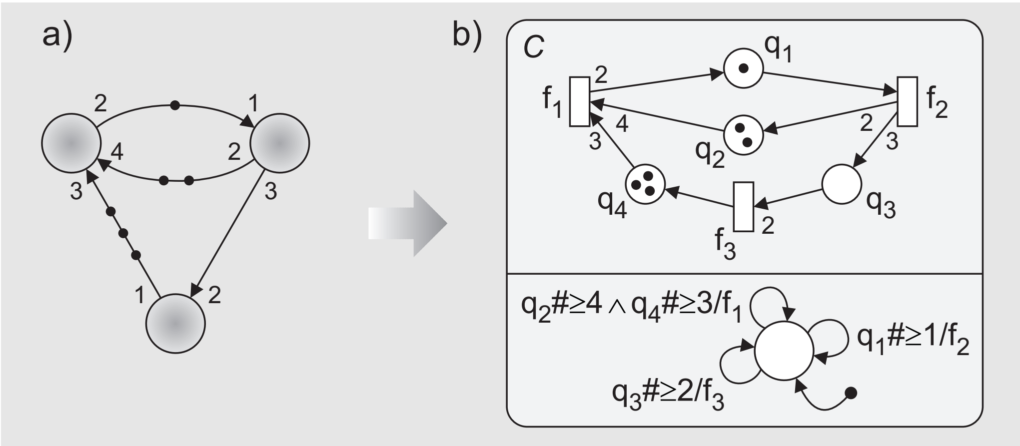

Figure 18. Synchronous dataflow graph and corresponding *FunState* model.

An example of a *FunState* model with *local control strategy* is shown in Figure 19, also representing the SDF graph in Figure 18 a). In the following section, the local control strategy is used for representing cyclo-static actors.

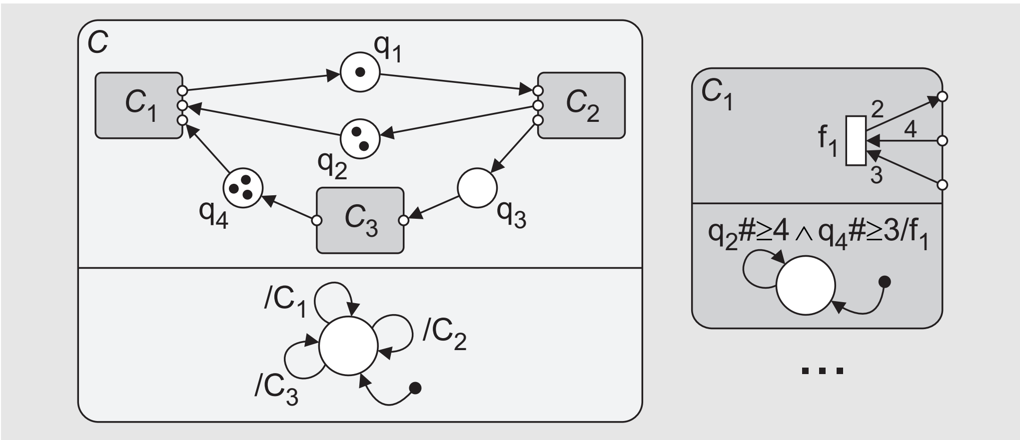

Figure 19. Example of a *FunState* model with local control. Only embedded component C_1 is shown.

2.2.5.3 Cyclo-Static Dataflow

In cyclo-static dataflow [BWE+93, EBLP94], production and consumption rates of actors are changing periodically. Figure 20 shows a cyclo-static actor and the corresponding *FunState* component. The different communication behaviors of the cyclo-static actor are represented by separate functions in the *FunState* component. The state machine of the *FunState* component steps through all possible consumption and production rates by cyclically activating the corresponding functions. The *FunState* components representing the actors are connected analogously to Figure 19 on the preceding page.

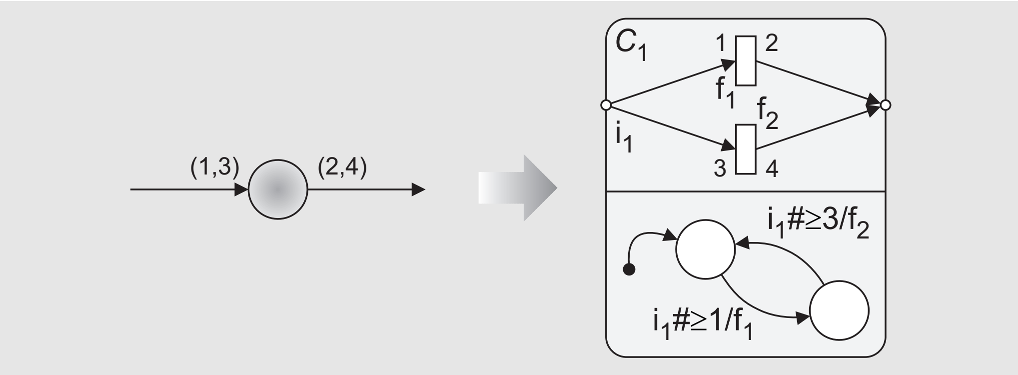

Figure 20. A cyclo-static dataflow node and its *FunState* equivalent.

2.2.5.4 Petri Nets

At a first glance, the *FunState* model seems to be almost equivalent to colored Petri nets (CPN) [Jen90]. But there are several major differences which as well tune the Petri net model to the application domain of the *FunState* model and at the same time generalize it. The following differences can be noted.

- The queues can be related to places in Petri nets. But queues in the *FunState* model have a FIFO behavior whereas this is not the case in CPN. This restriction matches the modeling power necessary for embedded systems and simplifies the operational semantics to a great extent.
- Usually, there are no registers defined in CPN. In order to model the usual mechanism of passing values through writing and reading of variables, this capability has been added.
- The activation and firing conditions are more general than in CPN as arbitrary predicates on the queues in the preset of a function can be used. Moreover, in the *FunState* model, these predicates can be different from the number of tokens removed while firing, e. g., it is possible that a function is activated if there are at least 4 tokens in an input queue, but at the time of firing only 2 of them are removed.

- In a CPN, the transitions are continuously ready for being activated. In the *FunState* model, this can be controlled by the finite state machine. This capability enables a simple consideration of limited resources and scheduling policies.

2.2.5.5 System Property Intervals

In contrast to *FunState*, SPI does not explicitly separate between data and control flow. Although SPI processes may have internal data and hence an internal state [ZER+98], this state is not explicitly represented and thus not visible. Differences in a SPI process external behavior due to state dependencies are modeled by uncertainty intervals. Even the refinement of process behavior using process modes [ZRE+98] does not have a notion of state since the execution mode of a process is determined only based on the contents of incoming channels and is "forgotten"at completion of execution. Thus, with the existing set of constructs (excluding function variants and configurations as proposed in [RZE+99]), the state of a SPI model is only composed of the channel contents (amounts of tokens and mode tags). *FunState* refines the SPI model by adding the capability of explicitly modeling state information and control flow separately from dataflow.

The most important difference between *FunState* and SPI is the control strategy. While SPI processes are autonomous like actors in dataflow models of computation, *FunState* functions and (embedded) components are controlled by a state machine. Due to state machines in *FunState*, it is not generally possible to represent every *FunState* model with SPI[5]. On the other hand, the representation of SPI models in *FunState* is generally possible and equivalent to the representation of dataflow models using a local control strategy (see Figure 19 on page 33).

Straightforward correspondences exist for the directly equivalent storage elements in *FunState* and SPI. Also, *FunState* functions and SPI processes without modes and hierarchy directly correspond. In [TSZ+99], translation rules are given and explained by means of simple examples.

2.2.6 Representing Schedules

In a hierarchical approach to solving complex scheduling problems it is necessary that the result of partially scheduling components can be represented in the same model. With this information, further scheduling steps can be performed. This stepwise refinement corresponds to the stepwise reduction of non-determinism in the model which is performed, for instance, by the scheduling approach described in Chapter 5. Here, only an example of a static schedule representation is shown. Further scheduling mechanisms which may be represented by *FunState* models are given in [TSZ+99, TTN+98].

[5]It is possible to explicitly model the state machine by a process that controls the execution of each element of the dataflow network. But the synchronous semantics is lost when doing this.

Consider a purely static periodic schedule of the synchronous dataflow graph shown in Figure 18 a) on page 33 for a uni-processor system. Methods to construct such a schedule are well known and thus not repeated here.

The chosen schedule executes the functions f_1, f_2, and f_3 iteratively in the order $f_2\,f_3\,f_1\,f_2\,f_3\,f_3$. In comparison with Figure 18 b), only the state machine of component C has to be changed in order to represent the schedule. Figure 21 shows two different possibilities, both reflecting the periodic schedule described above. The second possibility takes into account that the sub-sequence $f_2\,f_3$ occurs twice in the schedule and uses the AND composition facility of parallel state machines. Analogously, a compact representation of the more restricted class of *single appearance schedules* as described in [TZB98] is possible.

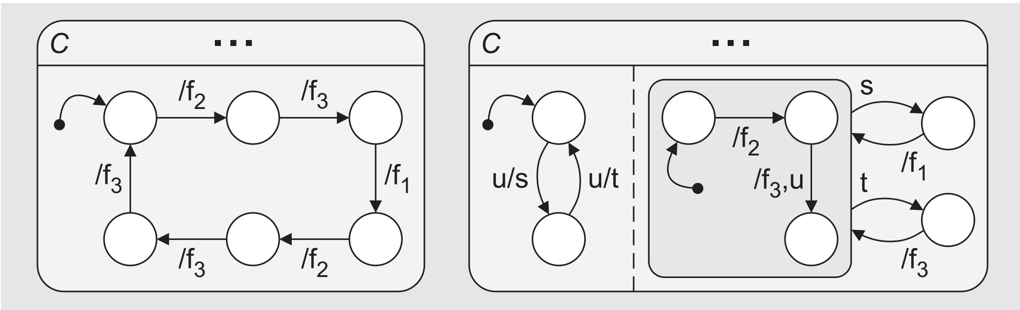

Figure 21. Two possibilities for static periodic scheduling.

2.3 Timed Automata

Many approaches have been introduced to modeling timing behavior of real-time systems, mostly derived from conventional finite state automata which are expanded to describe timing properties of the transition behavior, e. g., several classes of *timed transition systems* [HMP92], *timed graphs* [ACD93], *event clock automata* [AFH97], *state clock automata* [RS97], *timed transition graphs* [CC94], *quantitative temporal structures* [FGK96], *interval structures* [RK97a], or statecharts [Har87]. As one of the most universal approaches, Alur and Dill have proposed *timed automata* [AD94], represented by state-transition graphs with timing constraints using finitely many clocks.

For timed models of computation, the meaning of the term *state* comprises in addition to the usual discrete system state—as, for instance, in a finite state machine—some actual timing information of the system describing, e. g., part of the timed execution history. This timing information often is represented by clocks, timers, or intermediate stutter states.

2.3.1 The Timed Automaton

Figure 22 shows an example timed automaton. It may be used for modeling two independent non-determinate input oscillators (denoted by v_1 and v_2) of which the pulse widths are known only within given time ranges. The length of the 0-pulse of v_1 is in the range of $[1, 2]$, while the 1-pulse's length lies in $[2, 5]$. For v_2, the ranges are $[3, 5]$ and $[1, 4]$, respectively.

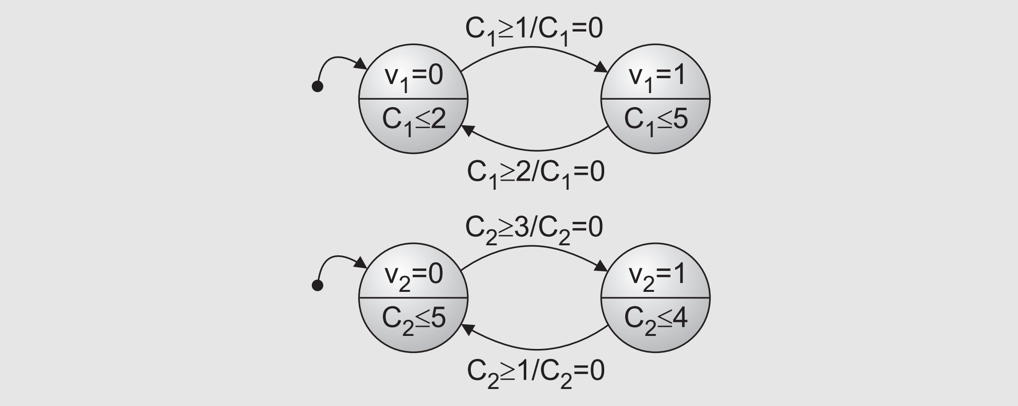

Figure 22. Example timed automaton.

First, a brief and informal introduction to timed automata is given. They are defined formally below. The automaton in Figure 22 has four *locations* depicted by circles and two clocks C_1 and C_2 which are supposed to be set to 0 at the beginning. v_1 and v_2 are state variables encoding the discrete automaton states. The product of the locations of the partial automata results in four discrete states $q \in \mathbf{Q}$ with $q = (v_1, v_2)$ and $\mathbf{Q} = \{ (0,0), (0,1), (1,0), (1,1) \}$. Starting with the *configuration* $(q, C_1, C_2) = \big((0,0), 0, 0\big)$, representing the entity of discrete state and all clock values, time progresses and makes the values of C_1 and C_2 increase uniformly. The automaton is allowed to stay in a given location as long as the corresponding *staying condition*—depicted in the lower part of each location—is satisfied. The *guards* at the transitions represent conditions which have to be satisfied to enable the respective transition. If a transition is taken, given clocks are reset to 0.

2.3.2 Definition

Timed automata are completely defined and described in [AD94]. In this section, the following definitions analogous to [ABK+97] are used. Bold-face letters are used for denoting vectors in $\mathbb{R}^d$, i. e., $\mathbf{v}$ stands for $(v_1, \dots, v_d)$ where $v_i \in \mathbb{R}$ for $i = 1, \dots, d$. For $\mathbf{u}, \mathbf{v} \in \mathbb{R}^d$, $\mathbf{u} \leq \mathbf{v}$ denotes that $u_i \leq v_i$ for $i = 1, \dots, d$. A set $\mathbf{S} \subseteq \mathbb{R}^d$ is said to be *monotonic* iff for every $\mathbf{u} \in \mathbb{R}^d$ satisfying $\mathbf{u} \leq \mathbf{v}$, $\mathbf{v} \in \mathbf{S}$ implies $\mathbf{u} \in \mathbf{S}$.

$\mathbf{G}_{qq'}$ denotes the subset of the clock space satisfying the transition guard from q to $q' \neq q$, while $\mathbf{G}_{qq}$ represents the set of clock values satisfying the staying condition of q. The number of clocks is denoted by

d. For timed automata, $\mathbf{G}_{qq'}$ and $\mathbf{G}_{qq}$ are restricted to be *k-polyhedral* subsets of $\mathbb{R}^d$—sets resulting from the application of set-theoretic operations to half-spaces of the form $\{\, \mathbf{v} \mid v_i \leq c \,\}$, $\{\, \mathbf{v} \mid v_i < c \,\}$, $\{\, \mathbf{v} \mid v_i - v_j \leq c \,\}$, or $\{\, \mathbf{v} \mid v_i - v_j < c \,\}$ for some integer $c \in \{\, 0, \ldots, k \,\}$. Such sets are called *regions* and constitute the *region graph* [AD94] of which the properties underlie all analysis methods for timed automata. $R_{qq'} : \mathbb{R}^d \to \mathbb{R}^d$ is the *reset function* associated with q and q', setting some of its arguments to 0 while leaving the others intact.

Without loss of generality, the timed automata considered are restricted as follows:

1. At most one transition is associated with each pair of locations,
2. the clock space is $[0, k)^d$ as the clock values are bounded by k,
3. $\mathbf{G}_{qq'}$ is convex for every $q, q' \in \mathbf{Q}$, and
4. $\mathbf{G}_{qq}$ is monotonic for every $q \in \mathbf{Q}$.

Any timed automaton may be easily transformed into one satisfying these properties.

$\mathbf{K}$ denotes the interval $[0, k)$ for dense time or the set $\{\, 0, \ldots, k-1 \,\}$ for discrete time. $\mathbf{z} + t$ stands for $\mathbf{z} + t \cdot \mathbf{1}$, where $\mathbf{1} = (1, \ldots, 1)$ is the d-dimensional unit vector.

Def. 2.3.1: **(Timed Automaton)** *A timed automaton is a triple $\mathcal{A} = (\mathbf{Q}, \mathbf{Z}, \delta)$ such that*

- $\mathbf{Q}$ *is a discrete* state set*,*
- $\mathbf{Z} = \mathbf{K}^d$ *is the* clock space *(*$\mathbf{Q} \times \mathbf{Z}$ *is the* configuration space*), and*
- $\delta : \mathbf{Q} \times \mathbf{Z} \to 2^{\mathbf{Q} \times \mathbf{Z}}$ *is the* transition relation *admitting the following decomposition: For every $q, q' \in \mathbf{Q}$, let $\mathbf{G}_{qq'} \subseteq \mathbf{Z}$ be a k-polyhedral monotonic set and let $R_{qq'} : \mathbf{Z} \to \mathbf{Z}$ be a reset function. Then, for every configuration $(q, \mathbf{z}) \in \mathbf{Q} \times \mathbf{Z}$,*

$$\delta(q, \mathbf{z}) = \left\{ (q', \mathbf{z}') \;\middle|\; \exists t \in \mathbf{K} \textit{ such that } (\mathbf{z} + t \in \mathbf{G}_{qq} \cap \mathbf{G}_{qq'}) \wedge \big(\mathbf{z}' = R_{qq'}(\mathbf{z} + t)\big) \right\} . \tag{2.1}$$

2.3.3 Time Forward Projection

The transition behavior of timed automata defined by equation (2.1) is explained in the following. It consists of both discrete state changes and the progress of time which is performed by simultaneously increasing clock values of the timed automaton.

The application of the transition relation $\delta(q, \mathbf{z})$ results in the set consisting of all configurations reachable from $(q, \mathbf{z})$ after waiting some time t (which may be zero) and then taking at most one transition. The process of waiting prior to the possible discrete transition is called *time forward projection* and is defined as a function $\Phi : 2^{\mathbf{Z}} \to 2^{\mathbf{Z}}$ with

$$\Phi(\mathbf{P}) = \{\, \mathbf{z} + t \mid \mathbf{z} \in \mathbf{P}, t \in \mathbf{K} \,\} \cap \mathbf{Z}. \tag{2.2}$$

$(q, \mathbf{P})$ denotes subsets of $\mathbf{Q} \times \mathbf{Z}$ of the form $\{q\} \times \mathbf{P}$ where $\mathbf{P}$ is k-polyhedral. All subsets of $\mathbf{Q} \times \mathbf{Z}$ encountered in the analysis of timed automata are decomposable into a finite union of such sets. Functions on elements are extended to functions on sets in the natural way, e. g., $\delta\big((q, \mathbf{P})\big) = \bigcup_{\mathbf{z} \in \mathbf{P}} \delta(q, \mathbf{z})$ and $R_{qq'}(\mathbf{P}) = \bigcup_{\mathbf{z} \in \mathbf{P}} R_{qq'}(\mathbf{z})$.

With $\mathbf{P}^{\Phi} = \Phi(\mathbf{P}) \cap \mathbf{G}_{qq}$ and $\mathbf{P}_{q'} = R_{qq'}(\mathbf{P}^{\Phi} \cap \mathbf{G}_{qq'})$ for every q', the immediate successors of a set of configurations $(q, \mathbf{P})$ are denoted as

$$\delta\big((q, \mathbf{P})\big) = (q, \mathbf{P}^{\Phi}) \cup \bigcup_{q' \neq q} (q', \mathbf{P}_{q'}). \tag{2.3}$$

Figure 23 shows the configurations reachable after up to one transition of each partial automaton in Figure 22 on page 37. The state $q = (v_1, v_2)$ corresponding to each of the three regions is given. Beginning with the initial state $q^0 = (0, 0)$ and configuration $(q, C_1, C_2) = (q^0, 0, 0)$, time is projected resulting in the oblique line in the figure, restricted by $C_1 \leq 2$. This corresponds to the time forward projection $\Phi(\mathbf{P}^0) \cap \mathbf{G}_{q^0 q^0}$ of the initial clock region $\mathbf{P}^0 = \{(C_1, C_2) \mid C_1 = C_2 = 0\}$, restricted by the staying condition of q^0.

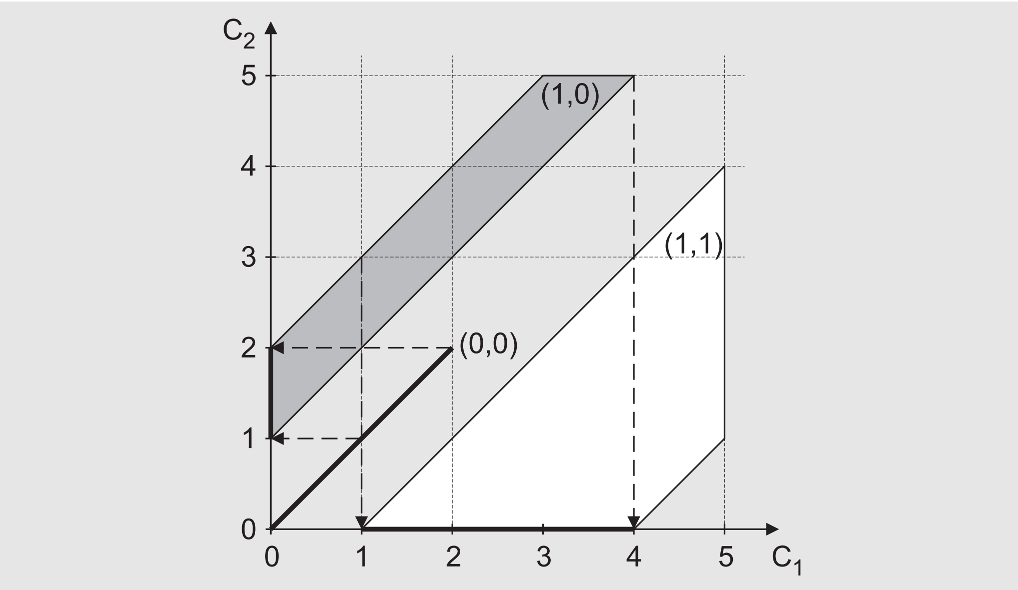

Figure 23. Configurations reachable at first.

For $C_1 \geq 1$, clock C_1 may be reset to 0 by taking the transition from $v_1 = 0$ to $v_1 = 1$. The result of this is depicted by the bold line at the left-hand border of the grey-shaded trapezoid. The set of immediate successors $\delta\big((q^0, \mathbf{P}^0)\big)$ consists of the union of the sets of configurations corresponding to both lines mentioned above. Projecting time up to $C_2 \leq 5$ yields the complete trapezoid for $(1, 0)$. Resetting C_2 for $C_2 \geq 3$ and further projecting time results in the white trapezoid for $(1, 1)$.

3

Symbolic Methods

FORMAL verification provides a mathematical proof of the correctness of a system's model with regard to a given specification which describes certain requirements. In contrast to other verification techniques such as simulation, formal verification considers *all* possible execution traces of a state transition model, not only a part. Hence, many formal verification approaches have to handle huge numbers of system states which has a significant impact on memory and computation time requirements.

To cope with the so-called "state explosion" problem, symbolic approaches have shown to provide at least a partial solution. The main characteristics of symbolic approaches are:

- System states are not enumerated separately but represented by means of sets of states.
- The state transitions of the system are not represented explicitly but implicitly in form of a transition relation.
- Sets are represented by means of characteristic functions, and operations on sets are performed by Boolean operations on their characteristic functions.
- The state space of the system model is traversed breadth-first.

The foundations of these methods have been laid in the late eighties by Coudert, Berthet, and Madre [CBM89a, CBM89b] as well as by McMillan *et al.* [BCMD90, BCM$^+$90, McM93], independently of each other. The symbolic approach revolutionized the whole area of formal verification of automata and had strong influences on the efficient implementation of proof methods for temporal logics as described in Chapter 4. In addition, symbolic techniques yielded to efficiency increases for scheduling applications as explained in Chapter 5. In this chapter, the techniques necessary for both application domains are presented. After motivating and explaining the symbolic approach to state space traversal in Section 3.1, methods dedicated especially to process networks are introduced in Section 3.2.

3.1 Symbolic State Space Traversal

Many formal verification techniques can be reduced to the problem of reachability analysis. The state space traversal used for this can be performed by a fixpoint computation composed of iterated calculations of sets of successor states, beginning with the initial system state. Modifications of this method enable more complex formal verification approaches and even scheduling applications. Thus, the use of symbolic techniques is motivated in the following with reachability analysis as a simple example for state space traversal.

3.1.1 Motivation

Algorithm 1 is a trivial fixpoint algorithm for reachability analysis, where $\mathbf{S}, \mathbf{S}_{old} \subseteq \mathbf{Q}$ are sets of states, $\mathbf{Q}$ is the entire state space, and $\delta : 2^{\mathbf{Q}} \to 2^{\mathbf{Q}}$ represents a transition relation in functional form. The function call *reachable*$(\{ x_0 \})$ returns the set of states reachable from the initial state $x_0 \in \mathbf{Q}$ after any number of state transitions.

Alg. 1. Determine the set of reachable states

1 **funct** *reachable*($\mathbf{S}$) $\equiv$
2 ⌈**do**
3 $\quad\mathbf{S}_{old} := \mathbf{S}$;
4 $\quad\mathbf{S} := \mathbf{S} \cup \delta(\mathbf{S})$
5 **until** $\mathbf{S} = \mathbf{S}_{old}$ **od**;
6 **return** $\mathbf{S}$⌋.

In general, extensions of this algorithm are employed. Nevertheless, the computationally most expensive stages of this and related kinds of algorithms in general is the application of the transition relation $\delta(\mathbf{S})$ at line 4. $\delta(\mathbf{S})$ computes the set of all those states $x' \in \mathbf{Q}$ which are direct successors of at least one of the states $x \in \mathbf{S}$, hence $x \xrightarrow{u} x'$ where u is a valid state transition. The union of $\delta(\mathbf{S})$ with the original state set $\mathbf{S}$ is performed either in a following stage or even combined with the computation of $\delta(\mathbf{S})$. For all state traversal algorithms related to Algorithm 1, great store has to be laid on enhancing the efficiency of the $\delta(\mathbf{S})$ computation as far as possible.

Besides the application of $\delta(\mathbf{S})$ at line 4, the equality test of $\mathbf{S}$ and $\mathbf{S}_{old}$ at line 5—which determines if a fixpoint has been reached—is the crucial operation with regard to efficiency. Thus, having a canonical, unique state set representation[1] is a fundamental factor because testing for equality of sets then often requires only constant time in efficient implementations. Otherwise, equality tests had to be performed, e. g., by explicitly comparing all set elements or by testing for satisfiability of set membership functions.

Symbolic techniques as introduced below turned out to provide the following advantages with regard to the above issues.

- Sets of states are stored using efficient representations, resulting in significant memory savings.
- The application of the transition relation can be computed efficiently—regarding both memory and computation time.
- Data structures are used which enable a canonical representation of state sets and thus fast set comparisons.

3.1.2 Preliminaries

The symbolic techniques treated in this monograph are based on Boolean functions[2] which represent characteristic functions of sets of system states. Like this, the set elements are enumerated implicitly instead of explicitly. The characteristic function of a set is a function with a value of 1 for those arguments that are elements of the set and 0 otherwise. Let $x = (x_1, x_2, \ldots, x_n)$ be a vector describing a system state. Then a state set $\mathbf{S}$ is represented by its characteristic function $s(x)$ with

$$s(x) = \begin{cases} 1 & \text{if } x \in \mathbf{S} \\ 0 & \text{otherwise} \end{cases} . \tag{3.1}$$

The transition behavior of the system is described by a transition relation. Alternatively to the intuitive representation used in Algorithm 1 on the preceding page based on a function $\delta : 2^{\mathbf{Q}} \rightarrow 2^{\mathbf{Q}}$ which maps a set of states onto the set of their successor states, a transition relation can be represented by means of a set $\mathbf{T}$ of pairs (x, x') with $x \xrightarrow{u} x'$. x is a state prior to and x' a successor state after a valid transition u of the state transition system. Hence, $\mathbf{T} = \left\{ (x, x') \,\middle|\, x' \in \delta(\{x\}) \right\}$ holds. $t(x, x')$ is the

[1] Hence, two set representations are equal iff the represented sets are equal.

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

characteristic function representing the transition relation $\mathbf{T}$, i. e.,

$$t(x,x') = \begin{cases} 1 & \text{if } (x,x') \in \mathbf{T} \\ 0 & \text{otherwise} \end{cases} . \tag{3.2}$$

3.1.3 Image Computation

In symbolic formal verification, an important operation is the application of a transition relation to a state set such that the set of all successor or predecessor states, respectively, is calculated in one single computation. The computation of $\delta(\mathbf{S})$ in Algorithm 1 on page 42 is an example of this process called *image computation*. The following definitions are taken from [HD93].

Def. 3.1.1: **(Image)** *The image $Im(\mathbf{S},\mathbf{T})$ of a state set $\mathbf{S}$ with respect to a transition relation $\mathbf{T}$ represents the set of all states that may be reached after exactly one valid transition from a state in set $\mathbf{S}$. The image is defined as $Im(\mathbf{S},\mathbf{T}) = \{\, x' \mid \exists x \text{ with } s(x) \wedge t(x,x') \,\}$.*

Def. 3.1.2: **(Pre-image)** *The inverse image $PreIm(\mathbf{S},\mathbf{T})$ represents all states that* can *reach a state in $\mathbf{S}$ after one transition and is defined as $PreIm(\mathbf{S},\mathbf{T}) = \{\, x \mid \exists x' \text{ with } s(x') \wedge t(x,x') \,\}$.*

Def. 3.1.3: **(Back-image)** *$BackIm(\mathbf{S},\mathbf{T}) = \{\, x \mid \forall x' \text{ holds } t(x,x') \implies s(x') \,\}$ represents the set of states that after one transition* must *end up in $\mathbf{S}$.*

The last operator behaves in some sense as the inverse of the image operator because $BackIm(\mathbf{S},\mathbf{T})$ is the largest set such that $Im\big(BackIm(\mathbf{S},\mathbf{T}),\mathbf{T}\big) \subseteq \mathbf{S}$. It can be computed using the equivalence $BackIm(\mathbf{S},\mathbf{T}) = \neg PreIm(\neg\mathbf{S},\mathbf{T})$[3].

$Im(\mathbf{S},\mathbf{T})$ is used for *forward traversal* as in reachability analysis, while the *backward traversal* operators $PreIm(\mathbf{S},\mathbf{T})$ and $BackIm(\mathbf{S},\mathbf{T})$ mostly are employed in more elaborate techniques as described in Section 4.2. The effect of the image operators is illustrated in Figure 24 on the next page. They are applied to some state sets $\mathbf{A}$ and $\mathbf{B}$ with respect to some transition relation $\mathbf{T}$.

3.1.4 Symbolic Computation

The key to symbolic techniques is to perform all calculations directly using Boolean expressions, rather than considering the state transition system itself. Thus, the computation of image operators is performed using Boolean operations on characteristic functions instead of explicitly manipulating sets. As an example, only the image $Im(\mathbf{S},\mathbf{T})$ is considered here.

According to the above definitions, the set $\mathbf{S}' = Im(\mathbf{S},\mathbf{T})$ can be ascertained by computing its characteristic function $s'(x') = \exists x : s(x) \wedge t(x,x')$.

[3]The negation $\neg$ of a set here and in the following denotes the set's complement with respect to $\mathbf{Q}$, hence $\neg\mathbf{S} = \mathbf{Q} \setminus \mathbf{S}$.

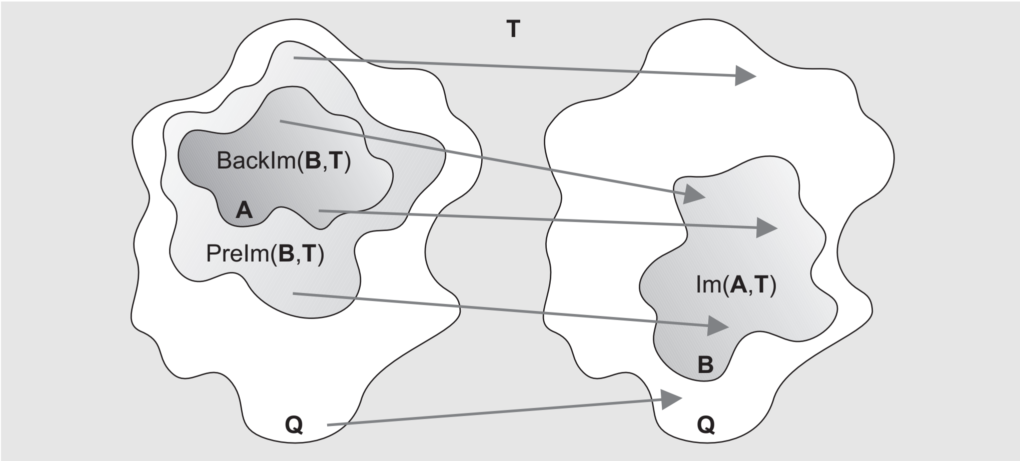

Figure 24. Image operators.

Using the equivalences

$$\begin{aligned} \exists x : f &= \exists x_1, \ldots, x_n : f, \\ \exists x_i, x_j : f &= \exists x_i : (\exists x_j : f), \text{ and} \\ \exists x_i : f &= f|_{x_i = d_{i1}} \vee \ldots \vee f|_{x_i = d_{im}} \end{aligned}$$

for $x_i \in \{ d_{i1}, \ldots, d_{im} \}$, the whole computation of $s'(x')$ and hence implicitly of $\mathbf{S}'$ can be performed using common Boolean operations.

By means of image operators, line 4 of Algorithm 1 on page 42 can be expressed equivalently to $\mathbf{S}_\cup = \mathbf{S} \cup \mathbf{S}'$. This union is computed via the characteristic function $s_\cup(x) = s(x) \vee s'(x)$, thus again using only Boolean operations. Note that a variable substitution is necessary here to transform primed variables in $s'(x')$ into unprimed variables in $s'(x)$.

Since even the set equality test at line 5 of Algorithm 1 can be performed by solely comparing Boolean functions, the whole state space traversal can be performed symbolically. For large system models, this often is the more efficient approach than using explicit, enumerative techniques.

Binary decision diagrams (BDDs) [Ake78, Bry86] used as symbolic representations of Boolean functions have formed the basis of the breakthrough of a very successful formal verification method called *symbolic model checking*, e. g., [BCM+92, BCL+94, McM93, CGP00]. BDDs are an efficient data structure for representing Boolean functions and enable their very fast manipulation. BDDs provide a canonical representation of Boolean functions making function comparisons (including the special cases tautology and satisfiability[4]) trivially easy. In the following, the reader is supposed to be familiar with the basic concepts of BDDs and their derivatives. Comprehensive expositions can be found in [Bry86] and [SF96].

[4]Tautology means the identity of a Boolean function with the Boolean constant 1, while satisfiability describes the fact that a function is not identical to 0.

3.2 Interval Diagram Techniques

Symbolic model checking as described in Section 4.2 has promised to enable the automatic verification of complex, real-world systems. For large classes of problems, however, this promise has yet to be fulfilled. The original symbolic model checking approach has been applied to conventional finite state systems only. Unfortunately, the use of BDD-based symbolic formal verification methods has several drawbacks with regard to process networks as explained in the following.

3.2.1 Motivation

3.2.1.1 The Problem

The traditional BDD-based method of automated verification suffers from the fact that a binary representation of the process network and its state is required. One severe problem is that the necessary capacities of the buffers in general are unknown prior to the analysis process or even may be its desired result. But to perform conventional symbolic formal verification, the buffer contents represented by an integer number have to be binary coded. To save memory and computing power, the coding should be selected such that it covers no more than the integer range necessary—which is 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 have only binary but integer values and thus are able to represent the state of a buffer 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 requiring an infinite number of MDD edges which is not possible. One strategy to avoid this is again to bound the variable domain to a finite range such that complementary sets are finite, too.

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

3.2.1.2 Interval Diagrams

Interval diagram techniques as introduced in [ST98] have been developed to overcome the above-mentioned limitations of traditional symbolic approaches with regard to process networks and related models of computation. Interval diagram techniques are based on both *interval decision diagrams* (IDDs) and *interval mapping diagrams* (IMDs), succeeding basically due to the use of a more reasonable way of describing the above-mentioned form of transition relations. The approach affords the possibility of representing 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. This way, the partial behavior mentioned above may be described by one single graph node denoting a *state distance* of 1. No artificial limitation by an upper state variable bound is necessary.

For the application of symbolic formal verification and symbolic scheduling to process networks, the combination of internal process states and buffer contents can be represented and manipulated efficiently using interval diagrams. Due to a similar transition behavior, timed automata can be handled as well.

This section introduces the formalism of interval decision diagrams, an efficient representation of discrete-valued functions. It is related to similar representations such as binary decision diagrams. Then, as a transition relation representation adequate especially to process networks, interval mapping diagrams are presented, together with the methods and techniques necessary for applying interval diagrams to symbolic verification and scheduling.

The major enhancements of IDDs and IMDs are:

- No state variable bounds due to binary coding or complementation are necessary as with conventional symbolic techniques.
- The transition relation representation is fairly compact—especially for the considered kind of models—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 are more compact than using BDDs.

3.2.2 Interval Decision Diagrams

3.2.2.1 Notation

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

$$f : \mathbf{P}_1 \times \mathbf{P}_2 \times \ldots \times \mathbf{P}_n \rightarrow \mathbf{Q}_f,$$

where $\mathbf{P}_i \subseteq \mathbb{R}$ are the domain sets of the variables x_i, and $\mathbf{Q}_f$ is the discrete and finite range set of f.

The term $x^{\mathbf{I}}$ represents a *literal* of a variable x with respect to a set $\mathbf{I} \subseteq \mathbb{R}$, that is the Boolean function

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

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

In the following, mainly intervals on $\mathbb{R}$ or $\mathbb{Z}$ are dealt with, denoted as real or integer intervals, respectively. Intervals can be of the forms $[a, b]$, (a, b), $(a, b]$, or $[a, b)$, where a and b represent the upper and lower bounds and $a \to -\infty$ and $b \to \infty$ is allowed[5]. 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 called *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 called *neighboring* iff they may be joined by union into a larger interval, where overlapping intervals are called neighboring, 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 $\mathbf{I} \subseteq \mathbf{P}_i$, f does not depend on x_i, i. e.,

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

then f is *independent* of variable x_i in $\mathbf{I}$, and the cofactor of f with respect to the literal $x_i^{\mathbf{I}}$ is defined by

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

In this case, $\mathbf{I}$ is called an *independence interval* of f with respect to x_i. From now on, all domain sets $\mathbf{P}_i$ are supposed to be intervals.

Def. 3.2.1: **(Interval cover)** *The set* $\mathbf{I}(\mathbf{P}_i) = \{ \mathbf{I}_1, \mathbf{I}_2, \ldots, \mathbf{I}_{p_i} \}$ *of* p_i split intervals $\mathbf{I}_j$ *represents an* interval cover *of* $\mathbf{P}_i$ *iff each* $\mathbf{I}_j$ *is a subset of* $\mathbf{P}_i$*, i. e.,* $\mathbf{I}_j \subseteq \mathbf{P}_i$*, and* $\mathbf{I}(\mathbf{P}_i)$ *is* complete*, i. e.,*

$$\mathbf{P}_i = \bigcup_{\mathbf{I} \in \mathbf{I}(\mathbf{P}_i)} \mathbf{I}.$$

Def. 3.2.2: **(Interval partition)** *An interval cover is* disjoint *iff*

$$\forall j, k \text{ with } 1 \leq j, k \leq p_i, j \neq k : \mathbf{I}_j \cap \mathbf{I}_k = \emptyset$$

holds, i. e., no element of $\mathbf{P}_i$ *is included in more than one split interval. A disjoint interval cover is called* interval partition*.*

[5] Then a round bracket like in $[a, \infty)$ is required for such infinite bound.

Def. 3.2.3: **(Independence interval partition)** *An* independence interval partition *is an interval 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 $\mathbf{I}(\mathbf{P}_i)$, given by

$$f = \bigvee_{\mathbf{I}\in\mathbf{I}(\mathbf{P}_i)} x_i^{\mathbf{I}} \wedge f_{x_i^{\mathbf{I}}}. \tag{3.3}$$

The operations $\vee$ and $\wedge$ in this equation suppress all but one function value of f that corresponds to the respective value of x_i.

Def. 3.2.4: **(Reduced interval partition)** *An independence interval partition is called* minimal *iff it contains no neighboring split intervals that may be joined into an independence interval. An interval partition* $\mathbf{I}(\mathbf{P}_i) = \{\, \mathbf{I}_1, \mathbf{I}_2, \ldots, \mathbf{I}_{p_i} \,\}$ *is* ordered *iff the higher bounds of all split intervals form an increasing sequence with respect to their indices. An independence interval partition which is minimal and ordered is called* reduced.

Theorem 1: *A reduced independence interval partition of a domain* $\mathbf{P}_i$ *is* unique.

Proof: The proof of this is by contradiction and only sketched here. Assume that two different independence interval partitions of $\mathbf{P}_i$ exist which both are reduced. Regarding each possible special case of being different (different numbers of contained intervals, unequal intervals, different orders) results in the conclusion that both partitions cannot be reduced and different at the same time. Hence, a reduced independence interval partition must be unique. ■

In order to exemplify the above definitions, consider a function $f(x_1, x_2, x_3)$ with the integer variable domains $\mathbf{P}_i = [0, \infty)$ and the range $\mathbf{Q}_f = \{\, a, b \,\}$ denoted as

$$f(x_1, x_2, x_3) = \begin{cases} a & \text{if } x_1^{[0,3]} \cdot x_2^{[0,5]} \vee x_1^{[4,5]} \vee x_1^{[6,\infty)} \cdot x_3^{[0,7]} \\ b & \text{otherwise} \end{cases}.$$

For instance, the cofactor $f_{x_1^{[4,5]}}(x_1, x_2, x_3)$ is a. The intervals $[0, 7]$ and $[4, 5]$ both are independence intervals of f with respect to x_3, while $[6, 9]$ is not. The set $\mathbf{I}(\mathbf{P}_1) = \{\, \mathbf{I}_1, \mathbf{I}_2, \mathbf{I}_3 \,\}$ with $\mathbf{I}_1 = [0, 3]$, $\mathbf{I}_2 = [4, 5]$, and $\mathbf{I}_3 = [6, \infty)$ is an interval partition. With respect to f, $\mathbf{I}(\mathbf{P}_1)$ is an independence interval partition and even a reduced interval partition.

3.2.2.2 Structure

The interval decision diagram which represents the above function $f(x_1, x_2, x_3)$ is shown in Figure 25. Though the edges are not depicted by arrows, the graph is directed such that all edges are oriented from top to bottom. In order 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 annotated with real or integer intervals.

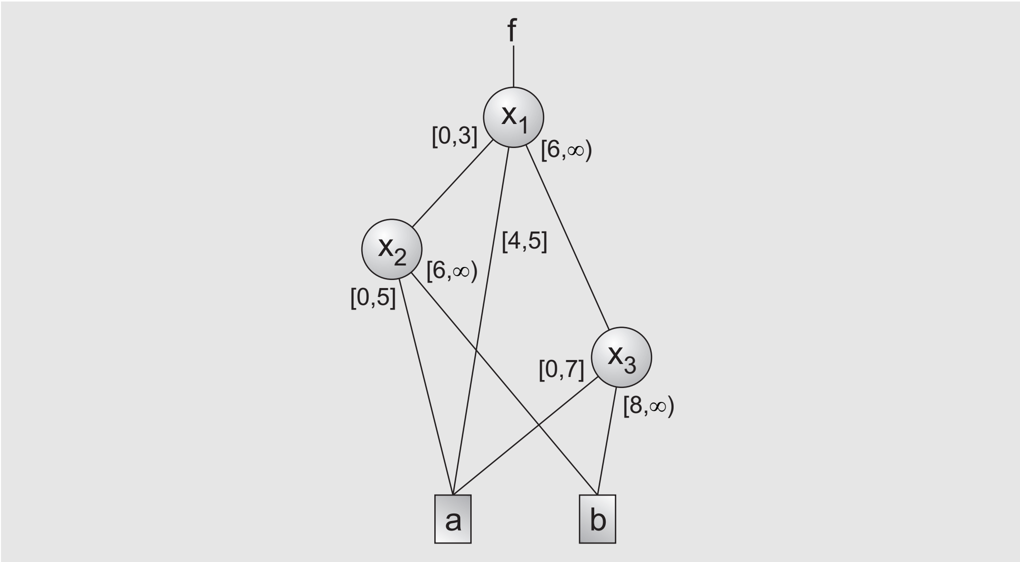

Figure 25. Example interval decision diagram.

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, analogously 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.

3.2.2.3 Representation

IDDs as sketched above are represented by canonical *function graphs*, similar to those of [SKMB90] and [Bry86] for BDDs.

Def. 3.2.5: **(Function graph)** *A function graph G is a rooted, directed acyclic graph with a node set* $\mathbf{V}$ *containing two types of nodes. A* non-terminal *node $v \in \mathbf{V}$ has as attributes an argument index $i = index(v)$, an independence interval partition $part(v) = \mathbf{I}(\mathbf{P}_i) = \{\mathbf{I}_1, \mathbf{I}_2, \ldots, \mathbf{I}_{p_i}\}$, and $p_i = |part(v)|$ child nodes denoted as $child_k(v) \in \mathbf{V}, 1 \leq k \leq p_i$. The split intervals $int_k(v) = \mathbf{I}_k \in \mathbf{I}(\mathbf{P}_i)$ of the partition* *are assigned to the* corresponding *graph edges* $\big(v, child_k(v)\big) \in \mathbf{E}$. *A* terminal *node v has as attribute a value $value(v) \in \mathbf{Q}_f$.*

The correspondence between function graphs and multi-valued functions is defined as follows.

Def. 3.2.6: *The function f_v associated with a node $v \in \mathbf{V}$ 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 described by the Boole-Shannon expansion as described in equation* (3.3) *on page 49, thus*

$$f_v = \bigvee_{\mathbf{I}_j \in part(v)} x_i^{\mathbf{I}_j} \wedge f_{child_j(v)}. \tag{3.4}$$

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

For instance, the IDD node v labeled with x_2 in Figure 25 on the preceding page denotes the function

$$f_v(x_1, x_2, x_3) = \begin{cases} a & \text{if } x_2^{[0,5]} \\ b & \text{otherwise} \end{cases}.$$

The cofactor $f_{x_1^{[1,2]}}(x_1, x_2, x_3)$, for example, is identical to this function.

In the context of decision diagrams, functions are considered as equivalent to their associated nodes. Hence, a function f_v associated with a node v with variable index i may be represented by a (p+1)-tuple $f_v = \big(x_i, (\mathbf{I}_1, F_1), \ldots, (\mathbf{I}_p, F_p)\big)$, where $(\mathbf{I}_k, F_k)$ denote pairs each consisting of split interval $\mathbf{I}_k = int_k(v)$ of the interval partition $part(v)$ and the function F_k associated with the respective corresponding child node $child_k(v)$. This description is directly associated with the Boole-Shannon expansion mentioned above.

Def. 3.2.7: **(Ordered function graph)** *A function graph is* ordered *iff for any adjacent pair of non-terminal nodes $\big(v, child_k(v)\big)$ the inequality $index(v) < index\big(child_k(v)\big)$ holds.*

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

Def. 3.2.8: **(Reduced function graph)** *A function graph G is* reduced *iff*

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.*

The function graph of the interval decision diagram shown in Figure 25 on page 50, for instance, is ordered and reduced. In the following, one of the major results of this chapter is described.

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

Proof: The proof proceeds along the same lines as those in [Bry86] and [SKMB90], by induction on the size of the dependence set $\mathbf{D}_f$. The dependence set of a function f is the set of arguments that f depends upon, i. e.,

$$\mathbf{D}_f = \{\, x_i \mid \exists b, c \in \mathbf{P}_i, b \neq c, \text{ such that } f|_{x_i=b} \neq f|_{x_i=c} \,\}\,.$$

The following statements are assumed without further explanation.

- Since 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 $|\mathbf{D}_f| = d > 0$, 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 $\mathbf{I}(\mathbf{P}_i) = \{\, \mathbf{P}_i \,\}$ 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 $\mathbf{D}_f$ which contradicts the assumption.

For $|\mathbf{D}_f| = 0$, 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 because 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 $|\mathbf{D}_g| < d$ holds: If the function graph H of g is reduced, it is unique. Then for each function f with $|\mathbf{D}_f| = d$, if its function graph G is reduced, then it is unique. The proof of this is subdivided into two parts. First, the claim is that as all subfunctions g associated with nodes w with $index(w) > i$ satisfy $|\mathbf{D}_g| < d$, all those subfunctions are different. Otherwise, the subgraphs of two equal subfunctions would be isomorphic due to the uniqueness of functions graphs. This would violate the fact that G is reduced. In particular, all children of the root node v represent different functions represented by non-isomorphic subgraphs. Now, consider the Boole-Shannon expansion of f with respect to x_i as described above. Let v be the root of G with $index(v) = i$. Its independence interval partition $part(v)$ is unique. Consequently, the number of outgoing edges from v is fixed, and their labels—associated intervals—are unique. Moreover, the edge with label $\mathbf{I}_j$ ends in the node $child_j(v)$ which is unique. Otherwise, there would exist two identical subfunctions associated with nodes with $index(w) > i$.

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

3.2.2.4 Implementation

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

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

Similarly to BDDs, the common IDD structure is stored using a hash table called *unique table* [BRB90] for efficiency as well as for canonicity. The canonical form is maintained by looking up in the unique table each time before inserting a node into the IDD. A new node is created only if no equivalent node exists in the unique table. Otherwise, this existing node is used instead of inserting a new one. The unique table allows a single multi-rooted directed acyclic graph to represent any number of Boolean functions simultaneously.

3.2.2.5 Variable Ordering

Assume that the functions being represented all have the same n arguments $x_1, \ldots, x_n$. The variable ordering must be the same for all functions. It is well known that the sizes of decision diagrams in the worst case often grow exponentially with the number of variables involved. But fortunately this is not the case for many real-world applications, where sometimes even only linear dependencies occur.

Nonetheless, the DD size heavily depends on the ordering chosen for the variables. While exact approaches to determine an optimal variable ordering mostly are infeasible in practice for BDDs, many approaches exist to determine a "good" ordering. Some of them make use of genetic or evolutionary algorithms [DBG95, DGB96], other approaches based on *sifting* [Rud93] pairwise swap BDD layers until a good ordering has been found. These methods can be applied both statically prior to the actual BDD computations or even dynamically, interrupting the computations. Another class are heuristics exploiting stuctural information of the problem such as the topology of the underlying system [MWBS88]. For IDDs, these three classes are directly applicable. In Section 4.2.3.3, a simple topology-based ordering heuristic especially dedicated to process networks is given.

3.2.2.6 Reducing a Function Graph

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

1. If all outgoing edges of a non-terminal node end in the same child, this node must be eliminated and its entering edges must be redirected into the child node.

2. If two equal-childed nodes exist or two distinct terminal nodes with the same value, in either case these two nodes 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. Neighboring edges are joined into one by uniting their split intervals.

A node which has to be removed due to rule 1 is called *obsolete*. The rule holds particularly if the interval partition consists of one single split interval only. The three reduction rules correspond directly to the respective properties of a reduced function graph introduced in Definition 3.2.8 on page 51. If none of the rules can be applied any more, all necessary conditions of a function graph being reduced are satisfied. Thus, the result always is a reduced function graph. The reduction of a function graph is exemplified in Figure 26. 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.

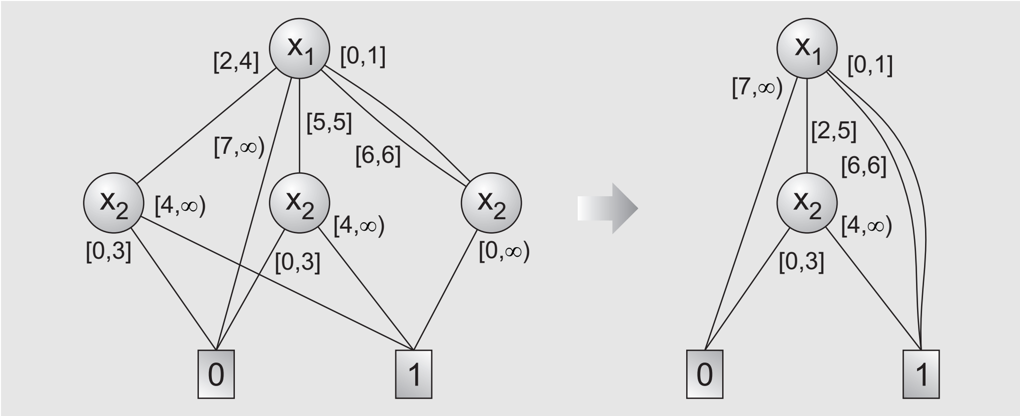

Figure 26. Graph reduction.

3.2.3 IDDs Representing Boolean Functions

For conventional symbolic formal verification of the described class of models of computation, mainly 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 the values 0 and 1 only and thus are called *0-* and *1-terminal nodes*, respectively. With the use of such kind of IDDs, truth functions and propositions such 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 clarity, only integer intervals are treated because 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, assume that all integer intervals are denoted by closed intervals, where the only exceptions are positive or negative infinite bounds (that have to be open).

3.2.3.1 Interval Cover Disjunction

Some operations described below make use of the transformation of a non-disjoint interval cover $\mathbf{I}(\mathbf{P}_i) = \{\mathbf{I}_1, \mathbf{I}_2, \ldots, \mathbf{I}_{p_i}\}$ into a partition $\tilde{\mathbf{I}}(\mathbf{P}_i) = \{\tilde{\mathbf{I}}_1, \tilde{\mathbf{I}}_2, \ldots, \tilde{\mathbf{I}}_{\tilde{p}_i}\}$ consisting of *greatest common intervals* (GCIs). By uniting some of these split intervals, each original interval may be expressed. The

GCI partition $\tilde{\mathbf{I}}(\mathbf{P}_i)$ of a cover $\mathbf{I}(\mathbf{P}_i)$ is the unique minimal interval partition with the property

$$\mathbf{I}_j = \bigcup_{k=1}^{\tilde{p}_i} (\mathbf{B}_{jk} \cap \tilde{\mathbf{I}}_k), \forall j \in \{1, 2, \ldots, p_i\}, \mathbf{B}_{jk} \in \{[\,], (-\infty, \infty)\}, \tag{3.5}$$

where $\mathbf{B}_{jk}$ indicate whether or not a GCI $\tilde{\mathbf{I}}_k$ is included in the original interval $\mathbf{I}_j$. The postulated property of $\tilde{\mathbf{I}}(\mathbf{P}_i)$ being minimal must hold with regard to the number $\tilde{p}_i$ of GCIs, thus supposing that all $\tilde{\mathbf{I}}_k$ are as large as possible, and no neighboring GCIs $\tilde{\mathbf{I}}_k$ 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 by Algorithm 2. First (lines 2 to 4), the original integer intervals $[a, b]$ are transformed into real ones $[a', b']$ including them as shown in Figure 27 on the next page. This is achieved by adding the value $-\frac{1}{2}$ to the lower and $\frac{1}{2}$ to the upper bound of the original interval. The resulting bounds are sorted implicitly by using the unique, sorted list L of real numbers. Then the disjoint intervals are ascertained by back-transforming successive real bounds into integer intervals $[\tilde{a}, \tilde{b}]$ (lines 5 to 7).

Alg. 2. Interval cover disjunction

1 **funct** *determineGCIs*$\big(\mathbf{I}(\mathbf{P}_i)\big) \equiv$
2 ⌈**foreach** interval $[a, b] \in \mathbf{I}(\mathbf{P}_i)$ **do**
3     $[a', b'] := [a - \frac{1}{2}, b + \frac{1}{2}]$;
4   add a' and b' to list L **od**;
5   **foreach** pair (a', b') of successive elements in L **do**
6     $[\tilde{a}, \tilde{b}] := [a' + \frac{1}{2}, b' - \frac{1}{2}]$;
7   add $[\tilde{a}, \tilde{b}]$ to $\tilde{\mathbf{I}}(\mathbf{P}_i)$ **od**;
8   **return** $\tilde{\mathbf{I}}(\mathbf{P}_i)$⌋.

3.2.3.2 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 clarity, only *ITE* is considered here. *ITE* is a ternary Boolean operation directly derived from the Boole-Shannon expansion and is denoted

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

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

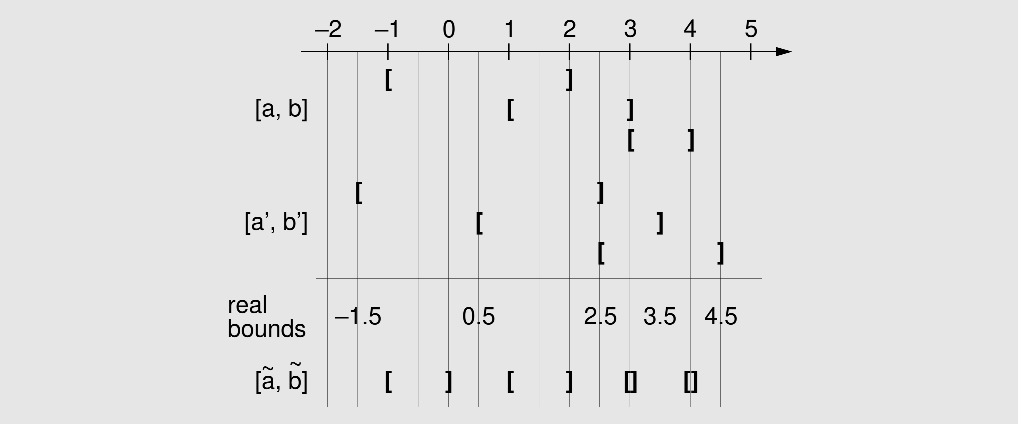

Figure 27. Example of interval cover disjunction.

Let $Z = ITE(F, G, H)$ and let x be the top variable of F, G, and H, i. e., the variable at the highest layer of their IDDs and thus with the lowest index. Analogously to [BRB90], using the Boole-Shannon decomposition *ITE* is recursively defined as

$$ITE(F, G, H) = \\ \Big(x, \big(\mathbf{I}_1, ITE(F_{x^{\mathbf{I}_1}}, G_{x^{\mathbf{I}_1}}, H_{x^{\mathbf{I}_1}})\big), \ldots, \big(\mathbf{I}_p, ITE(F_{x^{\mathbf{I}_p}}, G_{x^{\mathbf{I}_p}}, H_{x^{\mathbf{I}_p}})\big)\Big),$$

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 3.2.3.1 into an independence interval partition. The resulting partition $\{\mathbf{I}_1, \mathbf{I}_2, \ldots, \mathbf{I}_p\}$ is used for the recursive *ITE* calculation as described below. Since this is a mutual decomposition, the possibility of restricting F, G, and H to the split intervals $\mathbf{I}_k$ 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 prior to inserting into the IDD. The *ITE* algorithm is sketched as pseudo code of the function $ite(F, G, H)$ by Algorithm 3 on the following page.

The terminal cases and their results are given above. Within function *findOrAddInIDD*(v), first the unique table of the IDD is searched for the argument node v. If such a node already is included in the unique table, it is returned. Otherwise, a new node with the specified properties is created and inserted into the unique table. Since the new node is reduced, the reduced form of the IDD is maintained.

Analogously to BDDs, a *computed table* [Bry86, BRB90] consisting of a hash-based cache table of already computed results of $ITE(F, G, H)$ can be

Alg. 3. Computation of the *ITE* operator

funct $ite(F, G, H) \equiv$
⌈**if** terminal case reached
  then return result of terminal case **fi**;
$x :=$ 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 $\{\mathbf{I}_1, \mathbf{I}_2, \ldots, \mathbf{I}_p\}$;
foreach resulting split interval I_k **do**
  $child_k := ite(F_{x^{\mathbf{I}_k}}, G_{x^{\mathbf{I}_k}}, H_{x^{\mathbf{I}_k}})$ **od**;
merge neighboring $\mathbf{I}_k$ if corresponding $child_k$ equal each other;
if only $\mathbf{I}_1$ left
  then return $child_1$ **fi**;
return $findOrAddInIDD\big(x, (\mathbf{I}_1, child_1), \ldots, (\mathbf{I}_k, child_k)\big)$⌋.

employed for efficiency. The computed table serves as a memory functions and returns previously calculated (and not yet overwritten) results without further computation.

Figure 28 shows an example of the application of the If-Then-Else operator where $I = ITE(F, G, H)$. For the sake of clarity, the partial IDDs are depicted separately and not in one common, shared IDD representing all functions used. The calculation of the *ITE* operation is sketched in Table 1 on the facing page. The structure of the resulting IDD can directly be read from the last row of this calculation.

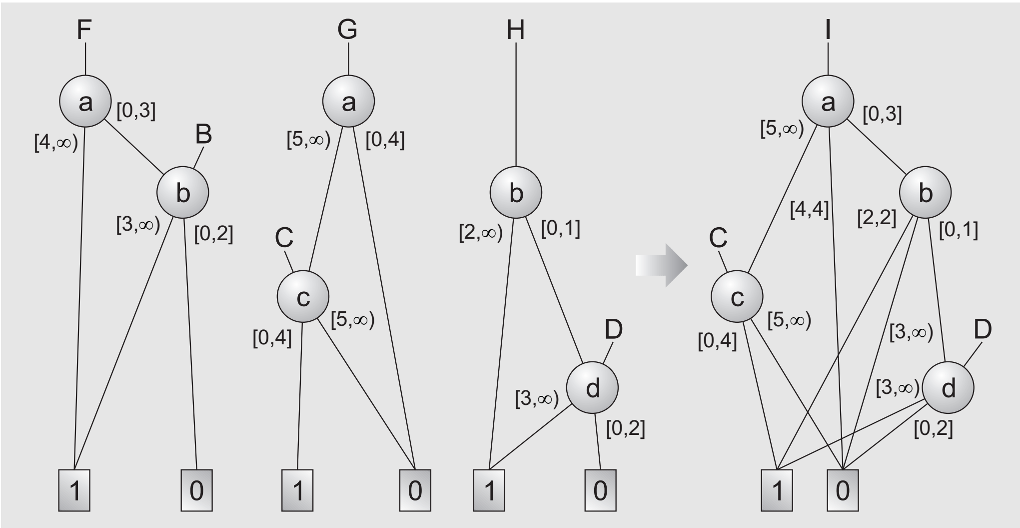

Figure 28. If-Then-Else operator computation $I = ITE(F, G, H)$.

3.2.3.3 Graph Unification

Under certain circumstances—e. g., when constructing an IDD during initialization—a graph may be of importance which is akin to the function graph of an IDD but contains one or several non-disjoint interval

Table 1. Example of *ITE* computation.

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

covers instead of interval partitions. Since the evaluation paths in such a graph are ambiguous, this cannot be a function graph based on the Boole-Shannon decomposition. Nonetheless, 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, a disjunctive behavior is defined where a path to a terminal node with the Boolean value 1 covers all other paths of the same variable assignment. This behavior may be described by adapting the definition of the function f_v associated with a node v in equation (3.4) on page 51 to interval covers as

$$f_v = \bigvee_{\mathbf{I}_j \in cover(v)} x_i^{\mathbf{I}_j} \wedge f_{child_j(v)}, \tag{3.6}$$

where $cover(v)$ replaces the partition of node v. Hence, the *covering rule* means that the value of the corresponding function is 1 for all assignments which have at least one evaluation path ending in the 1-terminal node, otherwise it is 0. An example is shown in Figure 29.

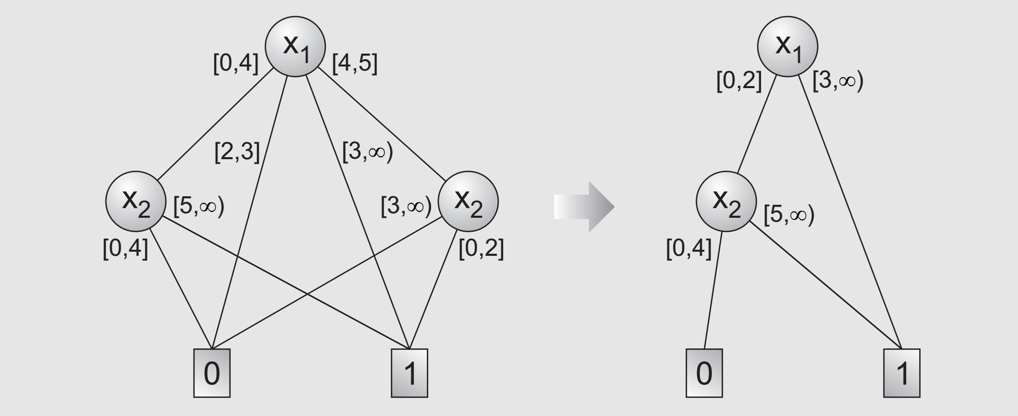

Figure 29. Covering rule.

In the following, 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 3.2.3.1 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 by Algorithm 4. 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.

Alg. 4. Node transformation during unification

funct *transformNode*(v) $\equiv$
⌈join neighboring edges of v;
if cover of v is disjoint **then return fi**;
create new node $\tilde{v}$ with $index(\tilde{v}) := index(v)$;
transform cover $\{\mathbf{I}_1, \mathbf{I}_2, \ldots, \mathbf{I}_{p_i}\}$ of v into partition $\{\tilde{\mathbf{I}}_1, \tilde{\mathbf{I}}_2, \ldots, \tilde{\mathbf{I}}_{\tilde{p}_i}\}$ of $\tilde{v}$;
foreach resulting split interval $\tilde{\mathbf{I}}_j$ **do**
$\mathbf{I} := \{\, \mathbf{I}_k \supseteq \tilde{\mathbf{I}}_j \mid child_k(v) \neq TN_0 \,\}$;
if $\exists\ \mathbf{I}_k \in \mathbf{I} \mid child_k(v) = TN_1$
then $child_j(\tilde{v}) := TN_1$
elsif $\mathbf{I} = \emptyset$
then $child_j(\tilde{v}) := TN_0$
else
$i_{min} := min\Big\{\, index\big(child_k(v)\big) \,\Big|\, \mathbf{I}_k \in \mathbf{I} \Big\}$;
create new node $child_j(\tilde{v})$ with $index(\tilde{v}) := i_{min}$;
foreach $\mathbf{I}_k \in \mathbf{I}$ **do**
if $index\big(child_k(v)\big) > i_{min}$
then add quasi-obsolete edge ending in $child_k(v)$ to $child_j(\tilde{v})$
else add all edges leaving $child_k(v)$ to $child_j(\tilde{v})$ **fi od fi od**;
replace v by $\tilde{v}$ and remove unused $child_k(v)$⌋.

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.

It turned out that the combined application of unification and reduction is recommendable to minimize the extent of computation. First, the three reduction rules of Section 3.2.2.6 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 30 shows the node transformation of the top node of an example graph. The application of the abovementioned covering rule concerning 0- and 1-edges is obvious.

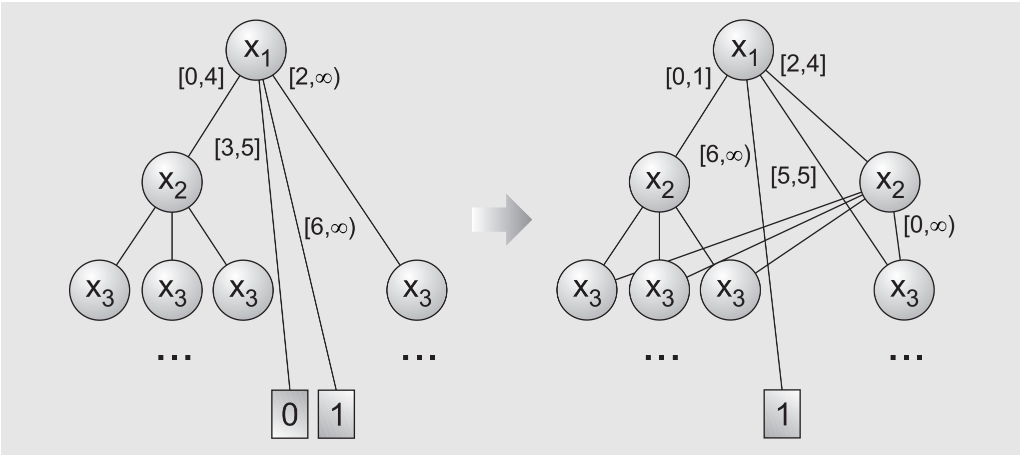

Figure 30. Graph unification.

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

3.2.3.4 MDDs and BDDs

Since 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.

In order to represent 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 represented 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.

3.2.4 Interval Mapping Diagrams

Interval mapping diagrams are used for representing transition relations in symbolic formal verification and symbolic scheduling. They map a set described by a Boolean-valued IDD onto a new set—described by such IDD, too—by performing operations such as shifting or assigning some or all values of the IDD's decision variables. For interval shifting, a simple interval arithmetic is used.

3.2.4.1 Notation

$\mathbb{I}$ denotes the set of all intervals, depending on the integer or real range, respectively. The underlying interval mathematics consists 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 allowed.

Def. 3.2.9: **(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.

Def. 3.2.10: **(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.

3.2.4.2 Representation

IMDs are represented by *mapping graphs* similar to the function graphs described in Definition 3.2.5 on page 50. Their edges are labeled with *interval mapping functions* $f : \mathbb{I} \to \mathbb{I}$, mapping intervals onto intervals.

Def. 3.2.11: **(Mapping graph)** *A mapping graph G is a rooted, directed acyclic graph with a node set* $\mathbf{V}$ *containing two types of nodes. A* non-terminal *node* $v \in \mathbf{V}$ *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 $n = |func(v)|$ child nodes denoted as $child_k(v) \in$ $\mathbf{V}, 1 \leq k \leq n$. The mapping functions $f_k(v)$ are assigned to the corresponding *graph edges $\big(v, child_k(v)\big) \in \mathbf{E}$. $\mathbf{V}$ contains exactly one* terminal node *v with $value(v) = 1$.*

The interval mapping function f mapping an interval onto itself, i. e., $f(\mathbf{I}) = \mathbf{I}$, is called *neutral*. An example of a mapping graph is given in Figure 32 b) on page 66 for a restricted subclass of interval mapping diagrams introduced later on.

3.2.4.3 Interpretation

Informally, the functional behavior of IMDs may be described as "set flow", similar to that of dataflow. The data consists of sets represented by unions of intersected intervals of state variable values, as described in Section 3.2.2.1 and represented by an IDD. Beginning at the root node of an IMD, the set data has the possibility of flowing along each path until reaching the terminal node. Each IMD edge transforms the data according to the respective mapping function. More precisely, the mapping function maps each interval of the corresponding state variable included in the actual set description onto a transformed interval. The effect of this may be, e. g., to shift, shrink, grow, or assign the whole set along exactly one coordinate of the state space. Then the modified set data is transferred to the next IMD node corresponding to another state variable, where the transformation continues.

If an interval is mapped onto the empty interval, this is a degenerate case as the set is restricted to an empty set, i. e., the set data effectively does not reach the end of the computation along this path. From a global view, the set data flows through the IMD from its top to the bottom along all paths in parallel and finally is united in the terminal node to the resulting set. The algorithm to achieve this behavior is described in detail in Section 3.2.5.

3.2.4.4 Reducing a Mapping Graph

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

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

1. If a non-terminal node has only one leaving edge and this edge is annotated with a neutral mapping function, the node is eliminated, and its entering edges are redirected into the child node.

2. Equal-childed nodes are joined into one by eliminating all but one and redirecting all incoming edges into the resting node.

3. Two edges starting in the same node and labeled with the same mapping function are replaced by such an edge to a new node having the same index as the higher one of the two child nodes. If both child nodes are on the same layer and non-terminal, the new node obtains all their leaving edges. If they are on different layers, the new node obtains all edges of the higher child node and an edge annotated with a neutral mapping function to the lower child node.

3.2.4.5 Predicate Action Diagrams

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

Def. 3.2.12: (Predicate action diagram) *A predicate action diagram is an interval mapping diagram containing only the following mapping functions:*

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

and

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

where f_+ is the shift function *and $f_=$ the* assign function*. $\mathbf{I}_P$ is called* predicate interval *and $\mathbf{I}_A$* action interval*. Each mapping function within a PAD can have different intervals.*

Thus, the interval mapping functions have one among two possible structures. Both types first intersect the argument interval with the predicate interval and then—if 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. 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 $\mathbf{I}_P/+\mathbf{I}_A$ is used for the shift function f_+ and $\mathbf{I}_P/=\mathbf{I}_A$ for the assign function $f_=$. The shift about $\mathbf{I} = [a, b]$ in reverse direction corresponding to interval substraction is achieved by addition of $-\mathbf{I} = [-b, -a] = \mathbf{I}_A$ and is denoted as $\mathbf{I}_P/-I$.

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 $\mathbf{P}_i/+[0, 0]$. Assign functions cannot be neutral. An example PAD is shown within the following section.

3.2.5 Image Computation

3.2.5.1 Using IDDs and IMDs

Figure 31 shows an example process network with uncertain consumption and production rates represented by intervals (depicted as edge labels). It is akin to a computation graph [KM66] where the consumption rate is independent of the threshold—denoted as a condition here. For instance, process A is enabled if queue u contains at least two tokens and at the same time queue v contains at least one token. If A is executed then, it consumes one or two tokens from u and one token from v, and produces one to three tokens in queue w.

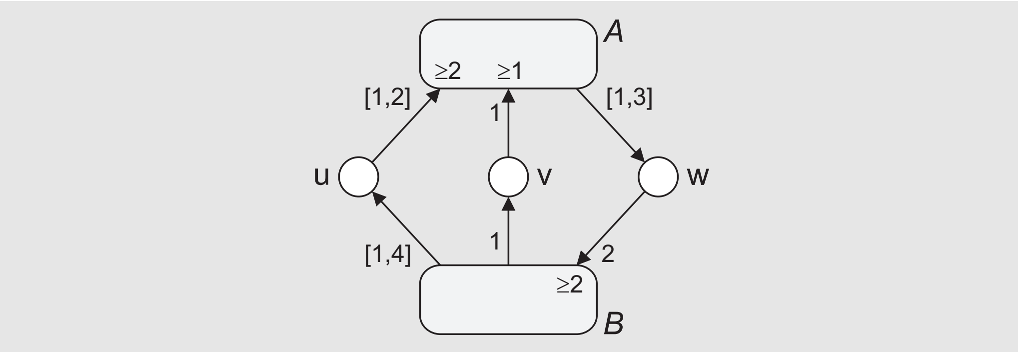

Figure 31. Example process network.

Figure 32 b) on the next page shows the corresponding transition relation represented by the IMD T—which in fact is a PAD—, Figure 32 a) an example state set IDD S. The state set is denoted as $s(u, v, w) = (v = 0) \vee (1 \leq v \leq 2) \wedge (w = 0)$, the transition relation as $t(u, v, w, u', v', w') = (u \geq 2) \wedge (u - u' \in [1, 2]) \wedge (v \geq 1) \wedge (v' = v - 1) \wedge (w' - w \in [1, 3]) \vee (u' - u \in [1, 4]) \wedge (v' = v + 1) \wedge (w \geq 2) \wedge (w' = w - 2)$, for $u, v, w, u', v', w' \in [0, \infty)$. Note that $P_i = \mathbb{Z}$ is assumed here with regard to the variable domains of the IDD in Figure 32 a). If $P_i = \mathbb{Z}_{\geq 0}$ were assumed, both nodes having edges labeled with $[0, \infty)$ would be obsolete.

The left-hand path of the IMD in Figure 32 b) corresponds to the transition behavior of process A in Figure 32 a), while the right-hand path corresponds to process B. The procedure of deriving an IMD from a given process network is described in Section 4.2.3.2.

For a clearer view, an IDD representation is used in the figures where the 0-terminal node and all its incoming edges are omitted. Thus, for each non-terminal node, all split intervals that miss for a complete interval partition have to be added implicitly and connected with the invisible 0-terminal node. This description has no impact on the real internal representation of an implementation. It should not be confused with the DD class of *zero-suppressed BDDs* (ZBDD) [Min93] which are a BDD derivative based on modified reduction rules.

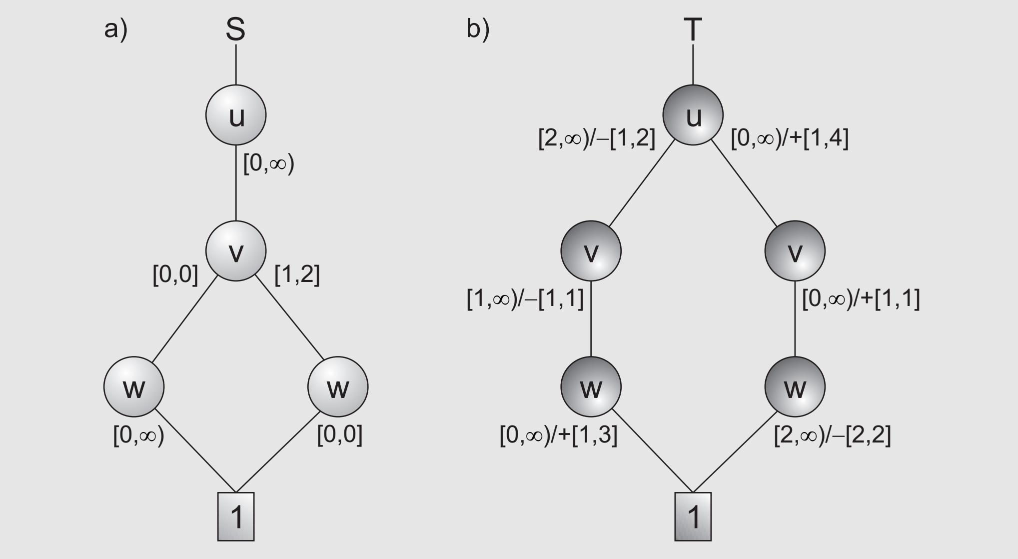

Figure 32. State set IDD and transition relation IMD.

3.2.5.2 Computation

In this section, it is described how to perform image computation using IDDs and IMDs. Conventionally, as mentioned in Section 3.2.1.1, the transition relation is represented by a BDD that explicitly stores all valid combinations of system state and successor state. Image computation is performed using Boolean operations such as existential and universal quantification $\exists$ and $\forall$, respectively—which internally can be computed using the If-Then-Else operator *ITE*, see [McM93], for instance. This strategy is possible for IDDs, too.

A more advanced technique is described in the following. A major advantage of IMDs in contrast to BDDs is the possibility of efficiently storing mapping functions describing partial transition behaviors instead of many state variable value pairs. This way, image computations cannot be performed as usual any longer. Instead, an IMD representing the transition relation is combined with an IDD storing a set of system states, in a way that resembles the piecewise application of an operator to two sets, similarly to Bryant's *Apply* operator [Bry86]. First, a general form of transition relation based on IMDs is described. Then the focus is on further restrictions using PADs which enable the efficient representation of state distances combined with the corresponding transition conditions.

- **Forward mapping**

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

The function *mapForward*(v, w)—over an IDD node v and an IMD node w—performs this image operation recursively, as sketched partially by

Algorithm 5 on the following page. The resulting IDD S' is constructed recursively by traversing depth-first both source graphs and creating new edges and nodes in a reduced manner, resulting from the interval mapping application by maintaining the respective graph structures. This computation is always based on reduced IDDs. Hence, no unification is necessary. 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 image computation is stopped at this point, continuing with the next edge. Omitted layers in the transition relation IMD are implicitly treated as "dummy" mapping functions returning their argument interval without changes. Omitted IDD layers implying quasi-obsolete nodes are treated separately. Only those edges have to be concerned not ending in the 0-terminal node.

As a memory function, a hash-based cache table of already computed results for pairs of v and w is maintained—omitted in Algorithm 5 on the next page—such that an included result may be returned without further computation, analogously to the *ITE* computation described in Section 3.2.3.2.

- **Backward mapping**
  As described above, the mapping functions are used for forward traversal during image computation with $Im(\mathbf{S}, \mathbf{T})$, e. g., for reachability analysis where time proceeds in forward direction. In order to perform symbolic model checking using $PreIm(\mathbf{S}, \mathbf{T})$ and $BackIm(\mathbf{S}, \mathbf{T})$, the reverse direction is needed. Thus, some kind of inverse mapping functions have to be used. For IMDs, depending on the structure of the mapping functions, this inversion is not always possible because the function result has to be representable by an interval. However, PADs have valid inversions as described below.

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

3.2.5.3 Image Computation With PADs

Image computation here is demonstrated with an example. As mentioned above, PADs are dedicated to performing image computation especially for process networks and related models of computation because

Alg. 5. Forward mapping for image computation

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

the state distance combined with the respective firing condition may be stored more efficiently than many state pairs. The state distance between two system states x and x' is defined as $\Delta x = x' - x$ (note that the variables involved are vectors). Thus, according to equation (3.2) on page 44, the transition relation $\mathbf{T}$ may be described as the characteristic function

$$\hat{t}(x, \Delta x) = \begin{cases} 1 & \text{if } (x, x + \Delta x) \in \mathbf{T} \\ 0 & \text{otherwise} \end{cases} . \tag{3.7}$$

In Figure 32 b) on page 66, only shift functions f_+ are used as mapping functions. In the case of buffer contents limited to non-negative numbers, the predicate intervals must ensure that the resulting state variables x'_i after a transition cannot 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. As shown in Section 4.3, even clock value changes such as uniform clock increase and clock reset can be represented by PADs.

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

Figure 33 on the following page shows the application of the image operator and its result. In the left-hand part, for a better understanding an intermediate IDD is depicted that is not really built during the computation but is more 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-hand part of Figure 33 on the next page—note that for efficiency *mapForward*(v, w) is implemented without the need for unification. The cut edges labeled with empty intervals represent such locations where the recursion of *mapForward*(v, w) is stopped due to an empty interval resulting.

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

Def. 3.2.13: (Inverse PAD mapping functions) *The inverse shift function is denoted as*

$$f_+^{-1}(\mathbf{I}) = (\mathbf{I} - \mathbf{I}_A) \cap \mathbf{I}_P$$

and the inverse assign function as

$$f_=^{-1}(\mathbf{I}) = \begin{cases} \mathbf{I}_P & \textit{if } \mathbf{I} \cap \mathbf{I}_A \neq \emptyset \\ [\,] & \textit{otherwise} \end{cases} .$$

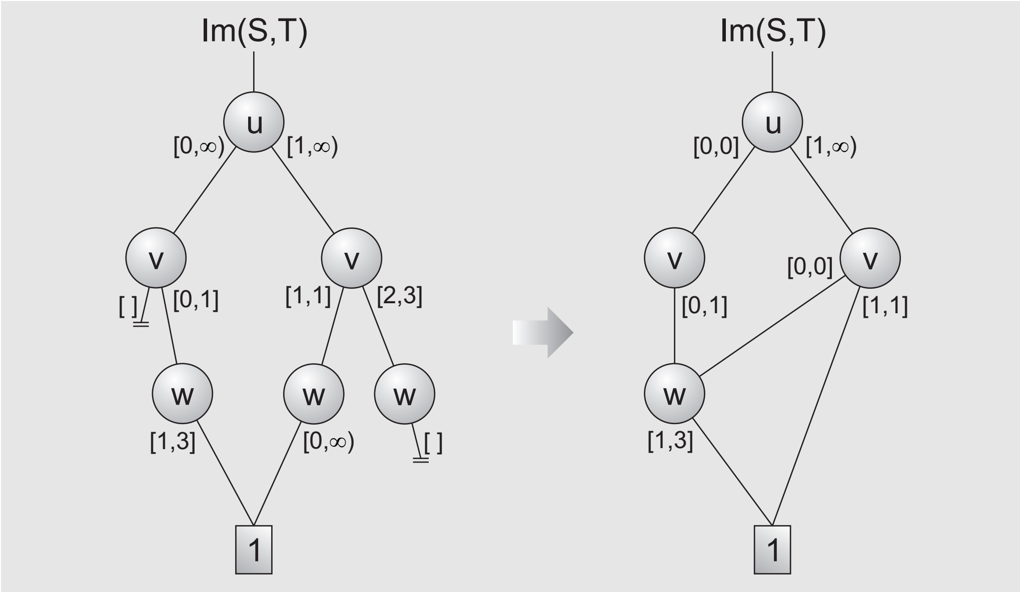

Figure 33. Computation of $Im(\mathbf{S}, \mathbf{T})$: IDD before and after unification.

3.2.6 Fuzzy Decision Diagrams

Based on interval decision diagrams, *fuzzy decision diagrams* (FuDDs) [SMTS99, SMTS00] have been developed. FuDDs can be used for representing, analyzing, and optimizing fuzzy rule bases. The variables involved are linguistic ones, and their values are fuzzy sets. FuDDs are comparable with multi-terminal IDDs of which the edges are annotated with fuzzy sets instead of crisp interval sets. Hence, in contrast to IDDs, several or all FuDD paths can be "active" at the same time but each of them more or less, according to the fuzzy philosophy. Using FuDDs, fuzzy if-then rule bases can be represented graphically.

Similarly to IDDs, fuzzy decision diagrams provide an efficient data structure for adequate representation, analysis, and optimization of rule bases. Moreover, FuDDs enable checking for completeness and consistency of a rule base. Ordered as well as free[6] FuDDs are of importance for rule base optimization. Moreover, hierarchical rule bases can be represented which implies the chaining of rules. For restricted classes of fuzzy rule bases, efficient algorithms for optimization and evaluation or rule bases exist.

[6]In free DDs, the variables along two paths can have different orderings.

4

Formal Verification

THE ever growing complexity of today's systems in the emerging fields of embedded computing and communications has seriously complicated or almost compromised the design of appliances with completely flawless behavior. However, for several reasons providing a correct design is a fundamental issue in many application areas. Roughly speaking, formal verification aims at two goals: saving lives and saving money.

4.1 Background

4.1.1 Why Formal Verification?

Saving lives, the first point, is to certify the correctness of a design in all possible circumstances. Especially for safety-critical applications such as those in traffic control, medical engineering, or avionics, simulation often is not sufficient to guarantee the correctness of a technical system's model because simulation times become prohibitive and the coverage of errors becomes poor.

The second point—saving money—is of economic nature. Detecting and eliminating bugs as early in the design cycle as possible—to prevent errors from being propagated from the conceptual into more detailed levels—has become a necessity as the 1994 Pentium® FDIV[1] bug showed which cost Intel Corp. about a half billion dollars [Hof95]. Moreover, simulation times of days or even weeks may be too expensive or not acceptable because of time-to-market constraints.

[1] FDIV: *floating-point divison*

Hence, rapid, highly automated, and easy-to-use tools for debugging support are required which can be employed at the earliest stages of the design process—when design errors are cheaper and easier to be fixed—and hence preferably on a high level.

In general, simulation cannot consider the entire execution behavior of a technical system's model but only one or a certain number of execution traces. For a non-determinate execution behavior—which is the rule with regard to the reactive nature of many embedded systems—this is not sufficient because an arbitrary number of faulty execution traces may remain undetected. Simulation cannot be completely exhaustive for complex designs.

Complementing or even substituting simulation, formal methods are employed to verify the system behavior and to prove temporal properties. The main characteristic of formal verification methods is that they provide sort of a mathematical proof of the correctness of a system model with regard to a given specification. Thus, all possible execution traces of the model have to be encompassed, at least implicitly. This concept can be compared with an exhaustive case analysis. The specification can be given, for instance, by means of a similar model to be compared with or by means of a set of logic formulae describing the desired system behavior.

The above circumstances have forced and strongly influenced development and application of various formal verification tools in several domains. Essentially, their industrial use emerged after the 1990 breakthrough of the promising symbolic model checking[2] approach [BCMD90, BCM$^+$90]. Today, a variety of both academic and commercial formal verification tools is available that have been applied successfully even to fairly complex real-world systems. Examples are SMV [McM93] by Carnegie Mellon University/Cadence Design Systems, Inc., TWISTER™ by Verysys Design Automation, Inc., or AFFIRMA™ FORMALCHECK by Cadence Design Systems, Inc.

4.1.2 Classification

Formal verification approaches often are divided into two categories. On the one hand, there are *proof-based* [Lon93] or *proof-theoretic* [Eme94] methods which emphasize the construction of a proof of correctness via axioms and inference rules for the specification language used [Hu95]. Examples are proof checkers, term-rewriting systems, and theorem provers. On the other hand, there are *state-exploration* [Lon93] or *model-theoretic* [Eme94] methods that search the state space of an implementation to check that it satisfies its specification.

In this work, only the state-exploration category is treated. Disadvantages of proof-based approaches are that they remain very time-

[2]Some of its main inventors (Bryant, Clarke, Emerson, and McMillan) received the 1999 ACM Kanellakis Theory and Practice Award for this reason.

consuming and often require a great deal of manual intervention. Hence, the user must cleverly guide the proof process which requires large amounts of experience, conscientiousness, and interaction. In contrast, state-exploration methods often are sufficiently fast and fully automated. Furthermore, they can be employed for debugging purposes as often a counter-example is provided in case of a detected bug. This means that if the model investigated does not satisfy some specification, an execution trace is produced that shows the reason for the discrepancy. This is particularly useful in finding and eliminating errors in complex systems.

An example of state-exploration methods is *sequential equivalence checking* [CBM89a] of state transition systems which proves whether or not two versions of a design behave identically. Equivalence checking can be used to ensure a complete exhaustive verification of a model after each design step. Both modifications by hand as well as tool-based changes, e. g., those by synthesis, can be checked. The sequential equivalence checker proves that there are no input sequences which may produce different output on the two designs. No assumptions are made about any correlation of the internal design structures. Sequential equivalence checking can be performed by reachability analysis of a product automaton representing both designs in parallel, where the reachability of unequal output states is checked while exploring all possible input scenarios.

In contrast to sequential equivalence checking, where the behavior of the "golden model"—the reference model—is specified implicitly in terms of its (partial) realization, model checking solves the problem of proving that a design model always exhibits some behavior specified explicitly. This state-exploration formal verification approach is explained in this chapter, applied to untimed process networks in Section 4.2 and to timed automata as well as the timed *FunState* model in Section 4.3. As an example of underlying data structures, especially interval diagrams are used. This approach is compared with conventional approaches with some experimental results.

4.2 Symbolic Model Checking

Symbolic model checking enables the verification of certain temporal properties of state transition systems, where the explicit construction of an entire reachability graph is avoided by implicitly describing it using symbolic representations. First, the conventional model checking approach—which does not make use of symbolic techniques—is briefly explained. Its symbolic form is treated later on.

4.2.1 Model Checking

Model checking was invented by Clarke and Emerson [CE81b] and independently by Quielle and Sifakis [QS81]. Specifications are expressed in a *propositional* temporal logic, while the system being verified is modeled as a state-transition system. Model checking explores the entire reachability graph of the system and checks whether or not the correctness property being checked is satisfied by the system model. The model checking algorithm propagates formulae in the reachability graph until a fixpoint is reached.

Commonly, the propositional, branching-time temporal logic CTL (*computation tree logic*) [CE81a, CE81b][3] is used for specifying the desired system properties [McM93]. CTL formulae describe properties of *computation paths* which are infinite sequences of states encountered by a system during some sequential execution [BCMD90]. In addition to atomic propositions and Boolean connectives, CTL consists of path quantifiers A (*for all paths*) and E (*there exists a path*), followed by one of the common temporal operators F (*eventually*), G (*always*), U (*until*), and X (*next time*).

The temporal operator X indicates a condition that holds in the next state of a computation, hence in the immediate successor of the actual state. G denotes a property that holds globally in all states of a computation path. F denotes a property that holds sometime in the future. $f \mathsf{U} g$ holds of a computation path if this contains a state s where g holds and if f holds in all the states preceding s. A temporal operator prefixed by the universal path quantifier A indicates that the temporal property must hold over all possible computation paths beginning in the current state, while the existential path quantifier E indicates that the property must hold along some computation path.

Def. 4.2.1: (Computation tree logic) *The formal syntax of* CTL *is defined as follows [BCL+94]:*

1. *Every atomic proposition is a formula in* CTL*, and*

2. *if f and g are* CTL *formulae, then so are $\neg f$, $f \vee g$, $\mathsf{EX}\, f$, $\mathsf{E}\,[f\ \mathsf{U}\ g]$, and $\mathsf{EG}\, f$.*

Due to the duality between universal and existential quantification, universal path quantifiers can be treated as syntactical abbreviations. The remaining operators are derived according to the following rules:

$$
\begin{aligned}
f \wedge g &\equiv \neg(\neg f \vee \neg g),\\
\mathsf{AX}\, f &\equiv \neg \mathsf{EX}\, \neg f,\\
\mathsf{EF}\, f &\equiv \mathsf{E}\,[\mathit{true}\ \mathsf{U}\ f],\\
\mathsf{AF}\, f &\equiv \neg \mathsf{EG}\, \neg f,\\
\mathsf{AG}\, f &\equiv \neg \mathsf{EF}\, \neg f,\\
\mathsf{A}\,[f\ \mathsf{U}\ g] &\equiv \neg \mathsf{E}\,[\neg g\ \mathsf{U}\ \neg f \wedge \neg g] \wedge \neg \mathsf{EG}\, \neg g.
\end{aligned}
$$

[3] CTL actually is a simple subset of the logics described there.

The CTL semantics is defined analogously to [BCL+94]. Here, only some example formulae are given for illustration. For instance, AG f is true in a state if for all paths starting from that state, f is always true. Hence, this describes an invariant. EF f is true in a state if there exists a path starting in that state which eventually leads to a state where f is true. This corresponds to reachability. CTL can be used to specify more complex properties such as safety and liveness properties[4]. Some example CTL specifications are:

- $\neg$EF *error* specifies that it is impossible to somehow reach an undesired state labeled with *error*,
- AG [$req \implies$ AF ack] describes that if a request *req* occurs, then it will be eventually acknowledged by *ack*,
- AG AF *enabled* means that *enabled* holds infinitely often on every computation path, and
- AG EF *restart* that the *restart* state is always reachable somehow.

Model checking conventionally is performed by repeatedly traversing the reachability graph of the system model. This is done within a depth-first search of the graph by labeling states with satisfied subformulae of the entire formula being verified. For instance, if a state is annotated with a formula f, then it can also be labeled with AF f (because AF f is satisfied even if f holds only in the current state). If f does not hold in this state, the state can be labeled with AF f only if all its direct successor states eventually become annotated with AF f. This is checked within the mentioned graph search. Nested formulae are evaluated from inner to outer subformulae.

The relation between the number of states in the model to the number of variables of the system typically is exponential—which coined the term state explosion problem. This occurs particularly if the system being verified has many components that can make transitions in parallel. In general, the complexity of CTL model checking is proportional to the product of the number of states in the model and the length of the formula being checked. Using binary decision diagrams as efficient, symbolic representations of state sets and transition relation turned out to be a promising approach for model checking of finite state machines.

4.2.2 The Symbolic Approach

In order to verify CTL formulae using symbolic techniques, image computation as described in Section 3.1.3 and Boolean operations are used in a fixpoint computation. The following explanations are taken from [BCMD90].

[4] *Safety* intuitively means that something bad does not happen, while *liveness* means that something good will eventually happen.

The CTL formula $\mathsf{E}\,[f \mathbin{\mathsf{U}} g]$ holds if either g is true in the current state, or if f is true in the current state and there exists a successor state in which $\mathsf{E}\,[f \mathbin{\mathsf{U}} g]$ is true. This can be expressed by the *least fixpoint* of a condition Z in the expression

$$Z = g \vee (f \wedge \mathsf{EX}\ Z).$$

This fixpoint is evaluated iteratively by setting Z to *false* initially and then evaluating the above expression repeatedly until a fixpoint is reached. Z can be denoted as a Boolean expression on state variables. Intuitively, this algorithm finds the set of states which satisfy g, then augments this set by adding the set of states which satisfy f and have successors in the state set represented by Z, and repeats this procedure until the set remains unchanged.

The formula $\mathsf{EG}\, f$ states that f is true in the current state, and $\mathsf{EG}\, f$ is true in some successor state. This condition is the *greatest fixpoint* of Z in the expression

$$Z = f \wedge \mathsf{EX}\ Z.$$

This fixpoint is computed by setting Z to *true* and repeatedly evaluating the above expression until a fixpoint is reached. This can be compared with beginning with the set of states in which f is true, then removing all those states which have no successors in the set until the set remains unchanged.

Using the above means, every CTL formula can be computed. Merely the computation of $\mathsf{EX}\, f$, on which $\mathsf{E}\,[f \mathbin{\mathsf{U}} g]$ and $\mathsf{EG}\, f$ are based, remains to be described. This is where image computation comes into play because $\mathsf{EX}\, s$ directly corresponds to the pre-image operator $\mathit{PreIm}(\mathbf{S}, \mathbf{T})$ where the characteristic term s represents the state set $\mathbf{S}$. Alternatively, following the quantification duality the correspondance of $\mathsf{AX}\, s$ and $\mathit{BackIm}(\mathbf{S}, \mathbf{T})$ could be used instead. This can lead to more efficient implementations which use dedicated algorithms instead of the computationally more expensive equivalences complementing Definition 4.2.1 on page 74.

The most frequently employed form of symbolic representation are BDDs and their derivatives. In order to perform conventional symbolic model checking, the investigated system is described using BDDs to represent its transition relation and the state sets. As mentioned in Section 3.2, for process networks this is not a convenient form of representation for several reasons. On this matter, Hu [Hu95] stated

> "high-level verification is hard for BDDs".

Advantages of interval diagram techniques are:

- Using IMDs and IDDs for the representation of transition relation and state sets, respectively, avoids some undesirable limitations of BDDs and binary coding.

- The introduced image computation is 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 represented by one IDD or IMD node.

Image computation using interval diagrams is performed as described in Section 3.2.5. Boolean operators such as $\vee$, $\wedge$, and $\neg$ can be computed using the *ITE* operator introduced in Section 3.2.3.2. Basically, symbolic model checking based on interval diagrams consists of four stages:

1. formulation of the specification as verification goal by means of a CTL formula,
2. representation of the state transition relation in form of an interval mapping diagram,
3. representation of state sets as interval decision diagrams, and
4. applying to IDDs Boolean operations and image computation using the above IMD until a fixpoint is reached, as described above according to the CTL specification.

4.2.3 Process Networks

Symbolic model checking of process networks using interval diagram techniques has been introduced in [ST98]. Only the quantitative system behavior is considered, i. e., only the number of tokens in each buffer, not their values.

4.2.3.1 Formal Verification Issues

Examples of conventional verification goals are the mutual exclusion of processes or the guaranteed acknowledgement of requests. Apart from such conventional issues, applications assisting in scheduling are possible. Boundedness can be ascertained either by computing the set of reachable states or by checking CTL formulae on the contents of buffers. For termination and deadlocks, respective CTL formulae may be formed easily. Additionally, the effect of certain scheduling policies on these measures may be investigated or improved. Deadlocks in artificially bounded process networks or inherent bounds may be detected. In this way, optimal schedules may be confirmed or even developed by determining least bounds and thus optimal static capacity limits for scheduling, constraining the required memory.

4.2.3.2 Modeling Process Networks

The behavior of many process networks may be described by decomposing the transition behavior of each process into atomic transitions, changing the internal state of the process and consuming and producing tokens in dependence on the internal state. For this decomposition, it is necessary that the process behavior can be described by means of a finite state machine. Non-blocking read or blocking write semantics may be represented, too. Even non-determinate models with multi-reader and multi-writer buffers such as, e. g., Petri nets are verifiable using IDDs and IMDs.

Each path in the transition relation IMD describes one possible state transition. The mapping functions along the path represent enabling conditions and the corresponding state variable changes. The transition is enabled if all conditions along the path are satisfied. Analogously to computation graphs [KM66], a threshold different from the consumption rate may be specified as shown in Section 3.2.5.1. Non-determinate consumption rates can easily be considered as intervals—introducing an additional degree of non-determinism. While changes of buffer contents are described using shift functions f_+, assign functions $f_=$ are used for internal state changes. The state variables are of either infinite domain—representing contents of unbounded buffers—or of finite domain—describing internal process states or bounded buffers.

4.2.3.3 Variable Ordering

As mentioned in Section 3.2.2.3, the variable ordering of an IDD has a significant impact on the number of its nodes. Decision diagrams in general are very sensitive to the variable ordering imposed which can make their size vary from linear to even exponential. In the following, a simple topology-based ordering heuristic is sketched which exploits the structure of the process network being represented. This approach proved feasible for practical applications.

For decision diagram applications, mostly it is advantageous to keep together variables representing components in close structural proximity. Hence, strongly connected processes and queues should be represented by DD variables at layers close to each other. The greedy, constructive strategy described below results in a variable ordering taking this into account.

In order to determine a favorable variable ordering, first the process with the largest number of connected queues is determined. The state variables of this process form the basis of the entire variable ordering and are placed in the center of the IDD. Then, the two queues with the largest number of edges to and from the chosen process are determined. Their corresponding variables are placed above and below the process state variables, respectively.

The following procedure is repeated until all queues have been considered. The construction of the variable ordering is done by placing variables alternately for each iteration above and below the actual variable ordering. Among all queues not considered yet, that queue with the largest number of processes connecting it with any queue in the actual variable ordering is determined. First, the state variables corresponding to the processes between the chosen queue and the actual ordering are appended to the ordering. Then, the queue state variable is inserted above or below, respectively.

4.2.4 Petri Nets

Interval diagrams first have been applied to symbolic model checking of Petri nets in [ST99a].

4.2.4.1 Formal Verification Issues

A few questions out of the wide variety of system properties being 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, RCP95] and are straightforwardly adaptable to IDDs, e. g., for

- deadlock freeness and diverse levels of liveness,
- boundedness, persistence, and home state property[5].

4.2.4.2 Modeling Petri Nets

To describe the principal use of IDDs and IMDs for symbolic model checking of Petri nets, Figure 34 shows an example Petri net [Jan83]. The Petri net places are of infinite capacity.

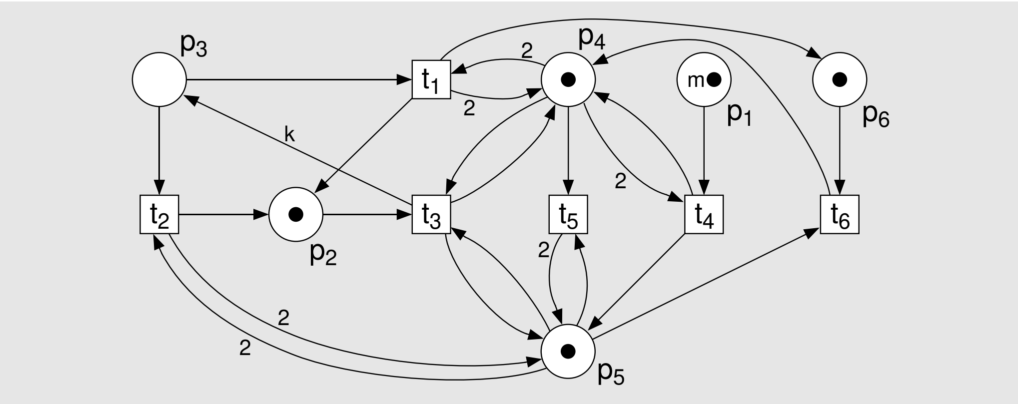

Figure 34. Example Petri net.

Figure 35 on the following page shows the reduced IMD of the corresponding transition relation. Singleton action intervals containing only one element are abbreviated by this element. Omitted predicate or action intervals stand for non-existing predicates or value changes, respectively.

Each path in the transition relation IMD describes one of the Petri net transitions. The paths in the example IMD correspond to the Petri net transitions t_i ordered from left to right. The mapping functions along

[5]Definition and meaning of these terms are given in [Mur89], for instance.

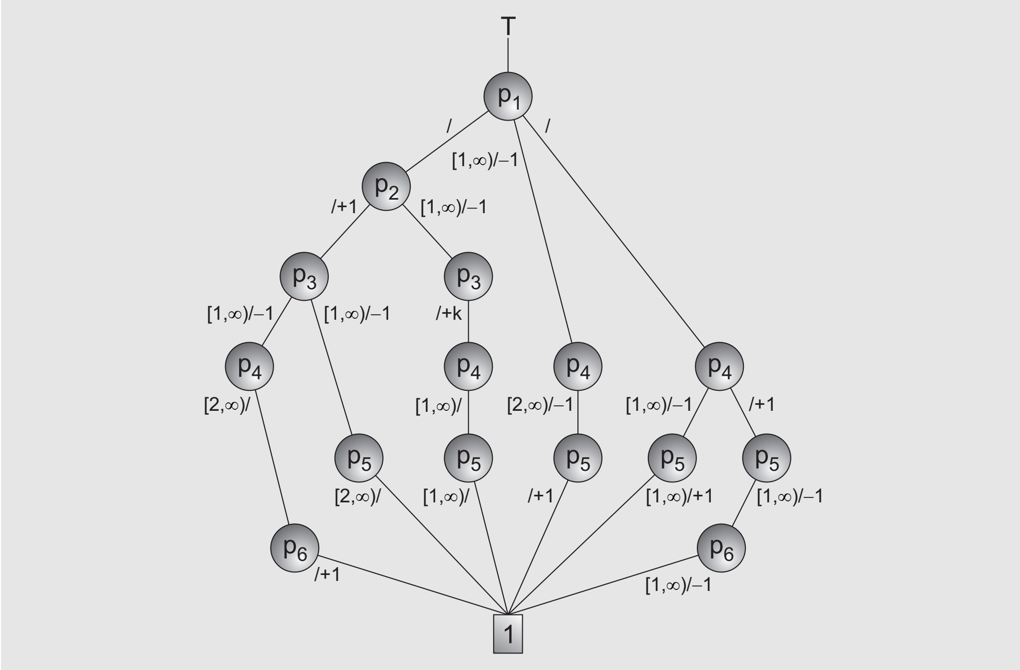

Figure 35. Transition relation IMD of Petri net.

a path describe the state distances and their corresponding firing conditions, i. e., the state variable changes that are the consequence of the execution of the respective transition if all its predicates are satisfied. For instance, if $p_3 \in [1, \infty)$ and $p_4 \in [2, \infty)$ hold, transition t_1 is enabled to fire by decreasing p_3 and at the same time increasing p_2 and p_6 by 1, while p_1, p_4, and p_5 remain unchanged.

Since predicate interval and action interval are independent of each other, even self-loop transitions and inhibitor arcs can be modeled. While unbounded state spaces with state variable domains of $[0, \infty)$ are favorable for describing Petri nets with infinite place capacities, an explicit bound may exist, e. g., as an implementational or conceptual limit of the size of the modeled element. In such case, the introduction of bounded state spaces is useful to avoid the necessity of adding restricting places to the original Petri net. For instance, the state variable domain of an element of capacity k is reduced to $[0, k]$, implying the use of the respective modified predicate intervals inhibiting that the number of tokens in a place may exceed its bound.

An 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 introducing lower and upper bounds, the set of pairs becomes finite and so does the IDD.

4.2.5 FunState

There are many different purposes of formal verification of an internal design representation. Instead of dealing directly with the system specification, properties of a representation can be checked which is the basis for design steps such as scheduling, binding, and allocation. It is possible to verify certain properties of a partially completed design. For example, one may want to prove that a chosen schedule results in a deadlock-free implementation or requires only a bounded amount of memory.

In this section, the formal verification approach introduced in [ST98] is extended from simple process networks towards the more complex *FunState* model containing both finite-state control components and infinite-state dataflow queues. The strategy proposed is based on the *FunState* representation in form of a regular state machine, see Section 2.2.4. Of course, during the verification process, the state space is not enumerated explicitly.

4.2.5.1 Formal Verification Issues

Consider the example *FunState* model in Figure 7 on page 19. In order to show that q_2 may never contain more than four tokens, the formula AG [$q_2\# \leq 4$] is checked. Since this formula evaluates to *true*, it is proven that the memory required by q_2 is bounded by 4. Another simple example is the formula AG EF [$q_1\# \geq 1$] which means that it is always possible to reach a system state which allows f_2 to be executed. Thus, this kind of formulae can be used for proving the absence of deadlocks.

For this purpose, the atomic propositions of CTL as previously defined are extended to first-order expressions such as a condition on a queue content, e. g., $q\# \geq 3$, or a state variable. Hence, the CTL-based logic used in this context is a *restricted first-order* temporal logic [Eme94] instead of a propositional one.

In summary, the symbolic model checking strategy presented here enables the efficient verification of certain temporal properties of state controlled process networks, where the explicit construction of an entire state transition graph is avoided by implicitly describing it using symbolic representations. Thus, using *FunState* to internally model a mixed hardware/software system enables its formal verification comprising the whole well-known area concerning the detection of errors in specification and implementation.

Further, formal verification may assist in the development of scheduling policies. The system model can be extended to describe a scheduling policy as well, of which the behavior then is verified together with the system model. Thus, properties such as the correctness of a schedule may be affirmed by proving the boundedness of the required memory and the absence of artificial deadlocks [Par95] as described above.

Besides symbolic model checking, symbolic equivalence checking can be applied to *FunState* models. This is important especially for an internal design representation like *FunState* to prove, for instance, that a refinement step has been performed correctly and the resulting implementation still exhibits the desired, previously specified behavior.

4.2.5.2 The Untimed *FunState* Model

In the untimed *FunState* model, function latencies are neglected. Function executions are reduced to atomic actions affecting solely connected queues. The *FunState* model is treated analogously to the description in Section 4.2.3.2 for general process networks.

In the following, again interval diagrams are used as an example of symbolic representation. Consider the *FunState* model shown in Figure 36 a). Assume all function latencies be zero. Figure 36 b) shows the interval mapping diagram representing this transition behavior.

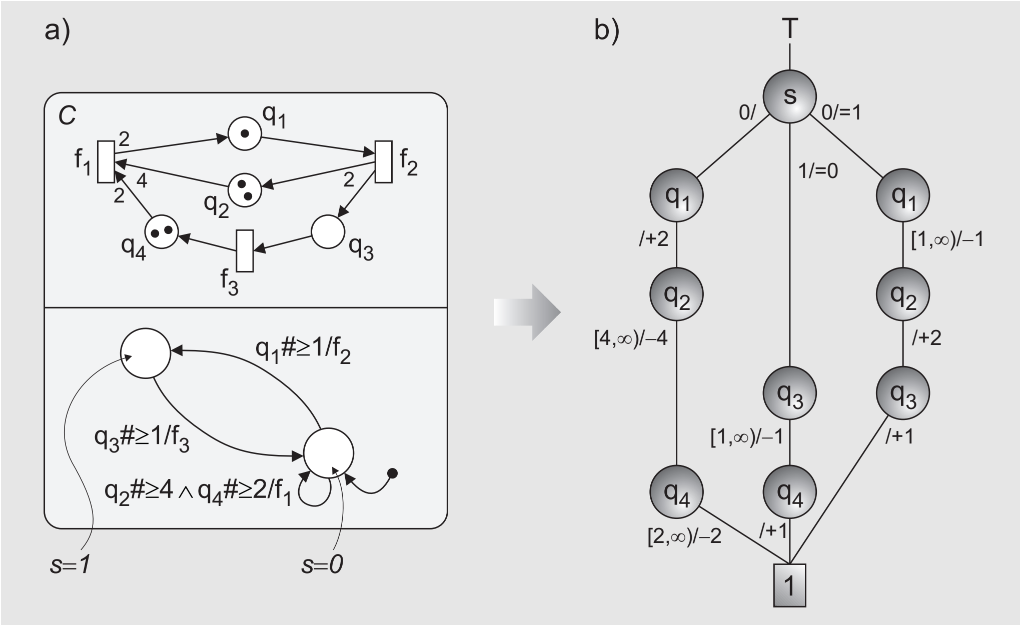

Figure 36. *FunState* example model and IMD representing untimed transition behavior.

4.2.5.3 The Timed *FunState* Model

Formal verification of the timed *FunState* model can be performed on two different levels of abstraction. On the one hand, the function execution times can be considered explicitly as specified, which is performed by combining the *FunState* model of computation with timed automata, as presented in Section 4.3.4.

On the other hand, function latencies can be abstracted by dividing them into zero and non-zero latencies only. Hence, the explicitly timed *FunState* model can be regarded as a refinement of the abstracted timed model, which in turn represents a superset of the execution behavior of the explicitly timed model.

Zero latency functions are treated analogously to functions in the untimed *FunState* model. In contrast, non-zero latency functions have to be modeled differently because during function execution, state machine transitions may occur and other functions may be invoked or terminated.

This is sketched in Figure 37. The steps of the operational semantics introduced in Section 2.2.3.4 are depicted over time together with a Gantt chart describing example function executions. Functions f_1 and f_2 have non-zero latency, while f_3 has zero latency. Note that at point in time t_2, function f_2 could have finalized together with f_1 if f_2's latency had been smaller.

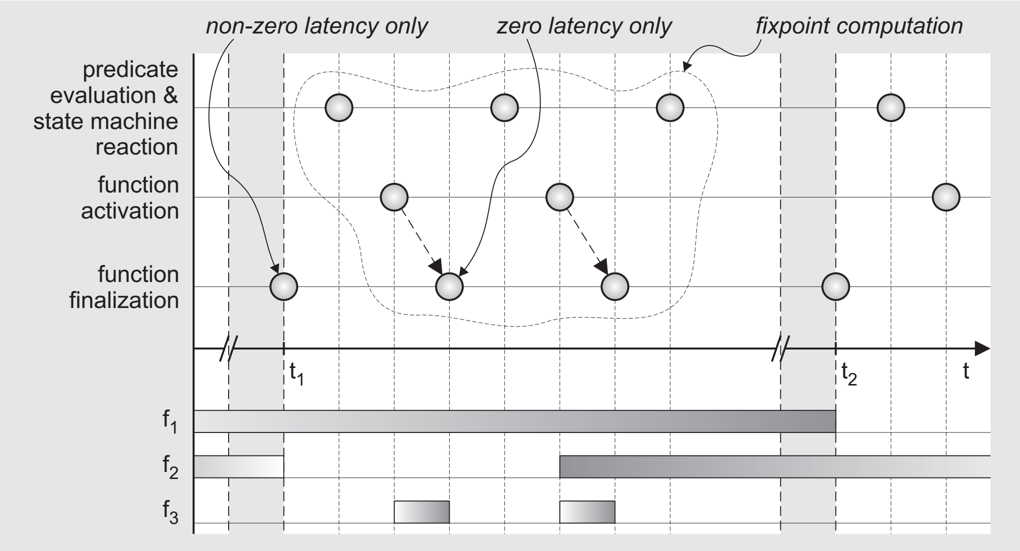

Figure 37. Abstracted timed execution behavior of *FunState*.

At each point in time, a fixpoint computation (similar to the delta cycle evaluation in VHDL[6]) is performed until no more transition predicates are satisfied. Only then, global time progresses. In order to avoid that a function in execution is activated again—which would correspond to multiple instantiation or pipelining of the function but is not permitted in the *FunState* model—, after reading its input tokens the function has to be blocked until its finalization. This can be done by either augmenting the function with an internal processing state or by refining it by two functions—e. g., one of zero latency, connected with the incoming edges, and the other one of non-zero latency, connected with the outgoing edges—and a queue of capacity 1 in between.

The transition behavior of the *FunState* model has to be subdivided into two stages. One stage iteratively performs predicate evaluation, state machine reaction, function activation, and function finalization of zero latency functions, in a fixpoint computation until the system state is stable. The other stage performs the function finalization of non-zero latency functions and hence implicitly the time progress. Both stages are applied alternately.

Likewise, image computation and transition relation are splitted into two parts. The first, reaction/activation stage is performed by applying a transition relation T_A until reaching a fixpoint. Another transition relation T_F represents the second stage of function finalization.

[6]VHDL: *VHSIC hardware description language*, VHSIC: *very high speed integrated circuit*

Assume the functions f_1 and f_2 of the *FunState* model in Figure 36 a) on page 82 be of non-zero latency and f_3 of zero latency. Figure 38 a) shows the interval mapping diagram representing T_A in the fixpoint computation of the transition relation. The IMD nodes labeled with p_1 and p_2 correspond to the internal processing states of the respective functions f_1 and f_2. p_i is 1 iff the respective function f_i is in execution. Function f_3 does not require such internal state because of its zero latency.

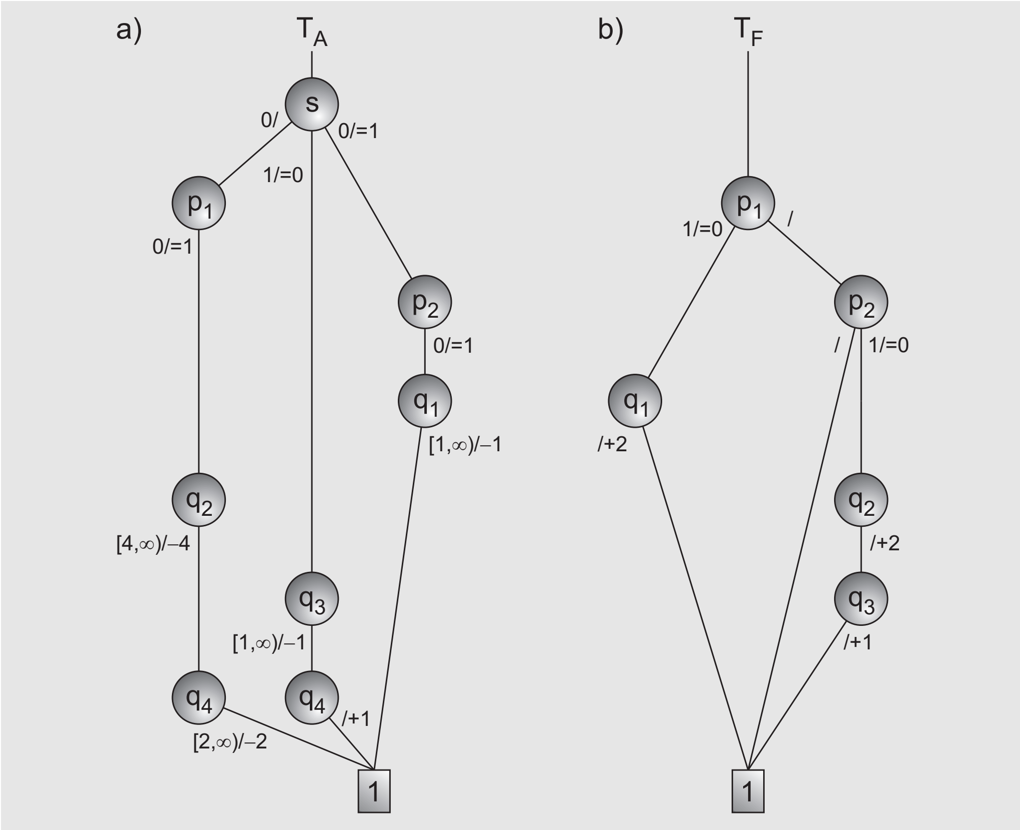

Figure 38. IMDs representing abstracted timed transition behavior.

By the IMD of T_A, functions f_1, f_2, and f_3 are activated, depending on the internal state s of the *FunState* component and on respective internal processing states. The zero-latency function f_3 is activated and finalized as well, within an atomic action. A function cannot be activated during its execution.

Figure 38 b) shows the IMD representing T_F which has to be used in a fixpoint computation as well. The IMD describes the finalization of f_1 and f_2 which can be performed only if the respective function is processing. Function finalization is independent of the *FunState* component's internal state. Note that the IMD of T_F—in contrast to that of T_A—contains a path not affecting the system state. This is necessary because possibly no function terminates at a given time. Although not apparently from Figure 37 on the preceding page, T_F has to be applied in a fixpoint computation as well because more than one function may be finalized at once.

For reachability analysis or sequential equivalence checking, the above described stages—each consisting of a separate fixpoint computation—simply are applied alternately. Hence, this results in a nested fixpoint computation due to the reachability iteration. CTL model checking becomes more difficult, though. The transition relation T_F has to be augmented by stronger predicates which ensure that exactly those states result from backward image computation which may occur in the transition fixpoint. Pre-fixpoint states must not be considered. Further, *fairness constraints*[7] have to be imposed to avoid that a function is executed infinitely long.

4.2.6 Empirical Results

In the following, the feasibility of the symbolic formal verification approach based on interval diagrams is shown with diverse experimental results. Interval diagram techniques are compared with a conventional BDD-based approach with regard to the computational resources required.

4.2.6.1 System Models

Several system models based on process networks with FIFO queues have been investigated which led to promising results. Starting with the initial state, the set of reachable states has been calculated as described in Section 3.1.1. Some results for different initial configurations are presented in the following, comparing IDDs and IMDs with BDDs. In the BDD version, the coding of the state variable values was directly binary. Investigations yielded promising results concerning the number of nodes and edges as well as the computation time.

In Figure 39 on the next page, the computation time T to determine the set of reachable states of a model of a flexible manufacturing system with automated guided vehicle [ABC+95] is depicted for increasing initial configurations n. The parameter n represents the initial number of tokens in a certain queue and hence is a measure of the size or the complexity of the model. Figure 40 on page 87 shows the size of the diagram representing the set of reachable states depending on the initial configuration n. For both criteria, IDDs and IMDs turn out to be significantly superior to the conventional approach using BDDs.

Figure 39 also shows the computation time using the symbolic model checker Cadence SMV which is based on BDDs. The same variable ordering as for IDDs and the standard optimization parameters have been used.

The implementation is not tuned for speed and does not contain common optimization techniques such as partitioned transition relations or sifting which are also applicable to interval diagrams. The memory overhead and the difference in the complexity of the operations (Boolean op-

[7] A fairness constraint ensures that in model checking only those execution paths are considered along which a given formula is satisfied infinitely often.

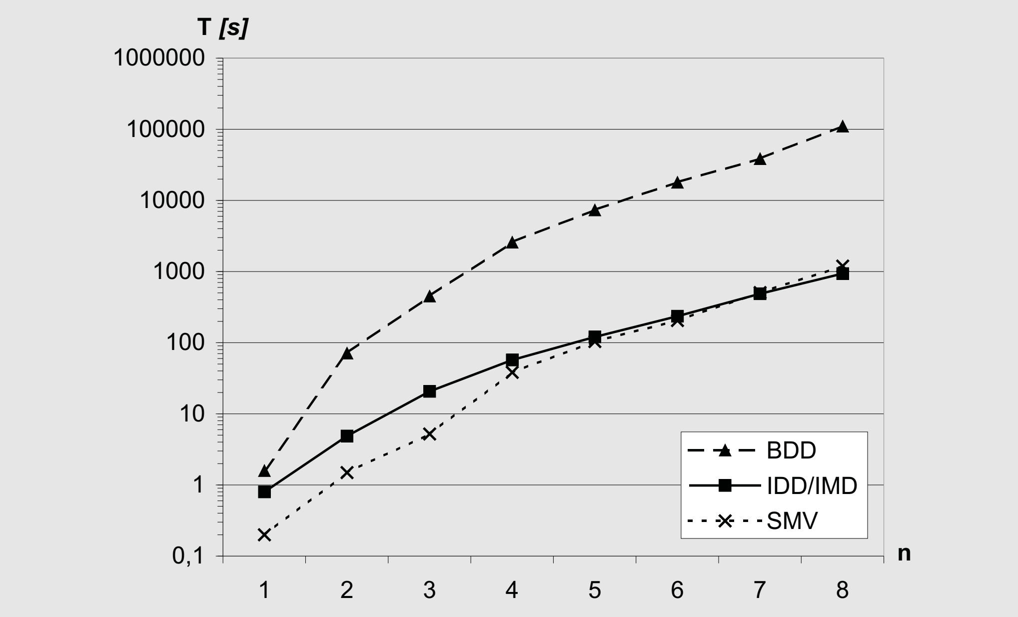

Figure 39. Computation time T for manufacturing system.

erations, image computation) of BDDs in comparison with interval diagrams are constant. Therefore, both implementations would profit in the same way from increasing the efficiency of the implementation and thus would scale similarly. For the implementation, the programming language Java has been used for experimental purposes. Current Java compilers and interpreters achieve implementation speeds which are about 5 to 10 times lower than those of C. Hence, equivalent platforms should shift the IDD/IMD computation time graph about up to one decade downward.

The BDD implementation of SMV is up to one or two decades faster than the BDD implementation used here, while interval diagrams are roughly as fast as SMV for large models. Due to the above arguments concerning the efficiency of the implementation, significant outperformings are expected even of the state-of-the-art model checker SMV about one or two decades, equivalently optimized implementations and further algorithmic efficiency improvements assumed.

Obviously, the memory required to store IDD nodes and edges is different in comparison with BDDs because IDD nodes can have more than only two edges and intervals are assigned to the edges. Figure 41 on page 88 shows the memory required for the diagram representing the set of reachable states depending on the initial configuration n.

The transition relation diagram size for the manufacturing system is compared for IMDs and BDDs in Table 2 on the facing page.

Figure 42 on page 89 and Figure 43 on page 90 show the computation time and the size of the state set diagram, respectively, for the reachability analysis of a readers and writers system accessing a common database.

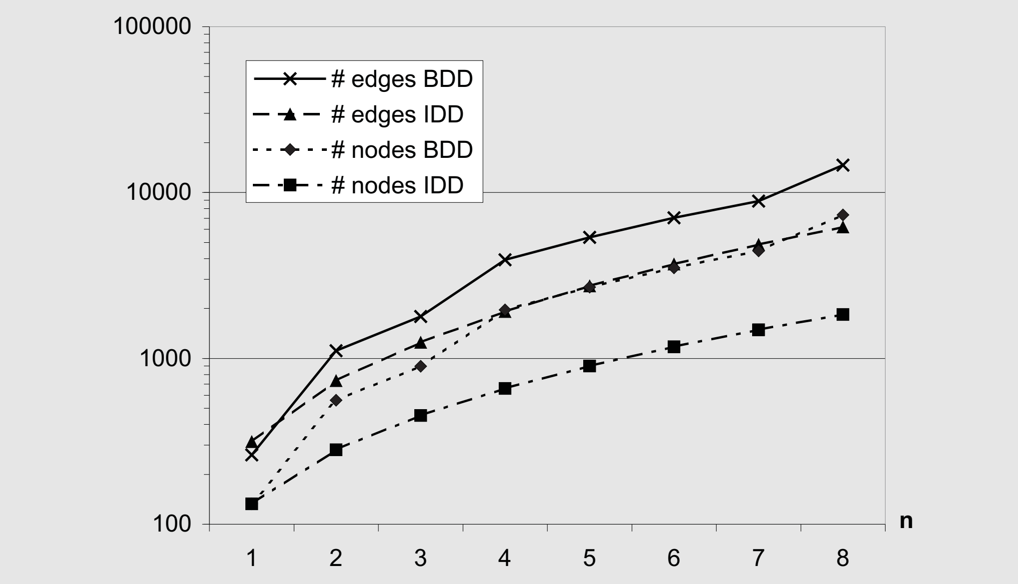

Figure 40. State set diagram size for manufacturing system.

Table 2. Size of transition relation diagram for manufacturing system.

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

In Table 3, the size of the transition relation diagram is compared for the readers and writers system. The significant reduction of the number of nodes in the transition relation IMD and in the state set IDD is obvious in comparison with the BDD equivalents. The IMD size is independent of the initial configuration, while the BDD size increases heavily.

Table 3. Size of transition relation diagram for readers and writers system.

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

In Figure 44 on page 91 and Figure 45 on page 92, the computation time and the state set diagram size, respectively, for a symmetric multi-server random polling system are shown.

Considering the computation time, BDDs additionally have one major disadvantage in contrast to IDDs and IMDs. As mentioned in Sec-

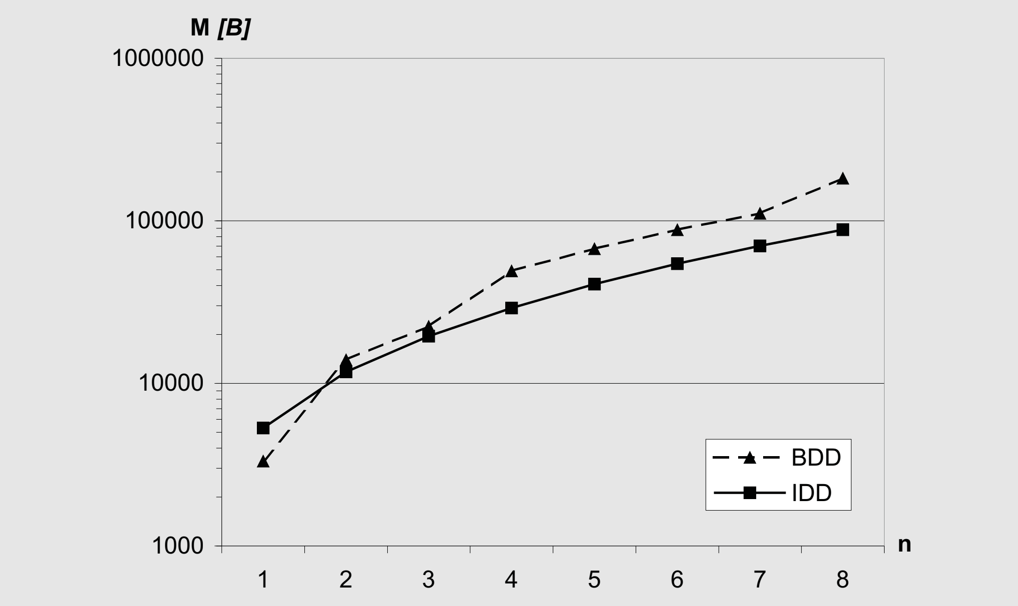

Figure 41. Memory requirements of state set diagram for manufacturing system.

tion 3.2.1.1, using BDDs requires an upper bound for the state variable values as they are binary coded. However, such a priori bound is not known in general. Using loose upper bounds causes substantial computation overhead—Table 4 and Table 5 on the next page show the dependence of the BDD size for the transition relation **T** and the set of reachable states **S** on the chosen coding length. In most cases, the alternative would be to increase the bounds incrementally—i. e., to add bits to the coding—until they are high enough but still tight. Each iteration of this "trial-and-error" method takes time not to be neglected, while no time is wasted using IDDs and IMDs because already the first run yields the final result.

Table 4. BDD size for polling system with $n = 15$.

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

4.2.6.2 Example Petri Net

Astonishingly, the Petri net in Figure 34 on page 79 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 n of place p_1. Depending on the values of those two parameters, the

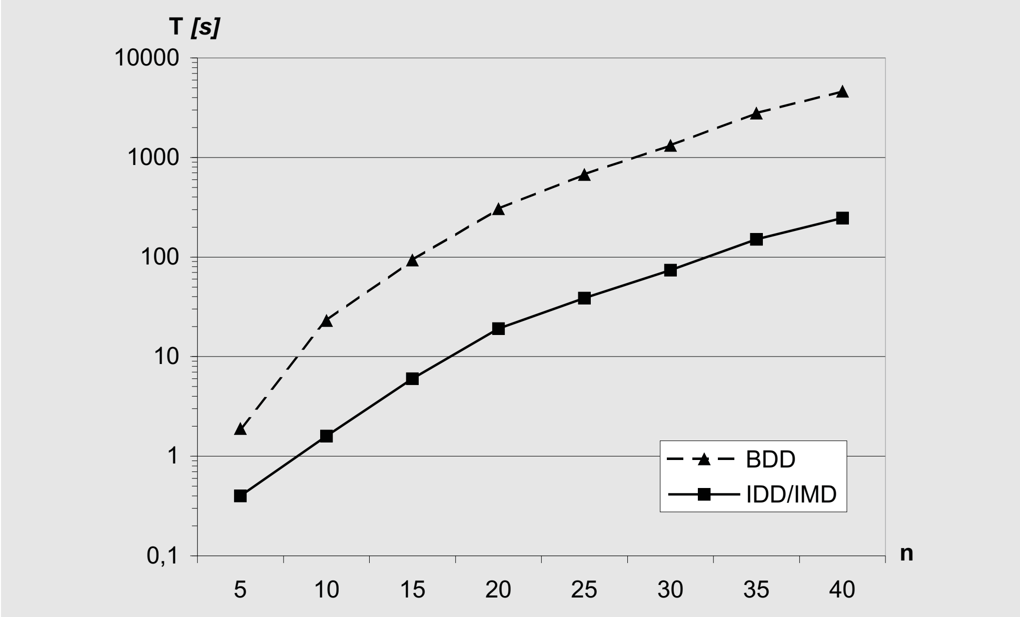

Figure 42. Computation time T for readers and writers system.

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:

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

The set of reachable states has been calculated by a series of image computations as described in [PRCB94]. Some results for $m = 1$ and $k = 3$ are shown in Table 5.

Table 5. BDD size for Petri net with $m = 1$ and $k = 3$.

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

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

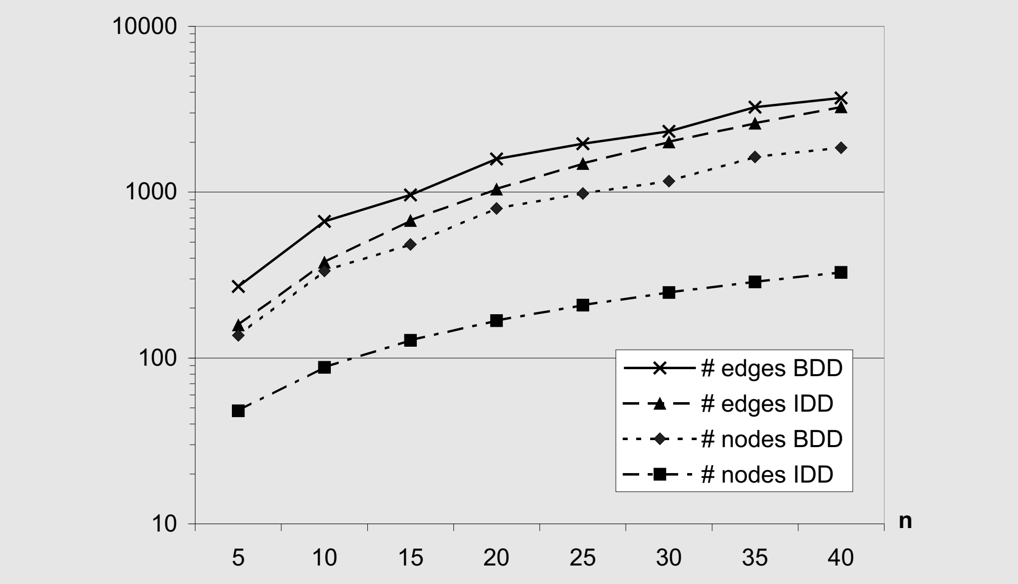

Figure 43. State set diagram size for readers and writers system.

length of the coding in the first BDD column of Table 5 on the preceding page suffices to represent all possible state variable values. 494 fixpoint iterations are necessary to ascertain the set of reachable states. Especially the reduction of the number of nodes in the transition relation IMD and in the state set IDD is significant.

For $m = k = 2$, the final result could not have been determined yet due to memory limitations. Figure 46 and Figure 47 on page 93 show the diagram size and the memory required during fixpoint computation, respectively, requiring a variable domain of $[0, 8191]$ and 156 layers for the transition relation BDD. Again, IDDs require greatly less nodes, edges, and memory than BDDs.

In the implementation used, 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 given 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 in order to minimize memory requirements and to reduce the length of the above-mentioned edge lists.

4.2.7 Further Related Work

An approach with aims similar to the above is the one of Godefroid and Long [GL96, GL99]. They verify system models—especially for lossy communication protocols—based on FIFO queues by binary coding the queue contents and representing them in form of *queue BDDs* (QBDDs).

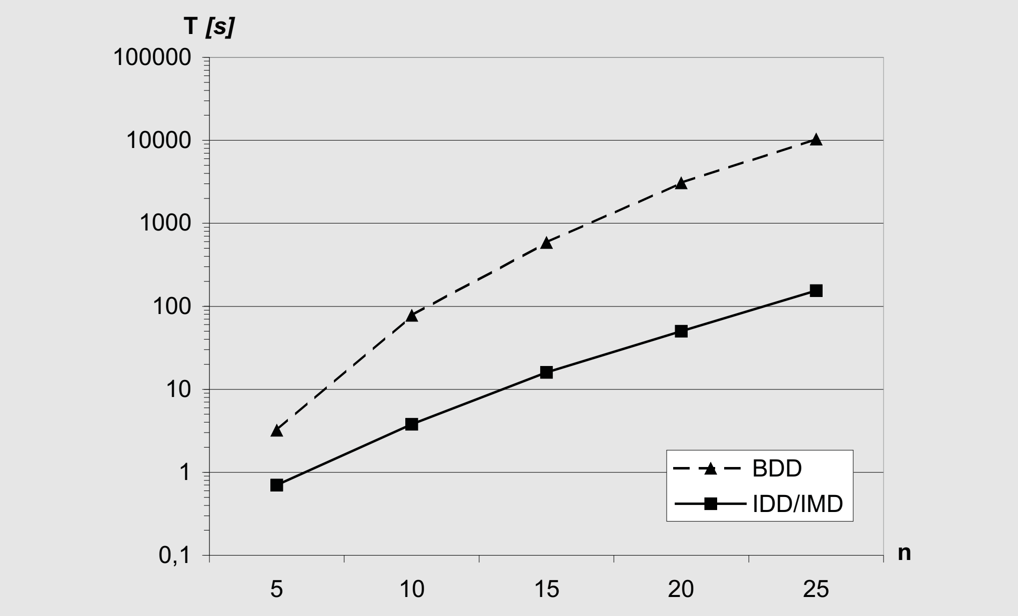

Figure 44. Computation time T for polling system.

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 enqueuing and dequeuing of single elements are treated. QBDDs are used for describing sets of system states but not the transition relation. The methods used for this require specialized enqueuing and dequeuing methods of which the possibility of being combined with conventional BDD techniques is not guaranteed.

Related work concerning symbolic model checking of Petri nets has been performed by Pastor, Roig, Cortadella *et al.* [PRCB94, RCP95]. Their focus is on *safe*, i. e., 1-bounded, Petri nets used for verifying speed independent circuits modeled by means of *signal transition graphs*. For this Petri net class, the binary BDD representation is adequate and allows direct conversions as no binary coding is necessary. Although a hint on possible extensions for k-bounded Petri nets is included in [PRCB94], this means no fundamental advance due to the reasons stated above. In [YHTM96], zero-suppressed BDDs are used instead of regular BDDs for speeding up the computation. General deficiencies of BDDs have been partially mitigated by applying novel encoding schemes to represent place contents [PC98, PCP99]. This is achieved by partially or completely transforming the Petri net into a set of finite state automata. [MC99] presents an MDD-based approach which exploits locality of transition firings to reduce both the number of iterations and the cost of each iteration during reachability analysis.

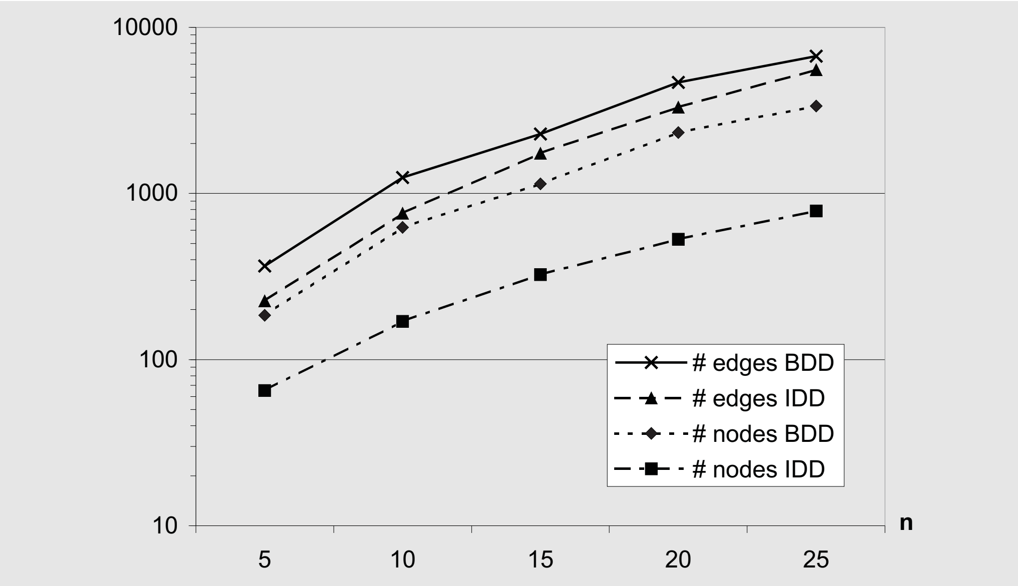

Figure 45. State set diagram size for polling system.

4.3 Symbolic Real-Time Verification

Conventional symbolic formal verification considers only the qualitative timed behavior of a system's mathematical model. Quantitative temporal properties however are neglected. In contrast, real-time formal verification methods provide the means of verifiying timing requirements on a model and of determining its quantitative temporal properties. For this purpose, the timed system behavior is modeled in fairly great detail using models of computation such as timed automata.

In this section, interval diagram techniques are suggested for formal verification of real-time systems modeled by means of timed automata. The approach is used for performing both timed reachability analysis and real-time symbolic model checking. The methods necessary for the approach are presented and its results are compared with another, similar verification technique—achieving a speedup of 7 and more [Str99]. Finally, timed automata are combined with the *FunState* model of computation to enable symbolic formal verification of the timed *FunState* model.

4.3.1 Preliminaries

In order to perform formal verification of real-time systems, the system first has to be transformed into a suitable model of computation the verification is based upon. In [MP95], Maler and Pnueli describe a possible application of timed automata for modeling asynchronous circuits. The digital circuit considered is transformed into a timed automaton reflecting timing behavior aspects such as uncertainties in gate delays and input arrival times. The constructed automaton can be used for formal verification and timing analysis. There exist several classes of techniques to perform this task.

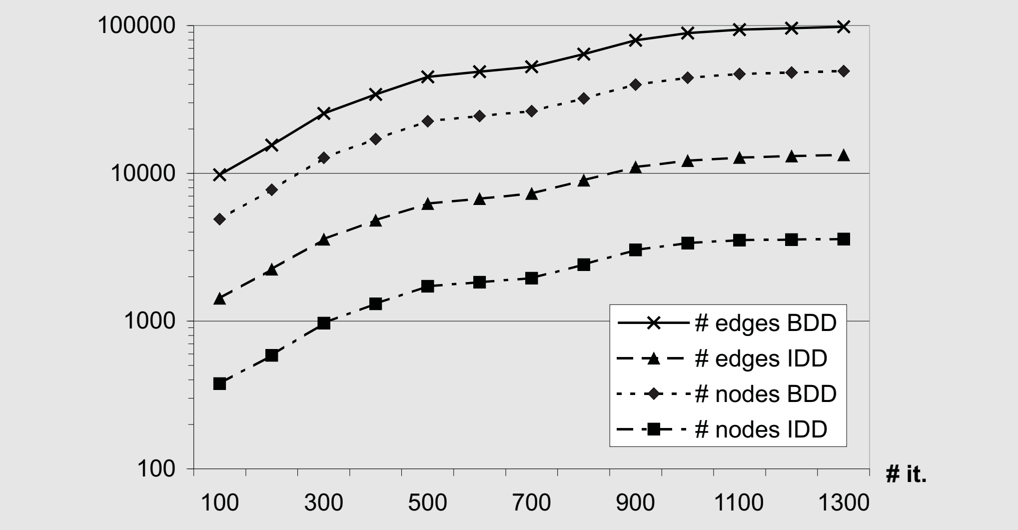

Figure 46. Diagram size during fixpoint computation for Petri net.

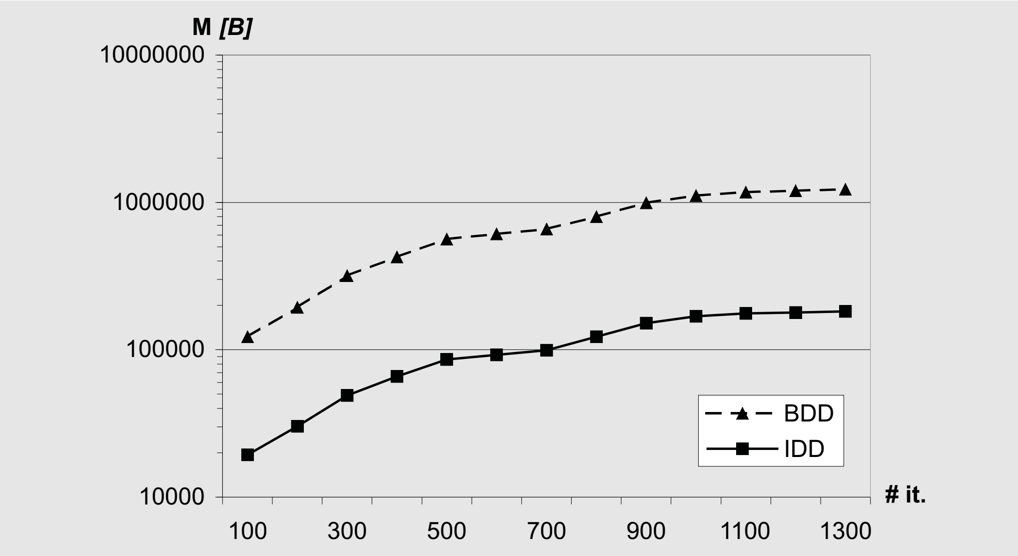

Figure 47. Memory requirements during fixpoint computation for Petri net.

- Timed reachability analysis determines the set of states reachable from a given set of initial states. As mentioned in Section 2.3, the system state here comprises a discrete state and some timing information represented by clocks. Many diverse verification tasks such as invariant or trace inclusion checking can be mapped onto reachability analysis. This allows proving that, e. g., an error state may never be reached or certain timing constraints are always satisfied.

- Besides reachability analysis, formal verification comprises real-time symbolic model checking, i. e., checking the satisfaction of timing properties expressed in one of various real-time temporal logics—most of them CTL derivatives—such as *real-time* CTL (RTCTL) [EMSS89], *timed* CTL (TCTL) [ACD90], *quantized* CTL (QCTL) [FGK96], or *clocked* CTL

(CCTL) [RK97a]. Complex properties such as the guaranteed acknowledge response within a specified amount of time after submitting a request can be proven. By extending statecharts and TCTL with probabilities, [Yam96] allows even computing probabilities of timing properties by means of probabilistic real-time model checking.

- Furthermore, timing constraints on input signals may be inferred or delay characteristics required in order to meet some given behavioral specifications may be calculated. For instance, [CCM96] presents methods to compute quantitative timing information about a system, such as the minimum and maximum delay between specified events. In [ACH97], accumulated satisfaction times of state predicates are computed. [TKY+98] presents an approach based on timed automata to accurately ascertain delays of combinational circuits even in the presence of cross-talk.

The most severe restrictions on formal verification of timed automata result from resource limitations, i. e., computing power and memory. To overcome these restrictions, techniques such as compositional methods and partial order reduction have been applied successfully. Further approaches employ efficient data structures and algorithms to represent and manipulate state sets and timing constraints on clock values.

There exist many formal verification tools dedicated especially to real-time systems, e. g., KRONOS [DOTY96], UPPAAL [BLL+95], and HYTECH [HHW95]. Reachability analysis of timed automata often is performed using *difference bounds matrices* (DBMs) [Dil89] for representing *clock regions* during computation, as explained later on. For instance, the original verification engines of both KRONOS and UPPAAL are based on DBMs. Since DBM methods often fail for large models, other approaches have been proposed using different kinds of region representations.

Besides methods extending and improving DBM approaches by using, e. g., more compact data structures or on-the-fly reduction [LLPY97], DBM independent methods such as those based on binary decision diagrams [FGK96], *numerical decision diagrams* (NDDs) [ABK+97, BMPY97], and multi-terminal BDDs [RK97a, RK97b] have been employed successfully.

In this section, interval diagram techniques are applied to formal verification of timed automata in a multi-clock setting over discrete time. The employed verification techniques using interval diagrams are presented and their run-time behavior is compared with that of the NDD approach.

4.3.2 Reachability Analysis of Timed Automata

With regard to timed automata, the image computation in conventional symbolic formal verification has its counterpart in time forward projection. In the scope of this section, mainly reachability analysis of timed automata in a multi-clock setting is considered. It is performed by iterated application of the transition relation δ as described in Section 2.3.3

until reaching a fixpoint. Only discrete time represented by integer clock values is considered here. [ABK+97] introduces a discretization scheme transforming dense-time models into discrete-time models and thus allowing analysis using NDDs as well as interval diagram techniques.

4.3.2.1 Using Difference Bounds Matrices

Difference bounds matrices as introduced in [Dil89] may be used for formal analysis of timed automata. DBMs are square matrices of bounds representing convex polyhedra canonically. Unfortunately, non-convex polyhedra, especially unions of convex polyhedra as arbitrary clock regions used in formal verification, have no canonical representations using DBMs but have to be represented, e. g., by lists of matrices instead. Thus, equality testing during fixpoint computation becomes more and more difficult and expensive as the system model grows. Moreover, DBMs may not easily be combined with symbolic representations of discrete system states.

4.3.2.2 Using Numerical Decision Diagrams

Essentially, numerical decision diagrams [ABK+97] are BDDs [Bry86] representing sets of integer vectors. The integer elements are binary coded using "standard positional encoding". The sets being represented may be described using conjunctions and disjunctions of inequalities on integer variables, similarly to IDDs. Since the binary encoding requires an upper variable value bound, only finite sets may be described in contrast to IDDs.

In contrast to DBMs, NDDs may be used as canonical representations of arbitrary clock regions. [ABK+97] provides a method for discrete-time formal verification of timed automata using NDDs. While this approach is based in the main on conventional BDD techniques such as Boolean operations, time projection requires a new algorithm using binary modulo subtraction. It is briefly described as a recursive procedure for forward time projection of only one clock, but obviously may be expanded for more than one.

4.3.2.3 Using Interval Diagram Techniques

Analogously to the models of computation treated in Section 4.2, interval diagram techniques are suitable for formal verification of timed automata due to similar reasons.

- Again, the structure of the transition relation is very regular and efficiently representable using IMDs—by means of the equivalent of state distances.
- Discrete-valued clocks may be regarded as particular integer state variables of which the values increase simultaneously when time progresses.
- Integer time forward projection may be performed by repeated and simultaneous incrementation of all clock values about a *time distance* of 1, depending on the actual system state and thus similarly to state distances.

Similarly to NDDs, IDDs allow canonical representations of arbitrary clock regions which is important with regard to fixpoint computations as mentioned above. Moreover, they provide a suitable combination with symbolic representations of the discrete part of the model.

Unlike other approaches, the one described here does not distinguish between time projection and discrete state transitions. Conventionally, both computation stages are performed *alternately*. First, starting with an initial configuration, time is projected to determine all configurations reachable from the inital one by only progressing time. Thereafter, all possible state transitions are performed concurrently, etc. In contrast to this, using interval mapping diagrams allows for a *conjoint* transition behavior consisting of partial time projection—increasing time by one time unit—and discrete state transitions at the same time. This is performed using image computation as for conventional reachability analysis. As previous investigations have shown, this seems to be significantly superior to the alternate way with respect to interval diagram techniques. Although more fixpoint iterations are necessary, each one is essentially cheaper than otherwise.

To achieve this, a modified transition relation $\tilde{\delta} : \mathbf{Q} \times \mathbf{Z} \to 2^{\mathbf{Q} \times \mathbf{Z}}$ with

$$\tilde{\delta}(q, \mathbf{z}) = \Big\{ (q', \mathbf{z}') \;\Big|\; (\mathbf{z} \in \mathbf{G}_{qq} \cap \mathbf{G}_{qq'}) \wedge \big(\mathbf{z}' = R_{qq'}(\mathbf{z})\big) \Big\} \cup \Big\{ (q, \mathbf{z} + \mathbf{1}) \;\Big|\; \mathbf{z} + \mathbf{1} \in \mathbf{G}_{qq} \Big\} \tag{4.1}$$

is used instead of equation (2.1) on page 38. This transition relation effectively performs either at most one discrete state transition with respect to the argument configuration or time projection of exactly one time unit.

Equation (2.2) on page 38 is replaced by using a bounded time forward projection $\widetilde{\mathbf{\Phi}} : 2^{\mathbf{Z}} \to 2^{\mathbf{Z}}$ defined as

$$\widetilde{\mathbf{\Phi}}(\mathbf{P}) = \{\, \mathbf{z} \in \mathbf{Z} \mid \mathbf{z} \in \mathbf{P} \vee \mathbf{z} - \mathbf{1} \in \mathbf{P} \,\}. \tag{4.2}$$

After redefining $\widetilde{\mathbf{P}}^{\Phi} = \widetilde{\mathbf{\Phi}}(\mathbf{P}) \cap \mathbf{G}_{qq}$ and $\widetilde{\mathbf{P}}_{q'} = R_{qq'}(\mathbf{P} \cap \mathbf{G}_{qq} \cap \mathbf{G}_{qq'})$ (note the difference to the previous definition here), the immediate successors of a set of configurations $(q, \mathbf{P})$ are denoted—analogously to equation (2.3) on page 39—as

$$\tilde{\delta}\big((q, \mathbf{P})\big) = (q, \widetilde{\mathbf{P}}^{\Phi}) \cup \bigcup_{q' \neq q} (q', \widetilde{\mathbf{P}}_{q'}). \tag{4.3}$$

Figure 48 on the next page shows the transition relation IMD T of the example timed automaton in Figure 22 on page 37. The two left-most paths of the IMD T describe the transition guards, state changes, and reset functions resulting from both transitions, respectively, of the upper partial automaton. For instance, the top-most transition is enabled if $(v_1 = 0) \wedge (C_1 \geq 1)$ is satisfied, i. e., $v_1 \in [0, 0] \wedge C_1 \in [1, \infty)$. The consequence of this transition is that state variable v_1 is set to 1, and clock C_1 is reset to 0. Similarly, the two paths in the middle of T represent the transitions of the lower partial automaton. Altogether, the paths of both automata describe the right-hand argument of the union operator in equation (4.3) on the preceding page.

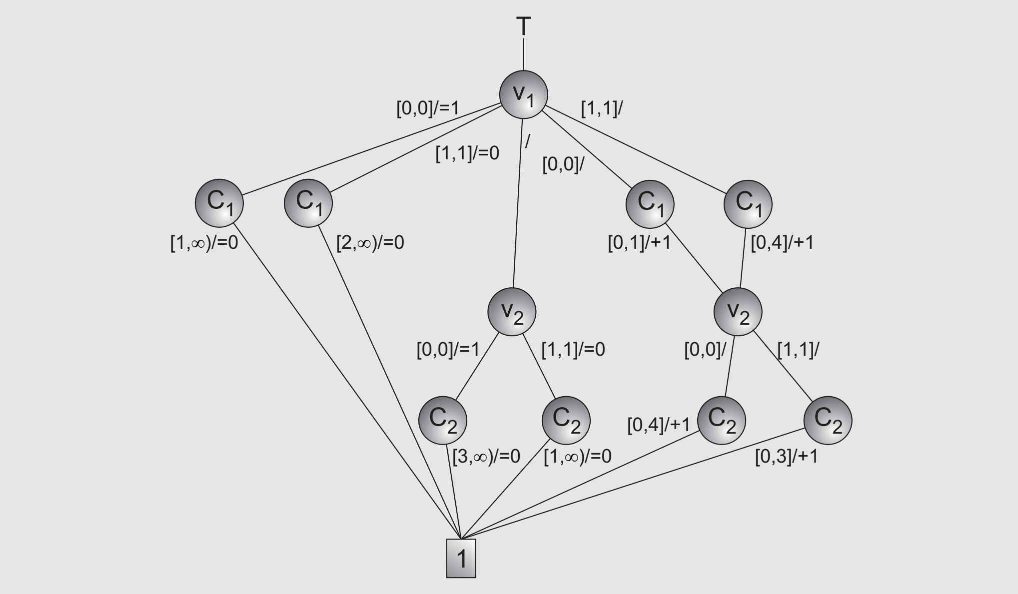

Figure 48. Transition relation IMD of timed automaton.

The right-most paths—four altogether—are required to model time progress depending on the actual state. Time can only progress if all clock values are increased simultaneously while not violating any of the staying conditions. The clocks increase about one time unit per step, but only if the respective conditions depending on the system state are satisfied. Thus, these paths describe the left-hand argument of the union operator in equation (4.3) on the facing page except for $(q, \mathbf{P})$ which is added algorithmically to the final result for $\tilde{\delta}\big((q, \mathbf{P})\big)$ later on.

In Section 3.2.5, an efficient algorithm is described for performing image computation using an IDD S for the state set and an IMD T for the transition relation, resulting in an IDD S' representing the image state set. This algorithm can be used for performing reachability analysis or real-time symbolic model checking by fixpoint computation.

Without further modifications, interval diagram techniques can be applied to the analysis of discrete hybrid automata (used in the fields of automation and control theory, for instance) on integer variables—analogously to timed automata—by replacing the common clock addition of $+[1, 1]$ by non-uniform integer rates.

4.3.3 Real-Time Symbolic Model Checking

Besides reachability analysis, the techniques described above are directly applicable to real-time symbolic model checking. There, not only the reachability of given states is proven or disproven, but also complex timing properties such as the satisfaction of timing constraints are investigated. In order to express such properties, real-time temporal logics formulae are used. For instance, [HNSY94] considers model checking of timed automata using the real-time temporal logic TCTL over dense time, [ACD93] is akin with regard to timed graphs. [CC94] describes symbolic model checking of timed transition graphs over discrete time using the logic CTL augmented by a bounded until operator. In [FGK96], QCTL symbolic model checking of quantitative temporal structures over discrete time is introduced, which has been refined to CCTL model checking of interval structures [RK97a] later on.

An interesting approach to real-time symbolic model checking of timed automata via reachability analysis is described in [ABL98]. The logical formula being checked is transformed into a *testing automaton*, which then is analyzed for reachability together with the original timed automaton being checked. Using this approach, real-time symbolic model checking based on interval diagram techniques can be performed directly as described in Section 4.3.2.3.

However, the straightforward extension of the reachability analysis techniques for real-time symbolic model checking is easily possible and obviously more feasible. Especially the approach of Ruf and Kropf [RK97a, RK98] to real-time symbolic model checking of interval structures using CCTL over discrete time is well suited to apply interval diagram techniques. In the following, a sketch of how to combine both approaches is given.

Interval structures are state transition systems of which the transitions are annotated with intervals representing the transition durations. Each interval structure has one clock for time measuring, but by composing several interval structures, a multi-clock setting emerges. The clock is reset to zero if a state of the interval structure is entered. A state may be left if the actual clock value lies within the interval of a leaving transition. The state has to be left at the latest when the maximal upper bound of all corresponding intervals is reached.

The CCTL logic used is an extension of CTL with time scopes and intervals. It contains the propositional operators $\neg$ and $\wedge$ and the temporal operator combinations $\mathsf{EX}_{[n]}$, $\mathsf{EG}_{[n]}$, and $\mathsf{EU}_{[m,n]}$. For instance, the timed next operator $\mathsf{X}_{[n]}$ indicates a condition that holds after n time units, while the timed globally $\mathsf{G}_{[n]}$ denotes a property that holds in all states during the first n time steps of a computation path. The timed until $\phi\ \mathsf{U}_{[m,n]}\ \psi$ holds of a computation path if this reaches after m to n time steps a state where ψ holds and if ϕ holds until then. Additional operators such as $\mathsf{EF}_{[n]}$ or $\mathsf{AG}_{[m,n]}$ are derived using the basic operations. Example specifications expressed using CCTL are:

- $\mathsf{AG}\ [req \implies \mathsf{AF}_{[80,100]}\ ack]$ means that a request *req* is followed by an acknowledge *ack* not before 80 time units but at the latest after 100 time units, while

- $\mathsf{AG}\ \big[req \implies \mathsf{A}\ [\neg ack\ \mathsf{U}_{[80,100]}\ ack]\big]$ specifies in addition to the above that there is no acknowledgement before 80 time steps.

Using the symbolic model checking approach of [RK97a, RK98], given interval structures first have to be composed by mapping them (using a complex and expensive reduction technique) onto one product interval structure—and thus one clock. Then this representation is used for performing efficient CCTL symbolic model checking by applying a dedicated technique for computing the formula $\mathsf{EX}_{[1]}\ \phi$ and by composing other temporal operators by means of the $\mathsf{EX}_{[1]}$ operator—for instance, using equivalences such as $\mathsf{EF}_{[n]}\ \phi = \phi \vee \mathsf{EX}_{[1]}\ \mathsf{EF}_{[n-1]}\ \phi$ for recursion.

Interval diagram techniques are applied to CCTL symbolic model checking analogously. The $\mathsf{EX}_{[1]}$ operator is computed using an IMD similar to that in Figure 48 on page 97 representing the timed transition relation. The necessary pre-image operation then is performed using the techniques for backward image computation as described in Section 3.2.5, replacing the above-mentioned forward image computation. Even a technique similar to the time prediction described in [RK98] is possible. Instead of computing temporal operations using recursion based on the $\mathsf{EX}_{[1]}$ operator, dedicated operations as described in [RK97b] are also possible for the sake of performance. In contrast to [RK98], the complex mapping of several interval structures onto one is not necessary using interval diagram techniques because many local clocks can be represented explicitly instead of one global clock.

4.3.4 The Explicitly Timed FunState Model

As mentioned in Section 4.2.5.3, for formal verification of the timed *FunState* model the values of function latencies can be considered explicitly instead of abstracting them. This is achieved by combining *FunState* and timed automata.

For this, the timed behavior of each non-zero latency function is modeled by a timed automaton as shown in Figure 49. The upper transition is invoked when function f_i is activated, resetting the corresponding clock C_i. As long as the timed automaton is in the state annotated with $p_i = 1$, function f_i is processing. When this state is left after the latency τ_i has passed, the finalization of f_i is performed. Latency uncertainties resulting in execution time interval ranges instead of determinate latencies can be expressed easily.

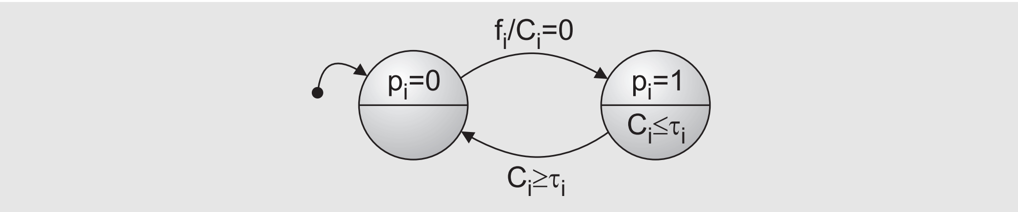

Figure 49. Timed automaton modeling timed behavior of *FunState* function.

4.3.5 Empirical Results

In [ABK+97], two parameterized models are used for comparing the NDD and the DBM approach. In addition to DBM methods, the NDD techniques have been implemented in the real-time formal verification tool KRONOS. Since NDDs seem to be greatly superior to DBMs—which on the other hand are suitable for directly handling dense time—with regard to computation time and memory resources, only NDDs are considered here. Their run-time behavior is compared with that of the interval diagram techniques approach [Str99].

The examples used are a timed automaton $\mathcal{A}$ with one discrete state and an automaton $\mathcal{B}$ with many states—shown in Figure 50 a) and Figure 50 b), respectively. A configuration parameter n indicates the number of self-loop transitions from and to $\mathcal{A}$'s only location or the number of concurrent partial automata—each one consisting of two locations and two transitions—of $\mathcal{B}$, respectively. For both $\mathcal{A}$ and $\mathcal{B}$, n denotes the number of clocks as well. The total number of states of $\mathcal{B}$ is 2^n.

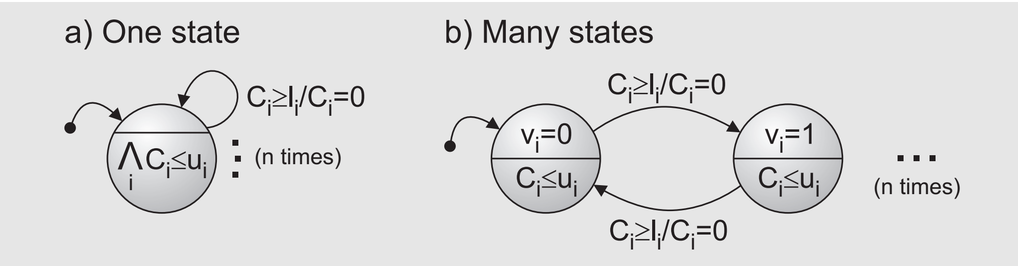

Figure 50. Timed automata $\mathcal{A}$ with one state and $\mathcal{B}$ with many states.

The timed automaton $\mathcal{A}$ may be used for modeling a system generating n events $\tau_1, \dots, \tau_n$ such that each occurrence of τ_i must be followed by another one within u_i time units, while every two occurrences of τ_i must be separated by at least l_i units. Similarly to the example automaton in Figure 22 on page 37, $\mathcal{B}$ represents n Boolean input signals of which the only constraints are that every two changes in their values are separated by some time between l_i and u_i.

Both kinds of automata are indispensable to analyzing system behaviors under all possible inputs. Due to the large degree of independence of the respective clocks, the reachable state spaces are extremely large, requiring immense computing power.

Since no NDD implementation was available and [ABK+97] and [BMPY97] only sketch the algorithms used, the comparison had to be performed based on the results reported in [ABK+97], obtained on a 170 MHz Ultra™ 1 computer by Sun Microsystems, Inc. The computation time results have been downscaled due to different computing powers—for the experiments, a Sun Ultra™ 60 with 360 MHz has been used—using a factor of 1.95, determined by extensive comparative experiments on both machines.

In Table 6, the computation times T to ascertain the set of reachable configurations of the "one state" automaton $\mathcal{A}$ and the "many states" automaton $\mathcal{B}$ are compared for the NDD and the IDD/IMD approach, depending on the configuration parameter n.

Table 6. Computation times T in seconds.

One state automaton $\mathcal{A}$

n	T IDD/IMD	T NDD
10	36.4	24.9
11	56.4	47.2
12	88.2	91.7
13	121	184
14	187	421
15	369	975
16	767	2574
17	1725	7387
18	3711	22140
19	8011	-

Many states automaton $\mathcal{B}$

n	T IDD/IMD	T NDD
5	29.7	18.0
6	83.9	69.3
7	218	387
8	506	1666
9	1303	9884
10	3049	-
11	7120	-
12	24412	-

In Figure 51, the computation time T of the reachability analysis of automaton $\mathcal{A}$ is depicted in logarithmic scale, depending on n.

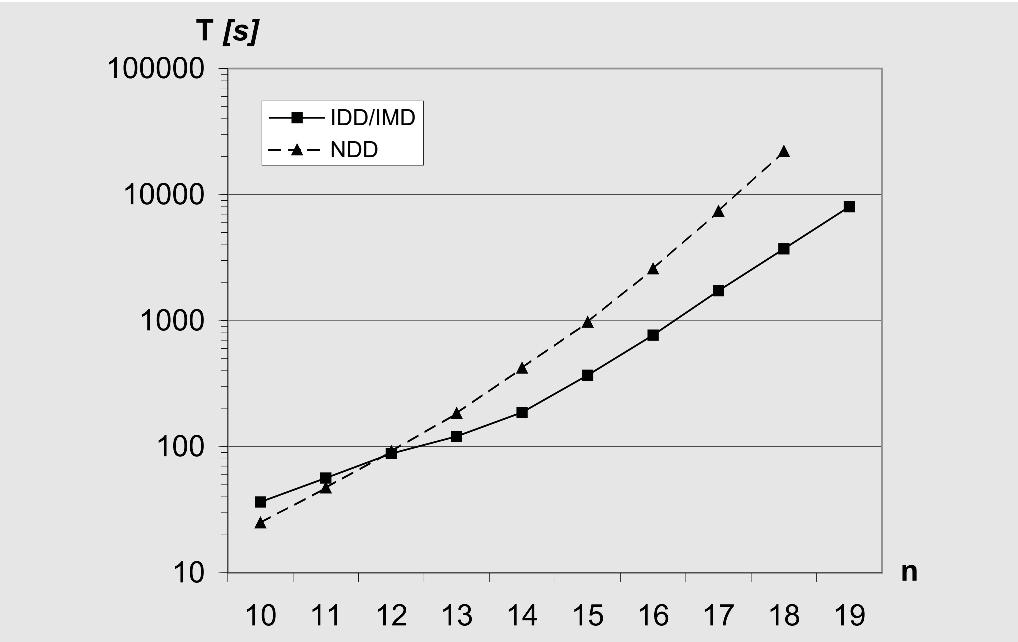

Figure 51. Computation time T for "one state" timed automaton $\mathcal{A}$.

Automaton $\mathcal{B}$ behaves very similarly. Figure 52 on the following page shows the computation time of its reachability analysis. Note that again a logarithmic scale is chosen for the sake of comparability, meaning that even small differences in the plot may correspond to large differences in the computational speed.

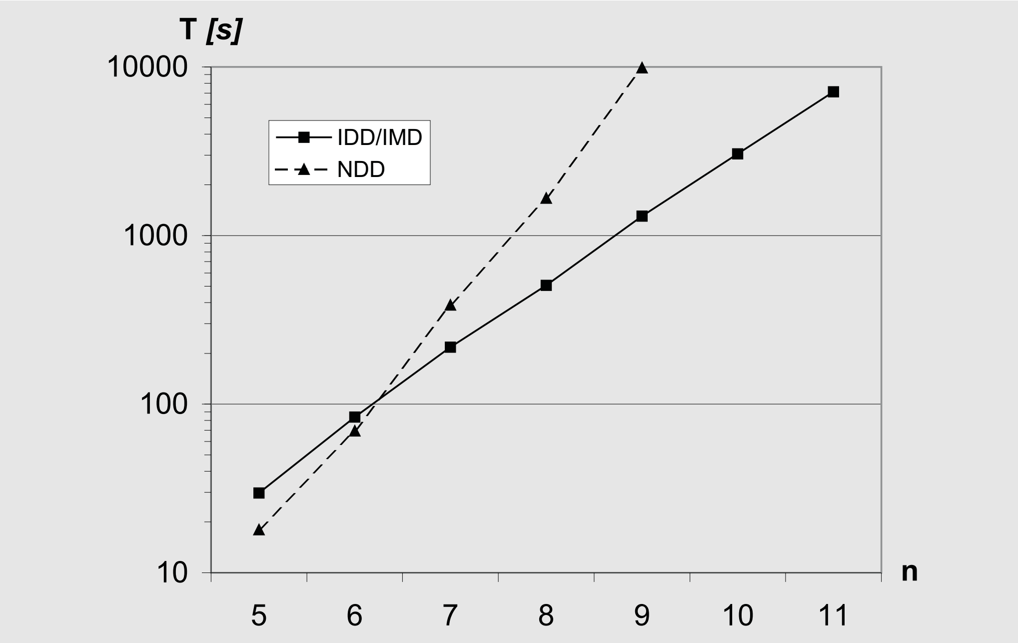

Figure 52. Computation time T for "many states" timed automaton $\mathcal{B}$.

For medium and large models, the IDD/IMD approach significantly outperforms the NDD approach. The break-even occurs in the region of one or several minutes of computation time which is of importance especially with regard to practical application. Most noteworthy is that the lesser slope of the IDD/IMD computation time seems to be an indication that the algorithmic complexity of this kind of application is better. The highest speedup achieved is more than 7. It is expected to be even greater for larger models—of which no NDD computation times have been available for comparison.

For experimental purposes, the Java programming language has been used in the interval diagrams implementation, while the NDD approach was implemented in C in the state-of-the-art tool KRONOS. Thus, equivalent implementations should reduce the IDD/IMD computation times about up to one decade such that this approach would outperform the NDD approach even for small parameter values of n.

4.3.6 Further Related Work

Besides the NDD approach described in Section 4.3.2.2, especially two other related approaches are relevant. In [FGK96], vectors of BDDs are used as a representation of the transition relation. This representation has been replaced in [RK97a, RK97b] for efficiency reasons by using MTBDDs [CFZ96] as state set representation. Time projection is performed by local manipulations in the MTBDD leaves. [RK98] presents significant gains in performance in comparison with the BDD approach. Further, the

MTBDD approach is compared with the KRONOS tool yielding a speedup of up to 3.5. Nonetheless, this may not be compared directly with the speedup of 7.5 achieved here because the models used for comparison are different.

An approach expressly inspired by interval diagram techniques (as stated by the authors) is introduced in [BLP$^+$99]. It makes use of *clock difference diagrams* (CDDs) which are based on a modification of IDDs. In addition to clock values, CDDs store differences between clock values, but unfortunately by abandoning a directly canonical representation and hence its advantages with regard to fixpoint comparisons. Methods for CDDs such as union, intersection, and complement are straightforwardly adopted from IDDs. For equality tests during fixpoint computations, CDDs first have to be transformed into a quasi-canonical form. This seems to limit run-time reductions significantly which is confirmed by the experimental results presented. Compared with the original UPPAAL tool based solely on DBMs, considerable space savings using a hybrid CDD/DBM approach are accompanied even by slight run-time increases. But in contrast to the interval diagram techniques as described in this monograph, CDDs can be used directly in a dense time setting.

5

Scheduling

BESIDES the applications to scheduling analysis described in Section 4.2.3.1, symbolic model checking principles have been applied to the synthesis area of symbolic scheduling, where correct schedules are generated automatically. In this chapter, a scheduling method for heterogeneous embedded systems is presented [STZ+99]. It is based on the *FunState* internal representation model which enables the explicit representation of non-determinism and scheduling.

The scheduling method is able to deal with mixed data/control flow specifications and takes into account different mechanisms of non-determinism as occurring in the design of embedded systems. Constraints imposed by other already implemented components are respected. The scheduling approach avoids the explicit enumeration of execution paths by using symbolic techniques and guarantees to find a deadlock-free and bounded schedule if one exists. A such schedule is called *feasible*. The result is a scheduling policy which may be implemented, e. g., as a software controller on a uni-processor. The generated schedule consists of statically scheduled basic blocks which are dynamically called at run time.

5.1 FunState and Scheduling

This chapter is concerned with a scheduling procedure adapted to the *FunState* internal representation. Problems which are typical for the design of complex embedded systems are, for instance,

- different kinds of non-determinism such as partially unknown specification (to be resolved at design time), data dependent control flow (to be resolved at run time), and unknown scheduling policy (to be resolved at compile time),

- dependencies between design decisions for different system components,
- correlations between processing modes in different components.

These properties necessitate particular scheduling approaches as the number of execution paths to be considered grows exponentially with increasing degrees of non-determinism. Moreover, the complexity of the models of computation and communication greatly increases the danger of system deadlocks or queue overflows, see, e. g., [LP95].

To overcome drawbacks of either purely static or dynamic scheduling approaches and to combine their advantages, Lee proposed a technique called *quasi-static* scheduling [Lee88]. Similarly to static scheduling, most of the scheduling decisions are made during the design process, providing only few run-time overhead and partial predictability. Only data dependent choices—depending on the value of the data or resulting from a reactive, control oriented behavior—have to be postponed until run time. Techniques related to quasi-static scheduling have been developed using, e. g., *constraint graphs* [KD92, CTGC95], *dynamic dataflow graphs* [Buc93], actors with data dependent execution times [HL97], *free-choice* Petri nets [SLWS98], and *equal conflict nets* [SLWS99].

The latter two approaches determine schedule sequences for certain fairly restricted subclasses of Petri nets. Free-choice Petri nets are Petri nets of which every arc is a unique outgoing arc from a place or a unique incoming arc to a transition (or both). In addition, all outgoing arcs of a place must have the same arc weight. Equal conflict nets[1] are little less restrictive in that transitions sharing some predecessor place must have exactly the same predecessor places—hence not only one—and the same corresponding arc weights. Both classes exhibit the property that whenever an output transition of a place is enabled, all the output transitions of that place are enabled. Therefore, the choice is local and said to be *free* since it does not depend on the rest of the system.

The scheduling approach presented in the following is more general since none of these properties is assumed. Both free-choice and equal conflict net approaches solve occuring problems with regard to data dependencies by avoiding them and seriously restrict the degrees of freedom in modeling. Further, not only Petri nets are treated in this chapter but the more complex *FunState* model of computation. Nonetheless, the above Petri net subclasses can easily be modeled and then scheduled using *FunState*. An approach related to the methods described in the following has been introduced in [TTS00]. There, a plain RSM-based scheduling approach is presented and applied to Petri nets.

[1] Equal conflict nets [SLWS99] previously had been named *extended* free-choice Petri nets [SLWS98] by the same authors, Sgroi *et al.*

The approach taken in this chapter is based on symbolic techniques which use a combination of efficient representations of state spaces and transition models as well as symbolic model checking principles in order to avoid the explicit enumeration of execution paths. Symbolic approaches often turned out to provide advantages regarding computation time and memory resources, compared with conventional approaches.

There exist some approaches to applying symbolic methods to control dependent scheduling for high-level synthesis. These exact and efficient symbolic scheduling techniques based on BDDs are used for performing control/data path scheduling by combining the advantages of both heuristic methods and techniques based on *integer linear programming* (ILP). BDDs are used for describing scheduling constraints and solution sets either directly [RB94] or encapsulated in finite state machine descriptions [CD94, HB98, HB99].

In particular, the following results with regard to scheduling of *FunState* models are described in this chapter.

- Different mechanisms of non-determinism as occurring in the design of embedded systems are classified.
- A scheduling method for heterogeneous embedded systems is developed which takes into account these different kinds of non-determinism and constraints imposed by other already implemented components and which deals with mixed data/control flow specifications.
- The resulting scheduling automaton is optimized with respect to the length of static blocks and the number of states.
- The approach is illustrated using a hardware/software implementation of a fast molecular dynamics simulation engine. Moreover, the feasibility of the approach is shown with an ATM switch model.

Applying a scheduling method to a *FunState* model may be regarded as an example of a refinement step using *FunState* as an internal design representation. The specification as well as the result of the scheduling procedure are represented as *FunState* models. The scheduling method presented here proceeds as follows.

1. The basis is a *FunState* model which specifies all admissible schedules by means of non-determinate transition behavior—representing all design alternatives.
2. By symbolic exploration of the resulting regular state machine, the state space is traversed to search for cycles representing valid schedules. This is motivated by the fact that after having traversed a cycle in the dynamic state transition diagram, an already visited state is reached for which the scheduling behavior is known. Thus, by finding all necessary cycles, non-determinism is reduced as far as possible. Hence, design alternatives are removed by taking decisions.

3. The extracted schedule consisting of paths in the dynamic state transition diagram is transformed into a finite state machine which then is compacted using state minimization techniques.

4. Finally, the result is embedded in the original *FunState* model by replacing the schedule specification part. Furthermore, it may be transformed into program code.

The remainder of this chapter is organized as follows. Section 5.2 explains the basic scheduling approach in detail, while Section 5.3 focuses on its symbolic form. Section 5.4 then sketches some extensions of the basic scheduling approach. Finally, Section 5.5 treats the two above-mentioned example models.

5.2 Conflict-Dependent Scheduling

As mentioned above, quasi-static and related scheduling approaches, e. g., [Lee88, CTGC95], try to combine the advantages of static and dynamic scheduling methods. To achieve this, the resolution of data or environment dependent control is done at run time whereas the tasks that need to be executed as a consequence of a run-time decision are scheduled statically. The aim is to make most of the scheduling decisions at compile time, leaving at run time only choices that, e. g., depend on the value of data.

5.2.1 A Scheduling Example

To exemplify this, consider the simple *FunState* model shown in Figure 53 on the next page. Assume that component B represents a processor transforming data streams between the components A and C. Let A and C be components mapped onto hardware such as an input or output device, respectively, or an interface to a sensor, an actor, or another processor.

Let the behaviors of A and C be specified by the respective state machine. Not taking into account these additional constraints may lead to less efficient or even incorrect schedules. The state machine of A describes that its functions always are executed in the order $f_1 f_2 \ f_1 f_2 \ldots$ Hence, it is guaranteed that after each firing of f_1, function f_2 is executed and vice versa.

The state machine of B shown in Figure 53 on the facing page describes a specification of possible schedules for B. This specification should be used for finding a feasible schedule which respects the additional information concerning other components. All transitions starting in a dark-shaded state represent design *alternatives* of which one may be chosen during schedule design. In contrast to this, a light-shaded state

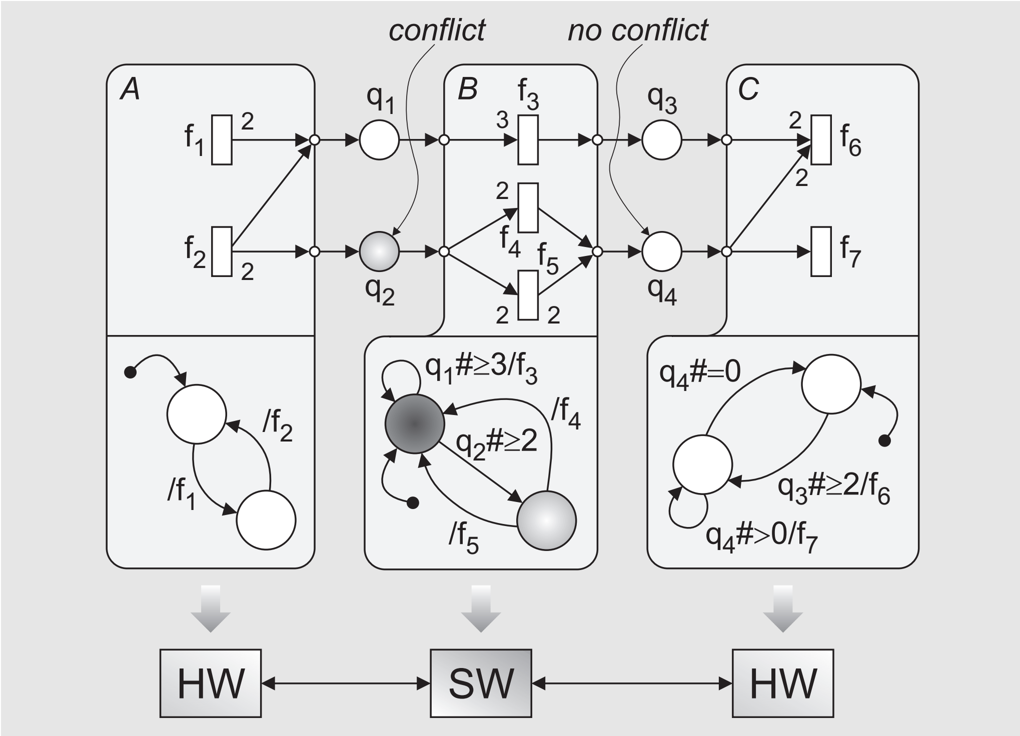

Figure 53. Example *FunState* model.

contains a *conflict* concerning its outgoing transitions. The conflict can be resolved only at run time. Hence, no design decision is possible. Conflicts occur, for instance, when decisions depending on the value of data or environmental circumstances have to be taken. Assume that tokens in queue q_2 elicit such dependencies with regard to the preceding functions while tokens in q_4 do not. White states in the *FunState* model are states which either have only one outgoing transition or of which all transitions have disjoint predicates. Thus, the transition behavior of these states is *determinate*. Note that in component C the state with two outgoing transitions is determinate for this reason.

The execution behavior of B depends on two different decisions:

- a decision at compile time—whether to fire either function f_3 or one of f_4 and f_5—, and
- a decision at run time—whether to fire either f_4 or f_5.

The scheduling issue now consists of balancing the executions of f_3 on the one hand and of f_4 or f_5 on the other hand, such that no deadlock or buffer overflow occurs. There exists no knowledge advantage can be taken of about the decision whether to fire f_4 or f_5.

Suppose that B and C execute sufficiently often (they are "faster" than the preceding component) such that there are no unbounded numbers of tokens simultaneously in q_1 and q_2 or in q_3 and q_4, respectively. An important issue of schedule development are feasibility and correctness of the resulting schedule. A possible schedule of B described by the specification is $(f_4|f_5)\ (f_4|f_5)\ldots$, where f_4 and f_5 are executed alternatively (indicated by "|") and iteratively—thus ignoring f_3. But this schedule is not feasible as the queue contents of q_1 and q_4 are not bounded. If $f_3\,(f_4|f_5)\ f_3\,(f_4|f_5)\ldots$ had been chosen—f_3 is executed, then either is f_4 or f_5, etc.—, this would result in an incorrect behavior of C as f_6 could attempt to read too much tokens from q_4 after some time.

In contrast to this, the schedule $(f_4|f_5)\,f_3\ (f_4|f_5)\,f_3\ldots$ is valid with respect to specification and component C, and it is feasible. An implementation of this schedule can profit from the fact that f_3 may be executed only if f_4 has been executed immediately before. From the behavior of A follows that for the execution of f_3 no condition is necessary because q_1 always contains enough tokens. Thus, the resulting schedule may be implemented more efficiently by considering only necessary conditions as less queue contents have to be determined. Data independent operations are scheduled statically in clusters starting with one data dependent operation. Only for the beginnings of these clusters dynamic scheduling is a necessity.

Using the *conflict-dependent scheduling* techniques proposed in this chapter, the above issues are taken into account. Intuitively, the scheduling is performed by replacing dark-shaded states by white states—taking decisions and thus removing design alternatives. Since explicit timing properties (execution times, timing constraints, etc.) are ignored in the scope of this section, the resulting schedule—similarly to the scheduling of SDF graphs, e. g., [TZB98]—consists of sequences of function executions.

In this section, only software scheduling in a uni-processor setting is considered. Extensions for hardware scheduling under resource constraints, the consideration of the timing behavior of the system—to allow for the specification of timing constraints—, or scheduling for several processors are easily possible and partly presented in Section 5.4.

Initially, the given *FunState* model contains a *schedule specification automaton* (B's state machine in Figure 53 on the page before) which extends the FSM part such that all possible schedule behaviors are modeled. This *FunState* model represents a totally dynamic scheduling behavior and is used for performing the symbolic scheduling procedure as described below. The result of this procedure is the *schedule controller automaton* which restricts the scheduling behavior to being only conflict-dependent. This automaton may replace the specification automaton of the original *FunState* model, e. g., for analysis purposes such as verification. Finally, the controller automaton may be transformed into program code to implement the controller. These stages are described in the following.

5.2.2 Conflicts and Alternatives

A conflict in the context of this monograph is a non-determinism in the specification which may not be resolved as a design decision, but of which all possible execution traces have to be taken into account during the schedule. Thus, the multi-reader queue q_4 in Figure 53 on page 109 does not represent a conflict as both following functions may read all tokens of q_4 independent of their value or possible external circumstances.

In contrast to that, the queue q_1 in Figure 54 a) is a multi-reader queue that may contain tokens which only one of the queue's readers f_2 and f_3 consumes (depending, e. g., on the token data) but the other one does not. Besides such data dependent conflicts, conflicts depending on environmental circumstances may occur.

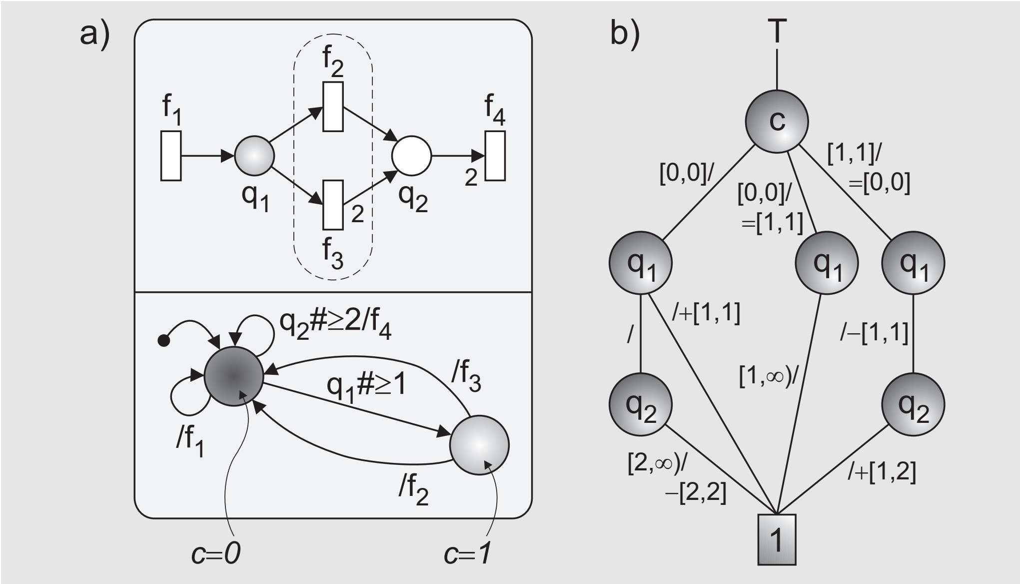

Figure 54. *FunState* model of conflict and transition relation IMD.

If the predicates of all transitions leaving a state are mutually disjoint, then the state is called *determinate*, otherwise *non-determinate*. In summary, two different kinds of non-determinism are distinguished which leads to a classification of non-determinate states as follows.

- **Conflict**
  Non-determinate states are involved in a conflict concerning its outgoing transitions if the non-determinism can be resolved only at run time. Hence, no design decision is possible. Conflicts occur, for instance, when decisions depending on the value of data are made or when environmental circumstances have to be taken into account. The transitions involved in a conflict are called *conflicting*.

- **Alternative**
  If among several transitions leaving a state any transition can be chosen, this fact represents an alternative. Like this, for instance, different scheduling policies—i. e., different orderings of actor executions—can be modeled. Such decisions do not directly depend on the value of data but describe design alternatives which may even be fixed at compile time.

Furthermore, different alternative algorithms can be modeled of which one or some can be selected during the design phase.

This way, different sources of non-determinism can be identified, and this information can be used for methods such as scheduling or formal verification.

The states of the FSM part of *FunState* models are divided into three types. According to Section 5.2.1, light-shaded states are called *conflict states*, dark-shaded states are *alternative states*, and determinate states are white. While the property of a state being determinate is derived directly from its transition predicates, the non-determinate states have to be divided explicitly into conflict states and alternative states as both are semantical properties.

All transitions leaving an alternative state represent design choices which may be made during schedule development. In contrast to that, all transitions leaving a conflict state represent decisions which may not be taken at compile time, but which keep their non-determinate character until run time. Hence, besides explicit conflicts incomplete constraint specifications resulting in a non-determinate behavior are modeled using conflict states. Otherwise the non-determinism would be treated as a design alternative and removed during the scheduling process.

Determinate states with only one outgoing transition are called *static* as there exists only one possibility of quitting them. Determinate states with more than one transition, alternative states, and conflict states are called *dynamic* because they represent a dynamic execution behavior with several traces depending, e. g., on queue contents or data.

5.2.3 Schedule Specification Automaton

In order to model the above-mentioned conflicts, a schedule specification automaton is built which represents all possible conflict behaviors and thus specifies all valid schedules. The lower part of Figure 54 a) on the page before shows the specification automaton used for describing the above-mentioned conflict behavior concerning f_2 and f_3 with regard to q_1. When one of the functions is enabled—q_1 contains at least one token—, the automaton can make a transition from the initial alternative state to the conflict state. Then, after executing either f_2 or f_3 it returns to the alternative state.

Besides the variables for the queue contents, a state variable c for the FSM states has been introduced. Figure 54 b) shows the interval mapping diagram representing the transition relation of the *FunState* model in Figure 54 a). This IMD is used for symbolic state space traversal as explained below.

Figure 55 on the facing page shows admissible schedules specified by the schedule specification automaton of the *FunState* model shown in Figure 53 on page 109. The reason for the infeasibility of the schedules S_1 and S_2 is given in Section 5.2.1.

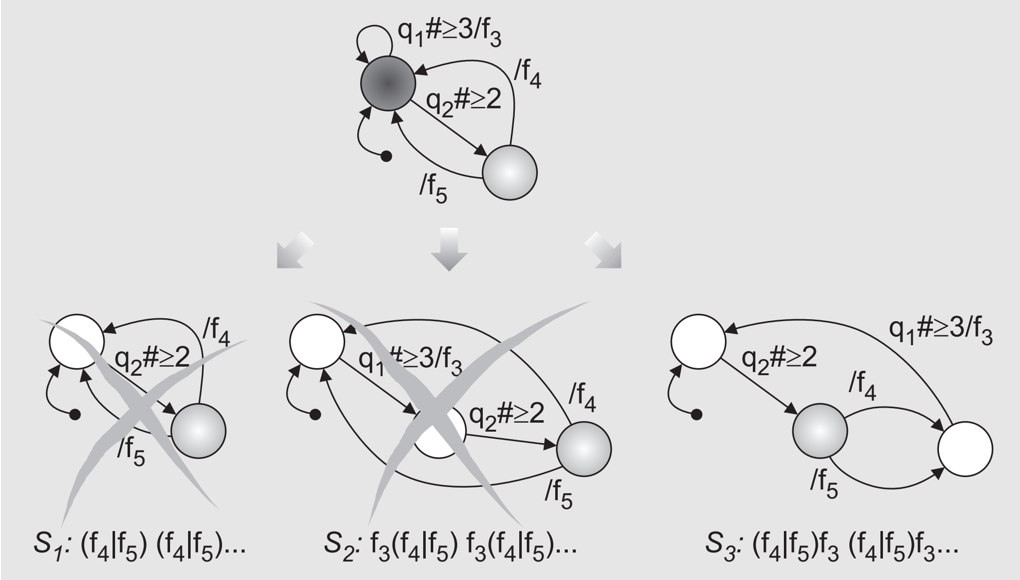

Figure 55. Several schedules specified by specification automaton in Figure 53.

5.2.4 Performing Conflict-Dependent Scheduling

The aim of the described scheduling process is to sequentialize functions specified as concurrent while preserving all given conflict alternatives. The resulting schedule has to be feasible, i. e., deadlock-free and bounded, as mentioned in Section 1.1.1.

Figure 56 shows the *regular state transition graph* of the *FunState* model in Figure 54 on page 111.

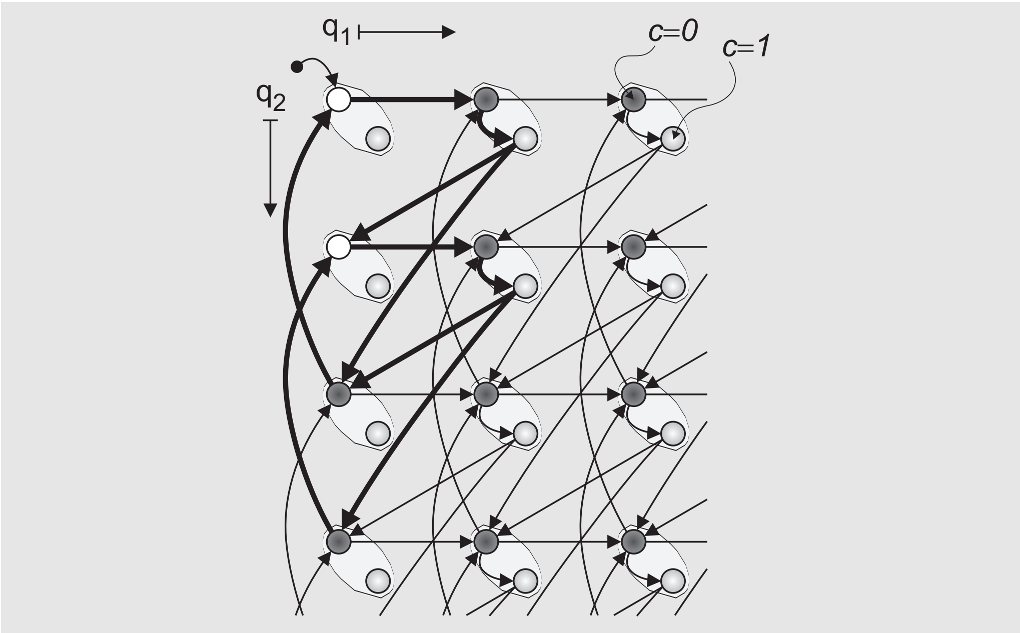

Figure 56. Regular state transition graph with schedule.

In the following, the scheduling procedure is explained based on Figure 57 on the following page. First, the dynamic state transition graph of the corresponding regular state machine is searched for the shortest paths from the initial state to itself or any state already visited during the search. One of these (possibly multiple) shortest paths—representing or at least containing a cycle—is selected as the basis of the following scheduling procedure—P_1 in this example.

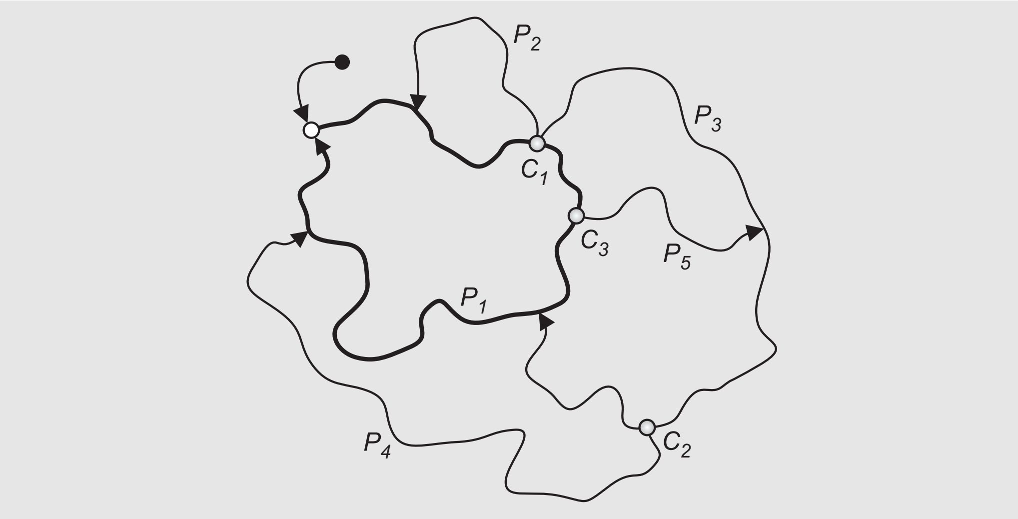

Figure 57. Paths in dynamic state transition graph describing schedule.

All conflict states of the selected path P_1 need further investigation as no conflict decision may be taken during schedule design. Hence, beginning with the successor states of the conflict state marked with C_1 again the dynamic state transition graph is searched until reaching any state visited already—resulting in the paths P_2 and P_3. Additional conflict states traversed during this search are also treated as described above. Thus, the conflict marked with C_2 results in the path P_4. Finally, C_3 causes P_5.

The schedule is complete when each successor state of each visited conflict state has been considered. Thus, it is guaranteed that any conflict alternative during run time may be treated by providing a determinate schedule until the next conflict to be resolved.

If no schedule has been found while traversing one of the conflict paths, another shortest path is selected to repeat the scheduling procedure. If checking all shortest paths does not result in finding a complete schedule, longer paths are selected. By introducing a bounding box on the state space, the search space may be restricted. Thus, the termination of the algorithm is guaranteed. Further, if a deadlock-free and bounded schedule exists, the above procedure will find it.

The length of the paths as the optimization objective is a heuristic criterion to minimize the number of actor executions and run-time decisions. This objective can be combined or replaced by criteria such as the least number of conflicts involved and their extent, or the shortest execution time along a path. In addition, the size of the bounding box on the state space is closely related to the amount of memory needed to store data of the scheduled *FunState* model. Therefore, restrictions on the run time as well as on the required memory can be included in the scheduling procedure.

5.2.4.1 The Algorithm

The scheduling procedure described above is performed using the function *scheduleGraph*$(a, \mathbf{B})$. An outline of its recursive algorithm is sketched in Algorithm 6 based on the regular state machine equivalent of the *FunState* model. The result is a directed graph representing the dynamic state transition graph of the schedule. The function arguments are a state a and a set of states $\mathbf{B}$ which may include a. The recursion is initiated by calling *scheduleGraph*$\big(x_0, \{ x_0 \}\big)$ where x_0 is the initial system state.

Alg. 6. Determine the state transition graph of the schedule

1 **funct** *scheduleGraph*$(a,$ **var** $\mathbf{B}) \equiv$
2 $\lceil x_0 \xrightarrow{t_0} x_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} x_n := \mathit{findPath}(a, \mathbf{B})$;
3 $\quad\mathbf{B} := \mathbf{B} \cup \{ x_0, x_1, \ldots, x_{n-1} \}$;
4 $\quad$initialize graph G with $x_0 \xrightarrow{t_0} \ldots \xrightarrow{t_{n-1}} x_n$;
5 $\quad$**for** $i := 0$ **to** $n-1$ **do**
6 $\qquad\mathbf{C} := \mathit{confl}(t_i) \setminus \{ t_i \}$;
7 $\qquad$**if** $\mathbf{C} \neq \emptyset$
8 $\qquad\quad$**then foreach** $t = (x, y) \in \mathbf{C}$ **do**
9 $\qquad\qquad\quad$expand G by $x \xrightarrow{t} y$;
10 $\qquad\qquad\quad$**if** $y \notin \mathbf{B}$
11 $\qquad\qquad\qquad$**then** expand G by *scheduleGraph*$(y, \mathbf{B})$ **fi od fi od**;
12 $\quad$**return** $G \rfloor$.

The function call is by reference with respect to $\mathbf{B}$. Hence, modifying the value of parameter $\mathbf{B}$ changes its value even in the calling instance of the function. The syntax $x_0 \xrightarrow{t_0} x_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} x_n$ represents a path from state x_0 to state x_n consisting of both the visited states x_i and the transitions t_i between them.

The directed graph G is an intermediate graph which finally represents the resulting state transition graph. Expanding G by another graph H at lines 9 and 11 means that by maintaining the graph structures the nodes and edges of H are added to G at those locations where the nodes of G and H match each other. Paths are treated as graph structures for simplicity in Algorithm 6 (lines 4 and 9).

A conflict is represented as a set of conflicting transitions of the dynamic state transition diagram $\mathbf{C} = \{ t_1, t_2, \ldots, t_n \} \subseteq \mathbf{T}$ where all $t_i \in \mathbf{C}$ must begin in the same conflict state. The function *confl*(t) returns the set of conflicting transitions $\mathbf{C}$ of the argument transition t with $t \in \mathit{confl}(t)$.

The function *findPath*$(a, \mathbf{B})$, where a is a state and $\mathbf{B}$ is a set of states which may include a, returns a path $a \xrightarrow{t_0} \ldots \xrightarrow{t_{n-1}} b$ from state a to any state $b \in \mathbf{B}$ with $n \geq 1$. This path is selected according to above-mentioned optimization objectives.

Unfortunately, the mentioned search for paths in the dynamic state transition graph performed by *findPath*$(a, \mathbf{B})$ is computationally very expensive since the number of paths to be considered potentially grows exponentially. Hence, for reasons of efficiency, symbolic methods have been applied to this graph traversal task as described in Section 5.3.

One necessary extension of *scheduleGraph*$(a, \mathbf{B})$ is not described in Algorithm 6 on the page before for the sake of clarity. Often no cycle exists which includes the initial state x_0. Thus, beginning with x_0, the dynamic state transition graph has to be traversed until finding a cycle which forms the basis for a valid schedule. As a result, the schedule obtained contains an initialization phase prior to becoming cyclic. To achieve this, the algorithm of *scheduleGraph*$(a, \mathbf{B})$ has to be modified such that the simple path search—trying to find a path from one given state to another given state—is replaced by searching a path from the given state to any state visited twice during this search.

It turned out that the following modification—only sketched here for simplicity—of Algorithm 6 on the preceding page is more feasible since it results in more compact schedules. In contrast to the given structure of *scheduleGraph*$(a, \mathbf{B})$ which recursively attempts to ascertain the conflicts resulting from a conflict as soon as this conflict is available, the modification proceeds as follows. First, the conflict states along a path are collected. Then paths resulting from these conflicts are determined, but conflicts occuring during this search are also collected instead of immediately investigated. Collected conflict states are investigated one after the other and hence in order of their appearance during the search.

The results of the modified algorithm seem to be better because first a basic framework of the schedule is determined which then is filled with details gradually. With regard to Figure 57 on page 114, the original algorithm of *scheduleGraph*$(a, \mathbf{B})$ determines paths and investigates conflicts in the order P_1, C_1, P_2, P_3, C_2, P_4, C_3, P_5. In contrast, the modified algorithm follows the order P_1, C_1, P_2, P_3, C_3, P_5, C_2, P_4.

5.2.5 Schedule Controller Generation

The resulting schedule consists of paths of the regular state transition graph as shown in Figure 56 on page 113. The corresponding subgraph in Figure 58 a) on the next page is the basis for the generation of the controller automaton. As a consequence of the scheduling process, all alternative states have been been replaced by determinate states—taking decisions and thus removing design alternatives. The predicate p identifies the run-time decision associated with the conflict node.

In order to reduce the implementation effort, this state transition graph may be simplified. Obviously, this process can be driven by many different objectives, for instance, minimizing the number of states in the schedule automaton or keeping sequences of static nodes.

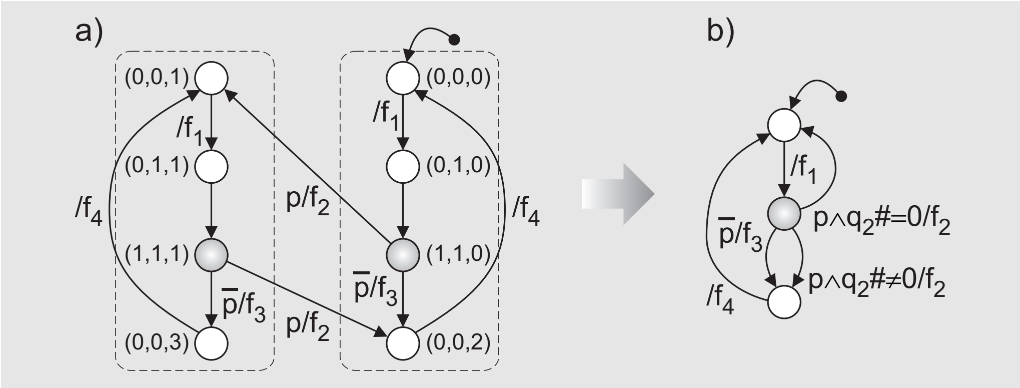

Figure 58. State transition graph of schedule for (c, q_1, q_2) and resulting controller automaton.

As an example, a procedure is described which minimizes the number of states under the condition that sequences of static nodes are not partitioned. This way, the number of dynamic decisions (at run time) is not increased in any execution trace. The optimization procedure is based on well-known state minimization methods and uses the following equivalence relation.

- Two static states are equivalent iff for any input they have identical outputs and the corresponding next states are equivalent.
- Two dynamic states are equivalent iff they are of the same type (conflict, alternative, or determinate) and they correspond to the same node in the non-scheduled state machine, i. e., they have the same state name but different queue contents associated.
- Static and dynamic states are not equivalent to each other.

This definition is used for performing the usual iterative partitioning, e. g., [DeM94], of the state set until only the equivalence classes are obtained as partitions. The initial partition consists of one block containing the static states and another block containing the dynamic states, hence satisfying a necessary condition of equivalence. These partition blocks then are iteratively refined by further splittings, according to the above equivalence rules. When the iteration converges, the corresponding blocks are equivalence classes. The ambiguity of the next states in the case of dynamic states finally is resolved by annotating outgoing edges with disjoint predicates.

Figure 58 b) shows the controller automaton as the result of this process. It may be transformed easily into program code as shown by Algorithm 7 on the following page as pseudo code. The predicate p checked at line 3 identifies the run-time decision associated with the conflict node.

Alg. 7. Controller program code

```
begin
  a: f1;
     if p
       then f2;
            if q2 = 0 then goto a fi
       else f3 fi;
     f4;
     goto a
end
```

5.2.6 Conflict Queues and Compositions

Up to here, a scheduling methodology has been explained which makes use of a schedule specification automaton given explicitly. In this section, a different methodology is described [STZE99]. It is based on an incomplete *FunState* model which does not contain a schedule specification automaton. For the component being scheduled, only the dataflow part is given in a manner similar to a conventional Petri net. Based on this model, the schedule specification automaton is generated automatically. This is the advantage of this methodology because the explicit construction of the schedule specification automaton is not always an easy task. This strategy is sufficient for the conflict behaviors of many kinds of system models with data dependent conflicts. However, some models still require an explicit schedule specification, e. g., if the conflict is caused by external circumstances. Figure 59 shows the modified methodology of the symbolic scheduling approach presented above. New terms used in the following explanations are defined below.

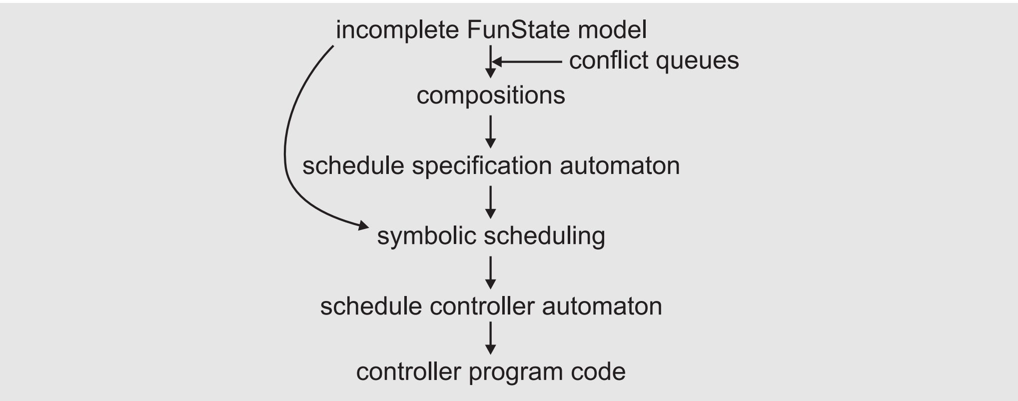

Figure 59. Symbolic scheduling methodology.

5.2.6.1 Preliminaries

As shown in Figure 59, an incomplete *FunState* model of the system being scheduled including scheduling constraints is the basis of the approach. The execution semantics of this model is analogous to that of Petri nets. For the incomplete model, the user has to specify those queues involved in a conflict of functions as explained above. Based on these *conflict queues*, *compositions* of conflicting functions are ascertained. Then the schedule specification automaton is generated automatically. This extended *FunState* model is used for performing the symbolic scheduling procedure as described above, resulting in the controller automaton and its program code.

Def. 5.2.1: **(Conflict queue)** *A* conflict queue *is a multi-reader queue that may contain tokens which only some of the queue's readers consume (depending, i. e., on the token data) but the others do not.*

Queues have to be specified explicitly by the user as conflict queues because this is a semantical property.

Def. 5.2.2: **(Composition)** *A* composition *is the set of all functions which are conflicting with respect to the same conflict queues.*

These conflicting functions are dependent on each other in the sense that they are connected via at least one common conflict queue in their presets.

The dataflow part of the *FunState* model shown in Figure 54 a) on page 111 may be regarded as an incomplete specification in the above sense. Let the light-shaded queue q_1 be a conflict queue. Then the conflicting functions f_2 and f_3 form the corresponding composition.

5.2.6.2 Conflict Resolution

The conflict of a composition is resolved by binding exactly one of the conflicting functions and storing this binding. This is represented by internal states of the schedule specification automaton. Basically, there are two possibilities of resolving a conflict: as soon as possible or just before executing one of the involved functions. The former is called *early* and the latter *late conflict resolution*. This section contains a description of how to generate a schedule specification automaton which models one of these resolution behaviors.

First, early conflict resolution is considered. A conflict is to be resolved as soon as possible, hence at the very beginning of the schedule or immediately after executing one of the conflicting functions of the corresponding composition. Figure 60 on the following page shows a *FunState* model including the schedule specification automaton which represents this resolution behavior. For simplicity, concurrent state machines are used in the FSM part.

The concurrent execution of partial state machines within one component and their communication via events is synchronous as described in Section 2.2.3.1. An in-state operator "M in s" has been introduced of which the result is *true* iff the partial state machine M is in its state s. Note that function calls in transition actions such as "$\ldots/f_2$" serve also as events for communication and thus may be part of a transition predicate such as "f_2" as shown in the right-hand FSM part in Figure 60 on the next page.

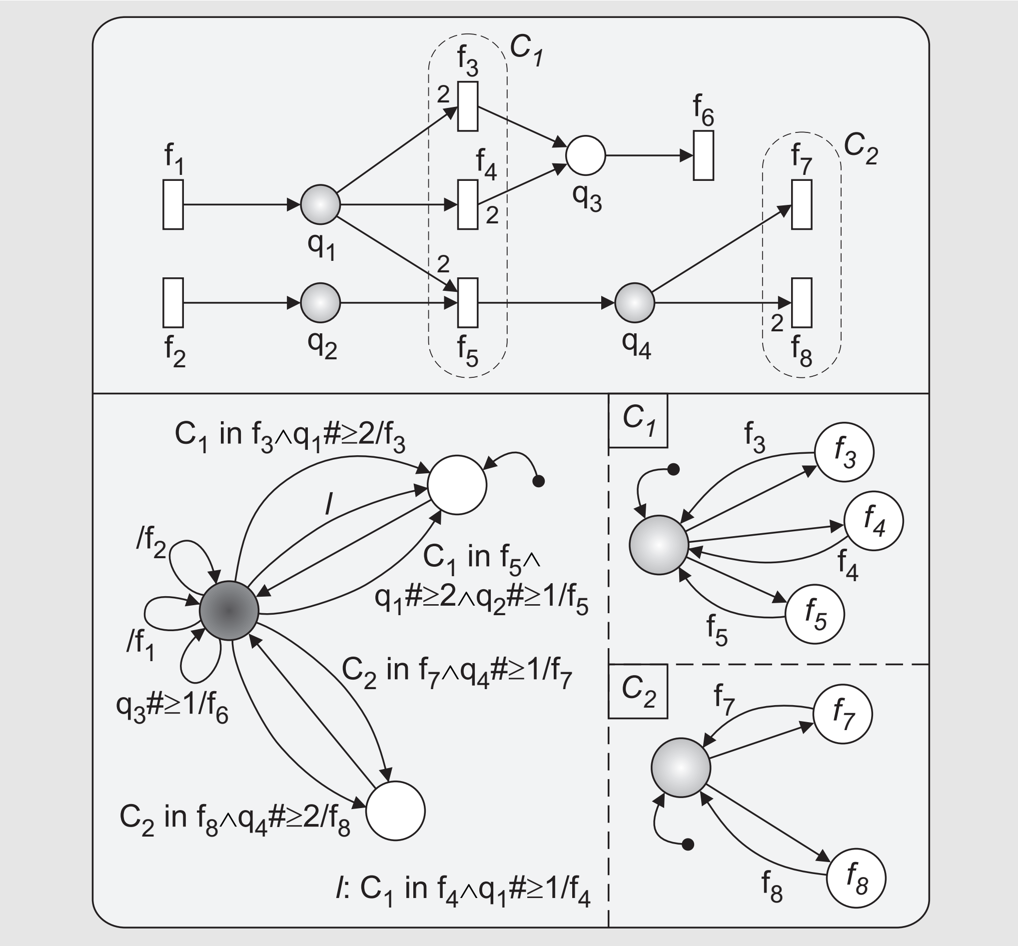

Figure 60. Early conflict resolution.

The queues q_1, q_2, and q_4 have been specified as conflict queues. This results in the compositions $\mathbf{C}_1$ and $\mathbf{C}_2$. The schedule specification automaton has been generated automatically. Each of the conflicts results in a partial automaton that performs the binding of the conflicting functions. Here, each function is represented by a state reachable from a conflict state. An additional partial automaton is used for controlling the entire schedule. It contains the only alternative state.

A drawback of early conflict resolution is that the corresponding regular state transition graph and the resulting schedule contain many similar paths that proceed "in parallel" and of which the visited states differ only slightly. This results from the fact that the execution paths are splitted very early. It is likely that this increases the size of the search space and the resulting schedule.

Late conflict resolution avoids splitting the execution paths too early since this is done only as late as possible. The conflict is resolved only when at least one of the functions could be executed. Figure 61 on the facing page shows the schedule specification automaton for late conflict resolution replacing that in Figure 60.

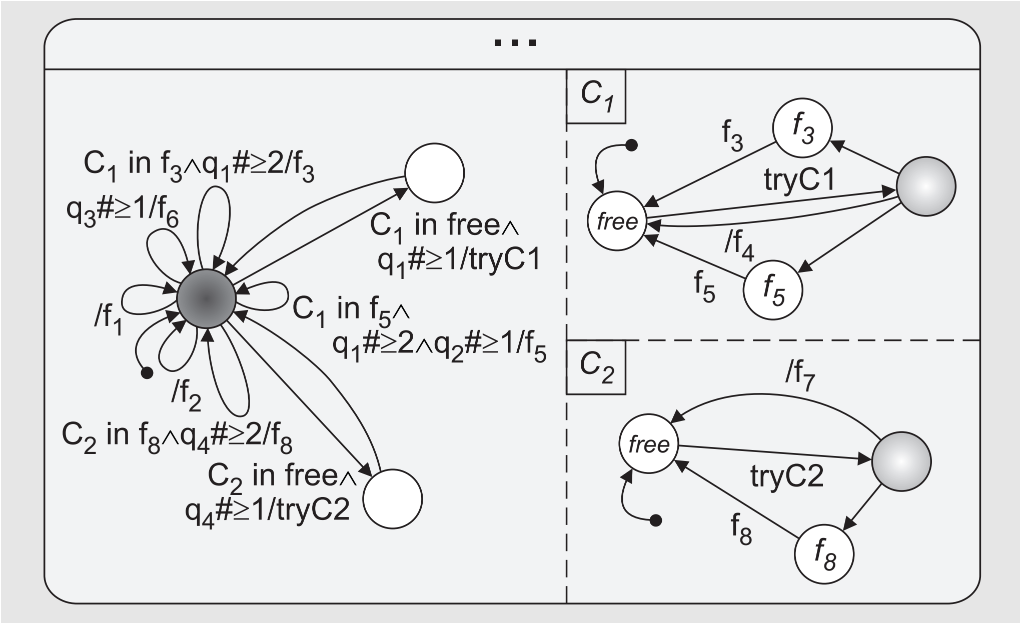

Figure 61. Late conflict resolution.

The events *tryC1* and *tryC2* are defined explicitly for communication between the concurrent automata. The transition predicates for generating these events in the left-hand part of Figure 61 are based on the disjunction of the firing predicates of all conflicting functions of the respective composition. As long as a conflict is unresolved, the corresponding partial automaton stays in its state *free*.

A disadvantage of late conflict resolution is that for certain kinds of models the generated schedule specification automaton is fairly complex, resulting from complex transition predicates and many similar transitions.

5.2.6.3 Strong Firing Conditions

The schedule specification automaton can be greatly simplified by introducing modified firing conditions for conflicting functions within a composition. By abandoning some freedom during scheduling, the size of the search space may be reduced significantly. The constrained *strong firing conditions* enable conflict resolution and thus function execution only when all conflicting functions within a composition are enabled—thus all predicates on the conflict queue contents with respect to the functions' consumption rates are satisfied.

The effect of strong firing conditions is that the execution of the function chosen is performed immediately after the conflict resolution. Hence, the binding does not have to be stored using an internal state. Figure 62 on the next page shows the schedule specification automaton representing the strong firing conditions with respect to Figure 60 on the facing page.

The predicates of the transitions reaching a conflict state are the conjunctions of the conventional firing predicates of the respective conflict-

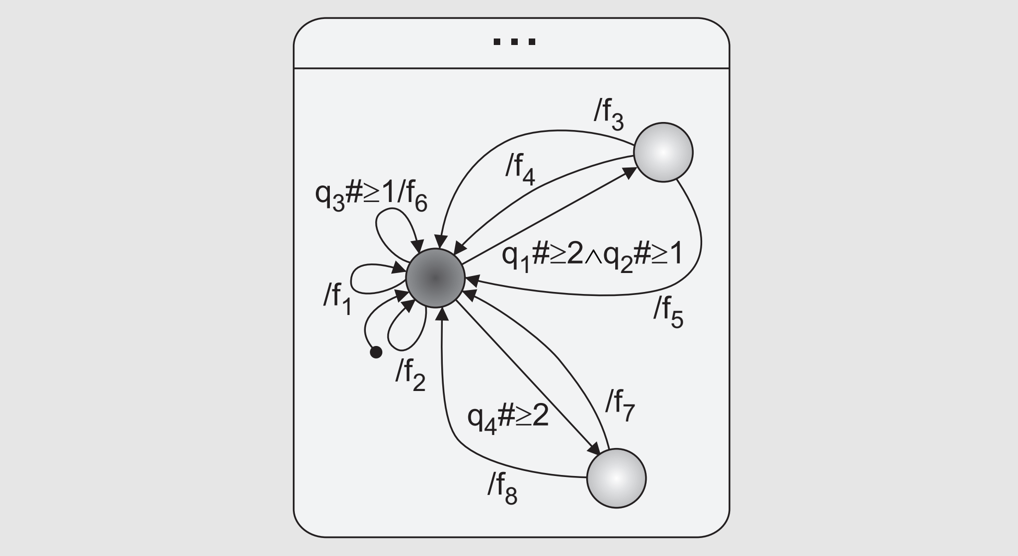

Figure 62. Strong firing conditions.

ing functions. Since some design freedom has been given up, using strong firing conditions may result more often in non-schedulable models than otherwise. Furthermore, resulting schedules may be less efficient. A special class of models obeying these strong firing conditions is equivalent to free-choice Petri nets and equal conflict nets.

5.3 Symbolic Scheduling

Unfortunately, graph traversal tasks such as the search for paths in the dynamic state transition graph used by many scheduling approaches often suffer from the state explosion problem for real-world applications. This means that the possibly exponential blow-up of the number of states to be considered severely restricts the feasibility of such techniques. In order to remedy this shortcoming, symbolic methods have been developed and applied to several scheduling approaches.

Symbolic methods for control dependent scheduling have shown to be effective techniques to perform control/data path scheduling. They often outperform both ILP and heuristic methods while yielding exact results. Moreover, all possible solutions to a given scheduling problem are computed simultaneously such that additional constraints may be applied to find optimal schedules. On the one hand, a closed form solution set can be generated in which all satisfying schedules for arbitrary forward branching control/data paths and resource constraints are encapsulated in a compressed BDD-based representation [RB94]. On the other hand, the design space of the schedule specification can be represented by a finite state machine from which a set of Boolean formulae is generated and manipulated in order to obtain a solution [CD94, HB98]. This approach recently has been extended by non-determinism, hierarchy, and abstraction [HB99]. Symbolic control/data path scheduling mostly is performed on the *register-transfer level* (RTL) which is located below the abstraction levels *FunState* is dedicated to. BDDs are well suited to represent RTL-based models.

A symbolic approach to conflict-dependent scheduling described in Section 5.2 has been introduced in [STZ+99] which uses interval diagram techniques and symbolic model checking principles in order to avoid the explicit enumeration of execution paths. The approach makes use of Algorithm 6 on page 115 which is refined as described in the following. Either interval diagram techniques or conventional BDD techniques can be used for performing the symbolic computations.

The shortest path criterion mentioned in Section 5.2.4 is assumed as optimization criterion. The dynamic state transition graph is traversed symbolically without constructing it explicitly. This is achieved by iterative image computation as explained in Section 3.2.5. An interval mapping diagram such as shown in Figure 54 b) on page 111 represents the transition relation, while interval decision diagrams are used for storing intermediate state sets. The efficiency of these techniques has been shown for the similar task of formal verification in Section 4.2.

This way, shortest paths are determined by symbolic breadth-first searches. Sets of states reachable from another set of states are considered—and computed in one single operation—instead of traversing the state transition graph path after path, state by state.

In Algorithm 8, a possible algorithm of *findPath*$(a, \mathbf{B})$ is described where a is a state and $\mathbf{B}$ is a set of states which may include a. The result is one—out of possibly several—shortest path $x_0 \xrightarrow{t_0} x_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} x_n$ with respect to a transition relation $\mathbf{T}$ from a to any element of $\mathbf{B}$ with $n \geq 1$. The algorithm is akin to those used for determining counter-examples and witnesses in symbolic model checking.

Alg. 8. Shortest path search

1 **funct** *findPath*$(a, \mathbf{B}) \equiv$
2 $\lceil \mathbf{S}_0 := \{ a \};\ n := 0;$
3 **do**
4 $\quad \mathbf{S}_{n+1} := Im(\mathbf{S}_n, \mathbf{T});$
5 $\quad n := n + 1$
6 **until** $\mathbf{S}_n \cap \mathbf{B} \neq \emptyset$ **od**;
7 choose a state $x_n \in \mathbf{S}_n \cap \mathbf{B}$;
8 **for** $i := n - 1$ **downto** 0 **do**
9 $\quad$ choose a state $x_i \in \mathbf{S}_i \cap PreIm(\{ x_{i+1} \}, \mathbf{T})$ **od**;
10 **return** $x_0 \xrightarrow{t_0} x_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} x_n \rfloor$.

The image operator $Im(\mathbf{S}, \mathbf{T})$ and its inverse $PreIm(\mathbf{S}, \mathbf{T})$ are introduced in Section 3.1.3. The choice of one state out of a set of states at lines 7 and 9 of Algorithm 8 on the preceding page preferably is done by selecting non-conflict states. This is a heuristic criterion with the purpose of reducing the number of conflicts to be considered and hence the size of the search space and the resulting schedule.

The plain RSM-based scheduling approach presented in [TTS00] which has been applied to Petri nets potentially is more efficient than the *FunState*-based approach described here because additional internal states are avoided. Nonetheless, the schedule modeling capacities of the RSM approach are limited compared with *FunState*.

5.4 FunState Scheduling Extensions

In previous sections, solely the untimed execution behavior of *FunState* has been used for scheduling. In the following, the timed *FunState* model is considered. It is sketched how timing constraints can be imposed on the resulting schedule and how timing properties of the *FunState* model can be used to find efficient schedules. Further, an approach to parallel scheduling in a multi-processor setting is presented, based on *FunState*.

To concentrate on the essentials, in this section no conflicts are involved. Hence, resulting schedules are completely static and can be considered to be optimal with regard to execution time. This results from the use of the shortest path criterion during the scheduling method, as described in Section 5.2.4.

5.4.1 Timed Scheduling

Analogously to formal verification of the explicitly timed *FunState* model of computation, timing behavior is taken into consideration by combining *FunState* with timed automata, as described in Section 4.3.4. Mainly, there are two possibilities of representing timing constraints, either implicitly in the underlying model of computation or explicitly in the system being modeled. Both means are described in the following.

5.4.1.1 Repetitious Deadlines

A repetitious deadline describes the restriction that the time between two activations of a function f_i of latency τ_i must be at least d_i and at most D_i, where $\tau_i \leq d_i \leq D_i$. Thus, minimum and maximum execution frequencies can be specified. This kind of constraint is best incorporated internally to the model of computation by restricting the execution behavior of the *FunState* model. For this purpose, the timed execution behavior of functions in *FunState* is constrained by a timed automaton as shown in Figure 63 on the next page instead of that in Figure 49 on page 99. In this context, it is assumed that the first firing of function f_i the earliest takes place after time d_i and the latest after time D_i. This initial restriction can easily be modified by altering the timed automaton.

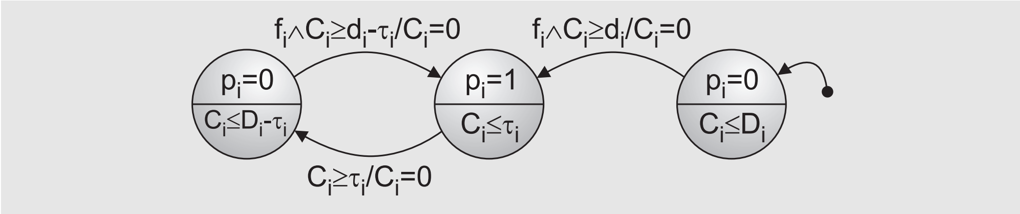

Figure 63. Timed automaton modeling repetitious deadline of *FunState* function.

Again, the processing state of function f_i is annotated with $p_i = 1$. When this state is left, f_i is finalized. The left-hand non-processing state included in the cycle has to be left after a time of at least d_i and at most D_i after the last invocation of f_i.

Setting $d_i = 0$ or $D_i \to \infty$ voids timing constraints imposed. By setting $d_i = \tau_i$, the forbidden multiple instantiation of functions (comparable with a pipelining effect) can be avoided inherently which means that f_i cannot be fired again while it is in execution. By further assigning $D_i = \tau_i$, a strictly periodic execution of f_i can be enforced. In the following, $d_i = \tau_i$ and $D_i \to \infty$ is assumed if not stated otherwise.

In Figure 64, a timed *FunState* model is shown of which function f_3 is contrained by repetitious deadlines $d_3 = 8$ and $D_3 = 12$.

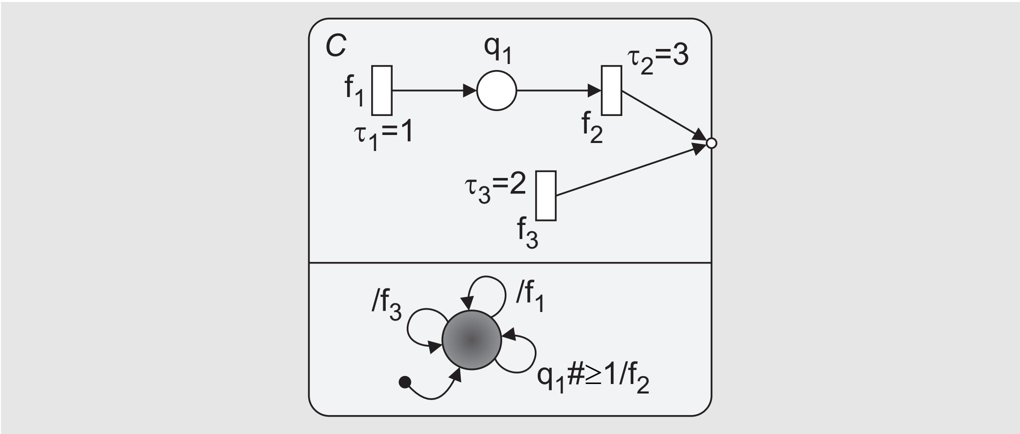

Figure 64. *FunState* model with repetitious deadlines.

Figure 65 on the following page shows the resulting controller automaton respecting the timing constraints imposed. The controller describes the periodic schedule $f_1\,f_2\,f_1\,f_2\,f_3\ldots$ which exhibits a period of 10.

Note that $d_i = \tau_i$ still is assumed which results in a different execution semantics of the *FunState* model. This is due to the fact that the timed automata extension restricts the execution of the sole *FunState* model. Namely a transition can only be executed if the annotated function is ready for being fired. This different semantics can be circumvented by either replacing each function involved by a function-queue-function combination or by interpreting the *FunState* model of the controller automaton using the untimed *FunState* execution behavior.

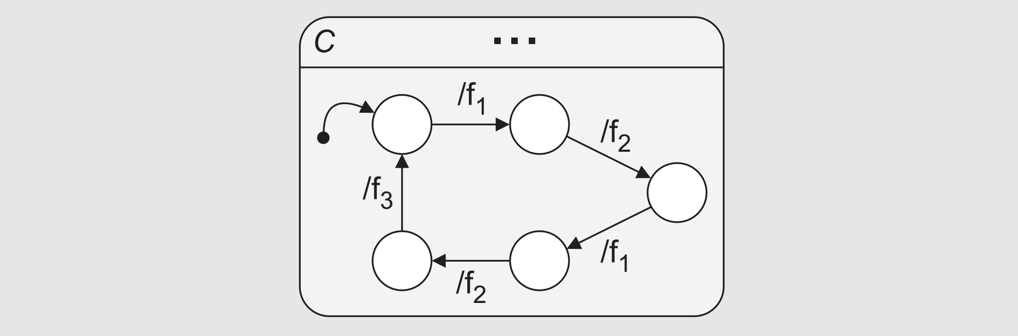

Figure 65. Controller automaton respecting repetitious deadline.

5.4.1.2 Periodic Deadlines

Periodic deadlines as often used in RTOS[2] applications prescribe that, for instance, a function is fired exactly once within each fixed period of time. This kind of deadline can be modeled using additional elements in the *FunState* model combined with repetitious deadlines as given above. In Figure 66, for the functions f_3 and f_4 periodic deadlines τ_1 and τ_2 are specified, respectively.

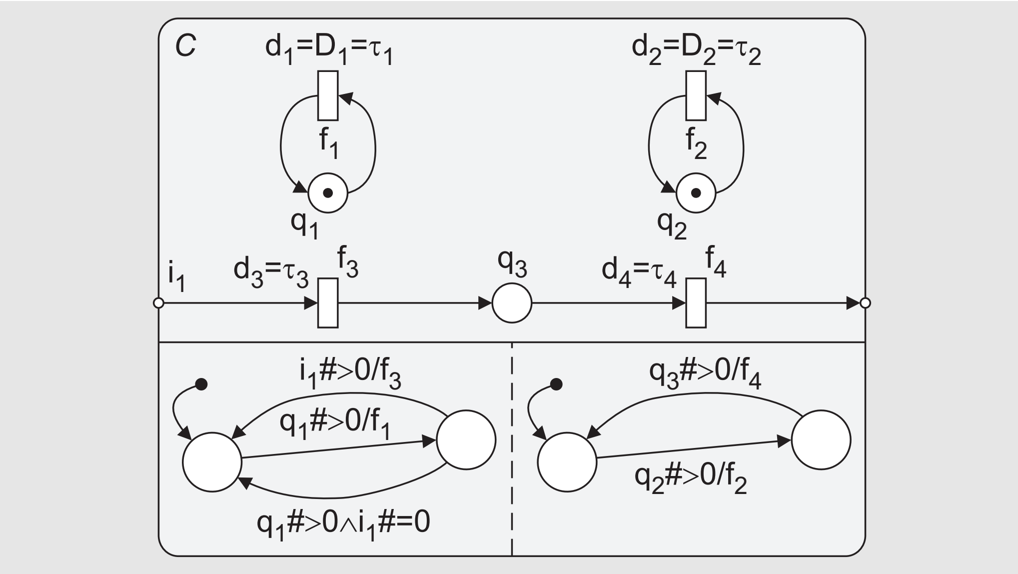

Figure 66. Periodic function execution deadlines modeled in *FunState*.

Function f_3 has to be executed only if sufficient input data are available, whereas f_4 has to be executed in any case within the given period of time. $\tau_3 \leq \tau_1$ and $\tau_4 \leq \tau_2$ is assumed. Similar timing constraints such as periodic deadlines with regard to function finalization require only slight modifications of the *FunState* model.

[2]RTOS: *real-time operating system*

5.4.2 Parallel Scheduling

If a feasible schedule for a single processor exists, then a workable schedule for a multi-processor system can also be generated. While trivially only one of the processors could be used, in general run time can be reduced significantly by more even load distribution.

Essentially, parallel scheduling makes sense only if the explicitly timed execution behavior is considered. Otherwise, all functions would be considered to have the same latencies. This could result in less efficient schedules because some relative timing information on function latencies got lost. Hence, timed scheduling as introduced in Section 5.4.1 is assumed in the following. Timing constraints presented there can be used for parallel scheduling as well.

Figure 67 shows a *FunState* model of which both components A and B are to be implemented on two different processors. Hence, using information about function latencies and component interaction can yield more efficient schedules because then function executions can be overlapped in time when mapped onto different processors. Again, $d_i = \tau_i$ is assumed.

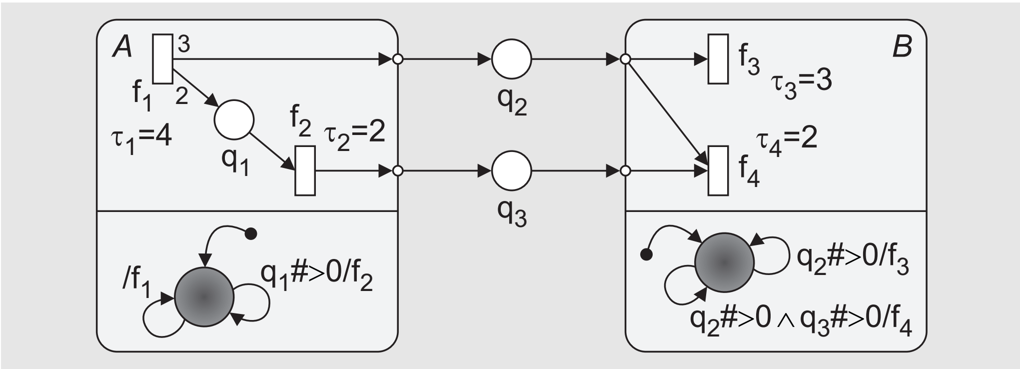

Figure 67. *FunState* model specifying parallel schedule.

The resulting periodic schedule controlling both components is shown in Figure 68 a) on the next page. Separating function firings of A and B from each other yields the controller automata of the respective components shown in Figure 68 b).

The parallel schedule described by the controller automata of A and B is shown in Figure 69 on the following page. Note that advantage has been taken of the possibility of parallel execution of functions on different processors which results in overlapping function executions.

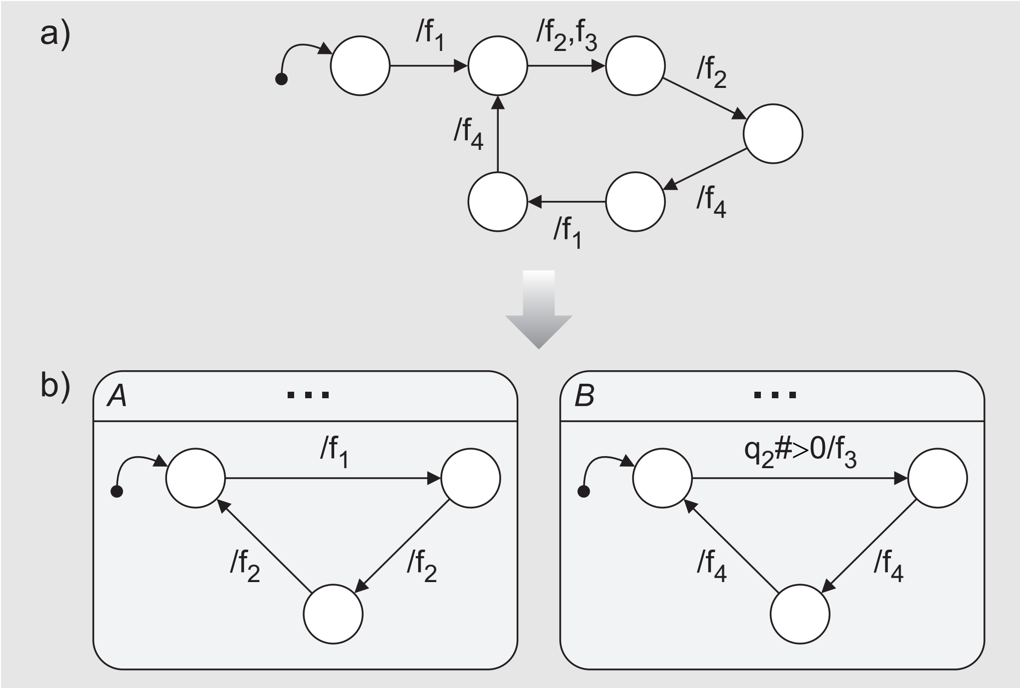

Figure 68. State transition graph of parallel schedule and resulting controller automata.

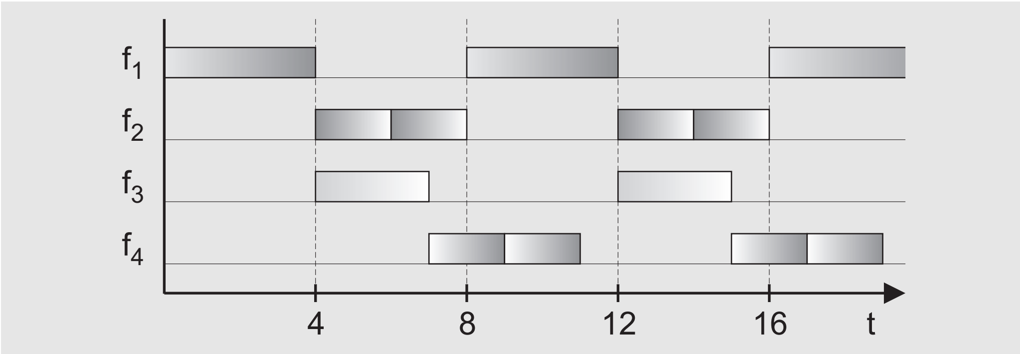

Figure 69. Gantt chart describing parallel schedule.

5.5 Example Models

5.5.1 Molecular Dynamics Simulation

The introduced untimed scheduling approach has been applied to performing conflict-dependent scheduling of a molecular dynamics simulation system. As shown in Figure 70 on the next page, the simplified fundamental simulation algorithm has been mapped onto a host workstation (*Host*) linked to a special purpose hardware accelerator serving as a coprocessor (*CoPro*). The concurrent execution of the state machines of *Host* and *CoPro* is assumed to be asynchronous and interleaved for simplicity[3].

[3]This can simply be achieved by employing a controlling state machine on a higher level, as shown in Figure 16 on page 32, for instance.

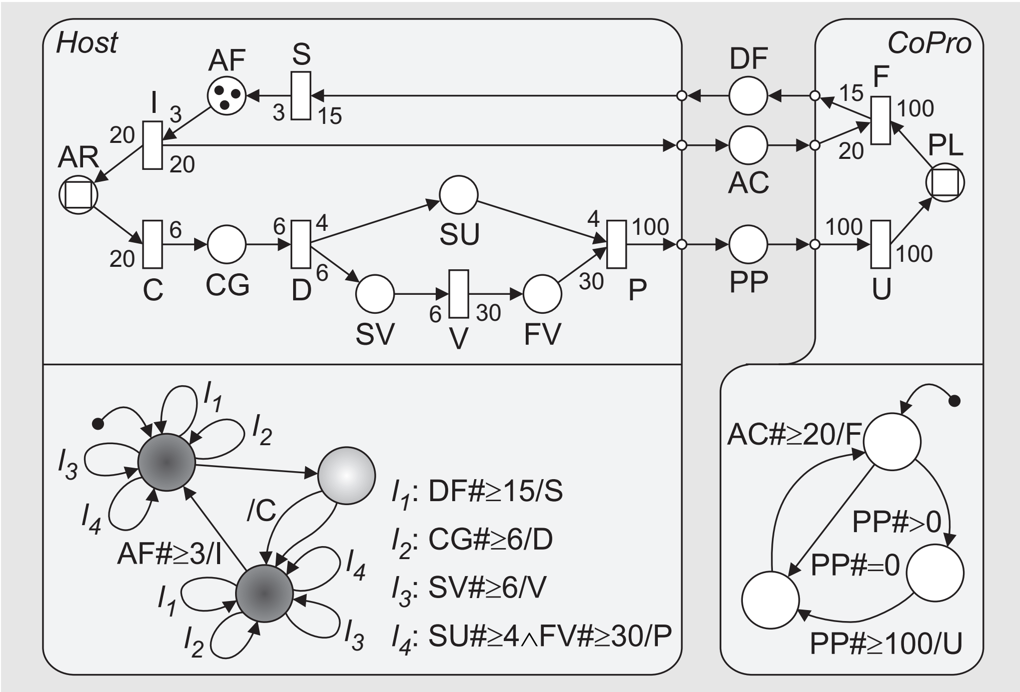

Figure 70. Molecular dynamics model with schedule specification automaton.

The simulation mainly consists of repeated computations in the feedback loop distributed among both processors where atom forces (*AF*) are computed (*F*), added up (*S*), and integrated (*I*) to calculate new atom coordinates (*AC*, *AR*). After a variable number of iterations, the central coordinates of slowly moving sub-molecules called charge groups (*CG*) are updated (*C*). Then, a new list of neighbors called pair list (*PL*) is computed (*D*, *V*, *P*, *U*).

The state machine of component *Host* describes a specification of possible schedules for *Host* (item 1 of the methodology description on page 107). Since the moment when to start this pair list computation is unknown until run time, this fact represents a conflict which is modeled using a conflict state. The major issue of the schedule specification is that there exists no cycle in the corresponding state transition graph which does not contain the conflict state. This is ensured by the fact that the transition executing *I* cannot be reached without visiting the conflict state. The schedule specification automaton has been given explicitly. The specification automata of more complex systems may be given by a concurrent representation for clarity (omitted here for the sake of simplicity).

Based on the above schedule specification, the symbolic techniques for conflict-dependent scheduling as described in Section 5.2.4 are performed (item 2). From the resulting state transition graph, transitions of the state machine of component *CoPro* have to be extracted. The result is the schedule controller automaton shown in Figure 71 on the following page (item 3) which may replace the automaton of component *Host* in

Figure 70 on the page before for analysis or synthesis purposes (item 4). The controller automaton consists of two static cycles and a conflict state switching between them. The schedule is respecting the specification of *CoPro*. Note that even the schedule of *CoPro* is not static as it depends on the content of queue *PP*. For implementational efficiency, the original comparison $DF\# \geq 15$ has been automatically replaced by the non-zero test $DF\# > 0$.

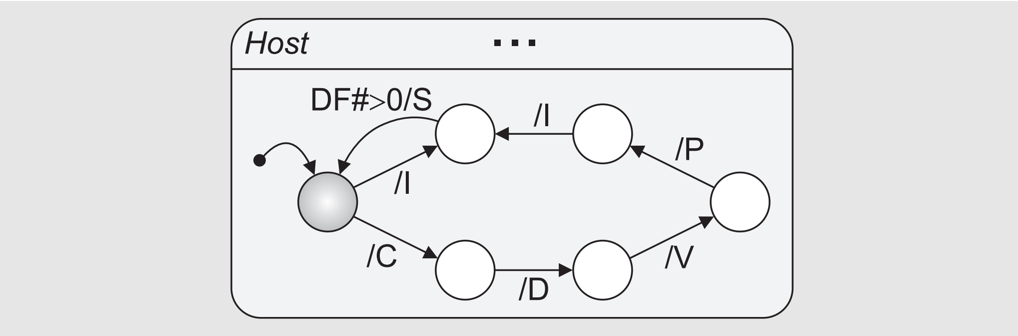

Figure 71. Resulting controller automaton.

The controller automaton can easily be transformed into program code as shown in Algorithm 9 as pseudo code.

Alg. 9. Controller program code of molecular dynamics model

```
begin
  a: if p
       then C; D; V; P fi;
     I;
     while DF# = 0 do od;
     S;
     goto a
end
```

5.5.2 ATM Switch

In this section, the approach to conflict-dependent scheduling is applied to a model of a shared memory ATM switch mapping ATM connections arriving at four input ports onto connections leaving on four output ports. Figure 72 on the next page shows the *FunState* model of the ATM switch including a schedule specification.

The model imitates the tasks in the ATM user plane of the ATM layer [ATM94] which are necessary for the switching of ATM cells. The interface to the physical layer is modeled according to the UTOPIA [ATM95] specification using the octet-level handshake mode.

All input and output buffers are realized in a single memory space which is shared by all ports. This space is subdivided into pieces which

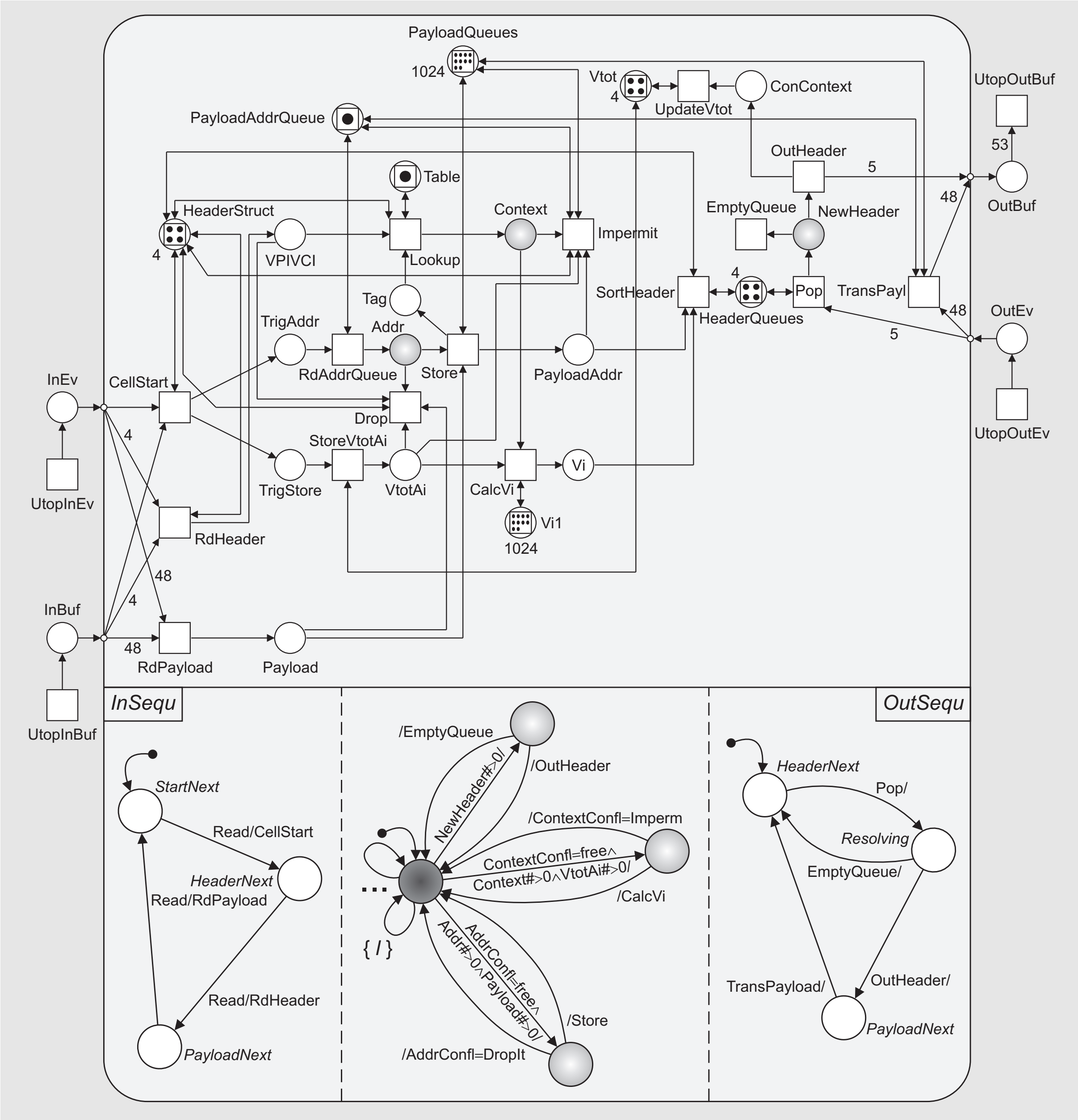

Figure 72. ATM switch model with schedule specification.

are large enough to store the information of an ATM cell. Usually, data and scheduling information for the output ports are stored separately. The 48 bytes data field (so-called payload) can be stored at arbitrary addresses in the shared memory since the address is stored in a FIFO organized queue of the corresponding connection together with scheduling information and the cell header field (five bytes). Another FIFO queue keeps track of addresses of available free memory pieces for storing payload fields. For each output port, there is a scheduler that must decide which connection will be allowed next to transfer a cell if a cell slot becomes available on the output port. Recently, variants of the weighted fair queuing scheduling algorithm [Zha95] have been used for this task.

The algorithm calculates a priority tag for each incoming ATM cell which keeps the reserved and currently used bandwidth of the corresponding connection in mind. Then, the cell with the highest priority tag is chosen by the scheduler for transmission.

This functionality is modeled as follows. Incoming ATM cells are processed byte by byte, by the UTOPIA interface. For each byte, the interface generates an event (*UtopInEv*) and produces a token carrying one byte of information (*UtopInBuf*). The first five bytes of the cell contain the cell header. When the beginning byte of a cell is signaled by the interface, the switch starts several tasks. In order to find a free memory location for the storage of the cell payload, the address queue is read out (*RdAddrQueue*). If no memory space is available, the whole ATM cell will be dropped (*Drop*). Furthermore, a priority tag must be calculated (*CalcVi*) using a virtual time measure at the arrival time of the cell (*StoreVtotAi*). This tag is used later on to schedule cells of different connections sharing the same output port. After the whole header has been transferred from the interface into the switch (*RdHeader*), the cell is mapped onto an output port (*Lookup*) according to connection information stored in a so-called lookup table and in the cell header itself. The header can now be stored in one of the output port queues of which each is sorted by increasing priority (*SortHeader*). The payload is stored independently in the FIFO organized queue of the corresponding connection (*Store*) at the address determined before. However, if the lookup ascertains an impermissible cell, e. g., if the cell belongs to a non-existing connection, the cell will be dropped (*Impermit*).

The readout of cells via the output ports is triggered by the corresponding output UTOPIA interface, again byte by byte. It generates events (*UtopOutEv*) for each free byte in the output buffer of the interface. If sufficient free byte slots are available, the cell content is reconstructed (*Pop*, *TransPayl*) by concatenating header and payload and then transferred to the corresponding output port buffer (*UtopOutBuf*). However, if there is no cell stored in the switch for transmission, the event tokens generated by *UtopOutEv* will be dropped by *EmptyQueue*. Finally, each cell transmission updates the global virtual time measure (*UpdateVtot*) used for priority tag calculations.

The following transition labels are used which correspond to the transitions abbreviated by "$\{l\}$" and "…" in Figure 72 on the preceding page, each starting and ending in the dark-shaded state.

$$
\begin{aligned}
\mathit{OutSequ}\ \text{in}\ \mathit{HeaderNext} \wedge \mathit{OutEv\#} \geq 5 &\ /\ \mathit{Pop} \\
\mathit{OutSequ}\ \text{in}\ \mathit{PayloadNext} \wedge \mathit{OutEv\#} \geq 48 &\ /\ \mathit{TransPayl} \\
\mathit{InSequ}\ \text{in}\ \mathit{StartNext} \wedge \mathit{InEv\#} > 0 \wedge \mathit{InBuf\#} > 0 &\ /\ \mathit{Read} \\
\mathit{InSequ}\ \text{in}\ \mathit{HeaderNext} \wedge \mathit{InEv\#} \geq 4 \wedge \mathit{InBuf\#} \geq 4 &\ /\ \mathit{Read} \\
\mathit{InSequ}\ \text{in}\ \mathit{PayloadNext} \wedge \mathit{InEv\#} \geq 48 \wedge \mathit{InBuf\#} \geq 48 &\ /\ \mathit{Read} \\
\mathit{ContextConfl} = \mathit{Imperm} \wedge \mathit{PayloadAddr\#} > 0 &\ /\ \mathit{Impermit}, \mathit{ContextConfl} = \mathit{free} \\
\mathit{AddrConfl} = \mathit{DropIt} \wedge \mathit{VPIVCI\#} > 0 \wedge \mathit{VtotAi\#} > 0 &\ /\ \mathit{Drop}, \mathit{AddrConfl} = \mathit{free} \\
\mathit{TrigAddr\#} > 0 &\ /\ \mathit{RdAddrQueue}
\end{aligned}
$$

$$
\begin{aligned}
TrigStore\# > 0 \;/\; StoreVtotAi \\
VPIVCI\# > 0 \wedge Tag\# > 0 \;/\; Lookup \\
ConContext\# > 0 \;/\; UpdateVtot \\
PayloadAddr\# > 0 \wedge Vi\# > 0 \;/\; SortHeader
\end{aligned}
$$

In addition to the explicit event *Read*, function calls in the FSM part are also used as events for communication between concurrent state machines. For the sake of clarity, two state variables $ContextConfl \in \{free, Imperm\}$ and $AddrConfl \in \{free, DropIt\}$—both initialized with *free*—have been introduced which could be replaced by further concurrent state machines, each with two states.

The queues involved in conflicts are marked by shading in Figure 72. All three conflicts have the common characteristic that usually one of their conflicting functions ($CalcVi, Store, OutHeader$) is executed and the other one ($Impermit, Drop, EmptyQueue$) only in case of an irregular operation which depends on the value of the token in the respective queue.

The *NewHeader* conflict differs from both the *Context* and the *Addr* conflict in that the transition predicates of both conflicting functions *OutHeader* and *EmptyQueue* are equal. Hence, the conflict model represents the fact that if one of the transitions is enabled, the other one is enabled as well. Then, based on the value of the token in *NewHeader* the conflict can be resolved directly by executing either *OutHeader* or *EmptyQueue*.

The latter is not the case for the *Context* conflict (and the *Addr* conflict as well). If *CalcVi* is enabled, then *Impermit* is also enabled and can be executed, but not vice versa. If the execution predicate of *CalcVi* is satisfied, the conflict is resolved based on the token in *Context*: Either *CalcVi* is executed directly or *Impermit* has to be executed—which possibly cannot be done due to its unsatisfied predicate. In order to avoid the necessity of repeatedly evaluating the conflict decision or of delaying the execution of *CalcVi*, the result of the conflict resolution is stored by changing the internal state of the conflict model and thus binding the conflict to *Impermit* by setting $ContextConfl = Imperm$. Later on, only the remaining execution predicates of *Impermit* have to be checked which results in less dynamic decision overhead than otherwise. The value *free* represents an unresolved conflict, while the values *Imperm* and *DropIt* denote that a conflict is bound to the respective transition.

For the sake of clarity, transitions resulting from the asynchronous behavior of the interfaces to the environment are disregarded here. Techniques for the automatic generation of a *FunState* model representing the conflict behavior from a given Petri net model as shown in Figure 72 have been described in Section 5.2.6.

Based on the *FunState* model in Figure 72, the conflict-dependent scheduling procedure introduced above has been applied. The dynamic state transition graph of the schedule resulting from the application of $scheduleGraph(x_0, \{x_0\})$ has been transformed into the controller automaton shown in Figure 73 on the next page. The result is a scheduling policy which may be implemented, e. g., as a software controller on a uniprocessor.

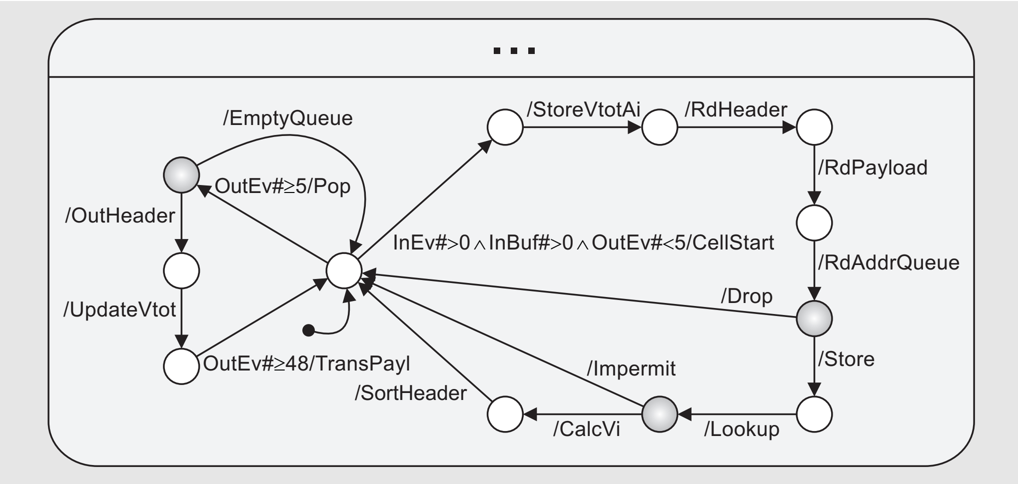

Figure 73. Resulting controller automaton of ATM switch model.

Conflict decisions remaining in the resulting schedule again are represented by conflict states in Figure 73. Besides conflict decisions—which cannot be resolved during compile time—, only three decisions had to be postponed until run time. Hence, the overhead by such dynamic decisions has been drastically reduced in comparison with the original *FunState* model. The scheduling process has been performed mainly using the symbolic model checking tool SMV by Carnegie Mellon University. The computation took 11.1 s on a Sun Ultra™ 60 with 360 MHz.

For an implementation, the state machine representing the schedule may be transformed easily into program code as shown in Algorithm 10 on the next page as pseudo code. The predicates p_i at lines 3, 10, and 13 identify run-time decisions associated with the respective conflicts and depending on token values.

Alg. 10. Program code representing schedule of ATM switch model

```
begin
  a: if OutEv# ≥ 5
       then if p_NewHeader
              then EmptyQueue; goto a fi;
            OutHeader; UpdateVtot;
            while OutEv# < 48 do od;
            TransPayl
     elsif InEv# > 0 ∧ InBuf# > 0
          then CellStart; StoreVtotAi; RdHeader; RdPayload; RdAddrQueue;
               if p_Addr
                 then Drop; goto a fi;
               Store; Lookup;
               if p_Context
                 then Impermit; goto a fi;
               CalcVi; SortHeader fi;
     goto a
end
```

6

Concluding Remarks

In this dissertation, the *FunState* model of computation has been presented as an internal design representation model for embedded systems design. *FunState* enables the explicit representation of non-determinism and scheduling using a combination of functions and state machines. The *FunState* model unifies many different well-known models of computation, supports stepwise refinement and hierarchy, and is suited to represent many different synchronization, communication, and scheduling policies.

As has been explained in this work, the *FunState* model enables the internal representation of complex system behavior. In order to cope with design complexity, the following hierarchical step-by-step approach is advertised and supported by the *FunState* model:

1. Restriction in some portions of the system, i. e., components, to well-known and simple models of computation. Within these subsystems, specialized and adapted methods can be applied.

2. Making use of the hierarchical composition to design hierarchical methods. To this end, it should be possible to restrict the scope of, e. g., scheduling, code generation, or verification to one component. Consequently, its environment as well as its embedded components must be simplified without sacrificing the quality or correctness of the overall yield. This simplification should take into account, for instance, previous scheduling or formal verification results for parts of the environment or embedded components.

It has been shown that the *FunState* model supports the first item as it can represent different important elementary models of computation. The major property required for the second item is abstraction. It has been shown in this monograph that the *FunState* model is able to represent scheduling results produced by some dedicated scheduling techniques.

Several methods involved in this iterative design process have been described in this monograph, namely formal verification and scheduling. Here, the focus has been put on symbolic forms which provide advantages with regard to computational resources required. Applications investigated are summarized in the following.

6.1 Symbolic Model Checking

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 described by means of huge BDDs. Especially for process networks, traditional approaches have shown not to be feasible due to shortcomings mentioned in this work. As an alternative, a different approach to symbolic formal verification of process networks and related models of computation has been presented. It is based on an efficient form of representation of multi-valued functions called interval decision diagram (IDD) and the corresponding image computation technique using interval mapping diagrams (IMDs).

IMDs are dedicated to representing the transition relation of the considered kind of models, describing transitions by combinations of predicates for the firing conditions and state distances for the production and consumption rates. Several drawbacks of conventional symbolic model checking of process networks with BDDs are avoided due to the use of interval diagram techniques. Especially the resulting transition relation IMD is very compact, enabling fast image computation. Furthermore, no artificial limitations concerning buffer capacities have to be introduced. The experimental results significantly outperform those of the BDD approach with regard to computation time and memory.

6.2 Symbolic Real-Time Verification

An approach to formal verification of real-time systems modeled by means of timed automata has been presented which makes use of interval diagram techniques as well. The verification methods necessary for timed analysis based on IDDs and IMDs have been explained. Reachability analysis in a multi-clock setting over discrete time has been considered, and extensions for real-time symbolic model checking have been described. The results outperform those of the NDD approach with regard to computation time by a factor of more than 7 and likely even more.

Both the approaches to symbolic model checking of process networks and to symbolic real-time verification of timed automata have been combined to enable the symbolic formal verification of the timed *FunState* model.

6.3 Symbolic Scheduling

The approach to conflict-dependent schduling of process networks has been presented, exemplified with the *FunState* model of computation. Mixed data/control flow specifications and different mechanisms of non-determinism as occurring in the design of embedded systems are dealt with. Symbolic techniques avoid an explicit construction of the entire system state space. The whole approach can be interpreted as a stepwise reduction of the non-determinism in a system specification and hence as a refinement step with regard to *FunState*'s role as an internal design representation model.

The conflict-dependent scheduling approach yields a deadlock-free and bounded schedule if one exists. Sequences of statically scheduled basic blocks are called dynamically at run time by the generated schedule. Extensions have been given to allow even timed and parallel scheduling.

6.4 Future Prospects

Several open issues still have to be considered. With regard to the *FunState* model of computation, some semantical details have to be investigated and prescribed definitely.

- The execution of components has been defined to always be synchronous with regard to the triggering transition. Instead, a trigger event could invoke several subsequent state transitions in embedded components, as mentioned in Section 2.2.3.3.

- To describe scheduling policies in a hierarchical manner, it proves useful to enforce transitions within embedded components from the outer component's state machine, whereas the defined semantics only allows to enable transitions. No direct feedback from the inner state machine occurs reporting whether some transition has been taken. This could be remedied by adding a second class of transitions which are executed only if components triggered by them can execute as well.

- On the other hand, transitions could be introduced which are coupled with function executions invoked by them. This would add the concept of timed execution even to state machine transitions.

It is desirable that a designer can fall back on more intuitive input formalisms for conflicts and alternatives as well as for timing constraints. This is connected with the need for a unifying framework where several input formalisms can be mapped onto *FunState* representations which then are analyzed or refined as stated above. Likewise, the relationship between *FunState* and the SPI model of computation needs further investigation as well as support by methods and tools for integration and transformation into each other.

In the context of conflict-dependent scheduling, the following issues are of interest.

- The symbolic techniques presented in this work might be compared and combined with enumerative methods exploring and making use of partial order dependencies between function firings.

- With regard to computational complexity, the question is still open if conflicts in a timed or parallel scheduling setting can be handled efficiently.

- Resource constraints may be incorporated in the state machine part of a *FunState* model.

Furthermore, properties specified within uncertainty intervals may be considered as conflicts and by this can be involved in the scheduling approach. This especially holds with respect to the SPI model of computation, where uncertain specifications are even more intrinsic.

Concerning the underlying symbolic methods, recently considerable efficiency improvements have been achieved regarding certain conventional symbolic model checking problems. Instead of binary decision diagrams, satisfiability (SAT) solvers have been used [BCC$^+$99, BCCZ99] for finding witnesses or counterexamples. Therefore, the system being verified as well as the temporal property being checked are translated into some propositional formula of which then satisfiability is solved. A similar SAT-based approach can be used for symbolic scheduling as well—possibly increasing efficiency significantly.

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.

[ABK+97] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli, and A. Rasse. Data-structures for the verification of timed automata. In O. Maler, editor, *Hybrid and Real-Time Systems*, volume 1201 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[ABL98] L. Aceto, A. Burgueño, and K. G. Larsen. Model checking via reachability testing for timed automata. In B. Steffen, editor, *Proceedings of the 4th International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 263–280. Springer-Verlag, 1998.

[ACD90] R. Alur, C. Courcoubetis, and D. L. Dill. Model checking for real-time systems. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 414–425, 1990.

[ACD93] R. Alur, C. Courcoubetis, and D. L. Dill. Model checking in dense real-time. *Information and Computation*, 104(1):2–34, May 1993.

[ACH97] R. Alur, C. Courcoubetis, and T. A. Henzinger. Computing accumulated delays in real-time systems. *Formal Methods in System Design*, 11:137–155, 1997.

[Ack82] W. B. Ackerman. Data flow languages. *Computer*, 15(2):15–25, February 1982.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AFH97] R. Alur, L. Fix, and T. A. Henzinger. A determinizable class of timed automata. In *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 1997.

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

[ATM94] The ATM Forum Technical Committee. *ATM User-Network Interface Specification, Version 3.1*. ftp://ftp.atmforum.com/pub/approved-specs/af-uni-0010.002.pdf.tar.Z, September 1994.

[ATM95] The ATM Forum Technical Committee. *UTOPIA, an ATM-PHY Interface Specification, Level 2, Version 1.0*. ftp://ftp.atmforum.com/pub/approved-specs/af-phy-0039.000.pdf, June 1995.

[BCC+99] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th Design Automation Conference (DAC '99)*, pages 317–320, 1999.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[BCG+97] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Press, 1997.

[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):401–424, 1994.

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *5th IEEE Annual Symposium on Logic in Computer Science*, pages 428–439, 1990.

[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.

[BCMD90] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th IEEE/ACM Design Automation Conference (DAC '90)*, pages 46–51, 1990.

[BHK99] W. Boßung, S. A. Huss, and S. Klaus. High-level embedded system specifications based on process activation conditions. *Journal of VLSI Signal Processing*, 21(3):277–291, July 1999.

[BLL+95] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal—a tool suite for automatic verification of real-time systems. In *Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, 1995.

[BLP+99] G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 341–353. Springer-Verlag, 1999.

[BMPY97] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[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 (DAC '90)*, 1990.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[BST92] W. Backes, U. Schwiegelshohn, and L. Thiele. Analysis of free schedule in periodic graphs. In *4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 333–342, San Diego, California, June 1992.

[Buc93] J. T. Buck. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Technical Report UCB/ERL 93/69, Ph.D dissertation, Dept. of EECS, UC Berkeley, California, 1993.

[BWE+93] G. Bilsen, P. Wauters, M. Engels, R. Lauwereins, and J. Peperstraete. Development of a static load balancing tool. In *Proceedings of the 4th Workshop on Parallel and Distributed Processing*, pages 179–194, Sofia, Bulgaria, 1993.

[CBM89a] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer-Verlag, 1989.

[CBM89b] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using Boolean functional vectors. In L. J. M. Claesen, editor, *Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, 1989.

[CC94] S. V. Campos and E. M. Clarke. Real-time symbolic model checking for discrete time models. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-Time System Development*, AMAST Series in Computing. World Scientific, 1994.

[CCM96] S. Campos, E. M. Clarke, and M. Minea. Analysis of real-time systems using symbolic techniques. In C. Heitmeyer and D. Mandrioli, editors, *Formal Methods for Real-Time Computing*. John Wiley & Sons, 1996.

[CD94] C. N. Coelho Jr. and G. De Micheli. Dynamic scheduling and synchronization synthesis of concurrent digital systems under system-level constraints. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-94)*, pages 175–181, 1994.

[CE81a] E. M. Clarke and E. A. Emerson. Characterizing properties of parallel programs as fixpoints. In *Automata, Languages, and Programming*, volume 85 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[CE81b] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[CFZ96] E. M. Clarke, M. Fujita, and X. Zhao. Multi-terminal binary decision diagrams and hybrid decision diagrams. In Sasao and Fujita [SF96], pages 93–108.

[CGP00] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[CH71] F. Commoner and A. W. Holt. Marked directed graphs. *Journal of Computer and System Sciences*, 5:511–523, 1971.

[CKL95] W.-T. Chang, A. Kalavade, and E. A. Lee. Effective heterogeneous design and co-simulation. In *Proceedings of the NATO/ASI Workshop on Hardware/Software Co-Design*, pages 187–212, 1995.

[CTGC95] M. Cornero, F. Thoen, G. Goossens, and F. Curatelli. Software synthesis for real-time information processing systems. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, pages 260–279. Kluwer Academic Publishers, 1995.

[DBG95] R. Drechsler, B. Becker, and N. Göckel. A genetic algorithm for variable ordering of OBDDs. In *Workshop Notes of the International Workshop on Logic Synthesis*, May 1995.

[DeM94] G. De Micheli. *Synthesis and Optimization of Digital Circuits*, chapter Sequential Logic Optimization, pages 441–503. McGraw-Hill, 1994.

[Den80] J. B. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, November 1980.

[DGB96] R. Drechsler, N. Göckel, and B. Becker. Learning heuristics for OBDD minimization by evolutionary algorithms. In *Parallel Problem Solving from Nature—PPSN IV*, volume 1141 of *Lecture Notes in Computer Science*, pages 730–739. Springer-Verlag, 1996.

[Dil89] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[DKMT90] G. De Micheli, D. Ku, F. Mailhot, and T. Truong. The Olympus synthesis system. In *IEEE Design and Test of Computers*, 1990.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In R. Alur, T. A. Henzinger, and E. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 208–219. Springer-Verlag, 1996.

[EBLP94] M. Engels, G. Bilsen, R. Lauwereins, and J. Peperstraete. Cyclo-static data flow: Model and implementation. In *Proceedings of the 28th Asilomar Conference on Signals, Systems, and Computers*, pages 503–507, Pacific Grove, CA, 1994.

[EHB93] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Design & Test of Computers*, pages 64–75, December 1993.

[EKP+98] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Scheduling of conditional process graphs for the synthesis of embedded systems. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE98)*, pages 132–138, 1998.

[Eme94] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, chapter 16, pages 995–1072. MIT Press, 1994.

[EMSS89] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In *Proceedings of the First Annual Workshop on Computer-Aided Verification*, 1989.

[FB99] J. Fleischmann and K. Buchenrieder. Prototyping networked embedded systems. *Computer*, 32(2):116–119, February 1999.

[FGK96] J. Frößl, J. Gerlach, and T. Kropf. An efficient algorithm for real-time model checking. In *Proceedings of the European Design and Test Conference*, pages 15–21, 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.

[GL99] P. Godefroid and D. E. Long. Symbolic protocol verification with queue BDDs. *Formal Methods in System Design*, 14:257–271, 1999.

[GLL99] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742–760, 1999.

[GSM97] T. Grötker, R. Schoenen, and H. Meyr. PCC: A modeling technique for mixed control/data flow systems. In *Proceedings of the European Design and Test Conference (ED&TC 97)*, pages 482–486, 1997.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HB98] S. Haynal and F. Brewer. Efficient encoding for exact symbolic automata-based scheduling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-98)*, 1998.

[HB99] S. Haynal and F. Brewer. A model for scheduling protocol-constrained components and environments. In *Proceedings of the 36th Design Automation Conference (DAC '99)*, 1999.

[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.

[HHW95] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: The next generation. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*, 1995.

[HL97] S. Ha and E. A. Lee. Compile-time scheduling of dynamic constructs in dataflow program graphs. *IEEE Transactions on Computers*, 46(7):768–778, July 1997.

[HMP92] T. A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In J. W. de Bakker et al., editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 226–251. Springer-Verlag, 1992.

[HN96] D. Harel and A. Naamad. The Statemate semantics of statecharts. *ACM Trans. Soft. Eng. Method*, 5(4), October 1996.

[HNSY94] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, June 1994.

[Hof95] R. D. Hof. Intel takes a bullet—and barely breaks stride. *Business Week*, pages 38–39, January 30, 1995.

[Hu95] A. J. Hu. *Techniques for Efficient Formal Verification Using Binary Decision Diagrams*. PhD thesis, University of California, Stanford, 1995.

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

[Jen90] K. Jensen. Colored Petri nets: A high level language for system design and analysis. In *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

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

[KD92] D. C. Ku and G. De Micheli. Relative scheduling under timing constraints: Algorithms for high-level synthesis of digital circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(6):696–718, June 1992.

[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.

[KMW67] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14:563–590, 1967.

[KS88] S. R. Kosaraju and G. F. Sullivan. Detecting cycles in dynamic graphs in polynomial time. In *20th Annual ACM Symposium on Theory of Computing*, pages 398–406, 1988.

[Lee88] E. A. Lee. Recurrences, iteration, and conditionals in statically scheduled block diagram languages. In R. W. Brodersen and H. S. Moscovitz, editors, *VLSI Signal Processing III*, pages 330–340. IEEE Press, New York, 1988.

[LLPY97] K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, 1997.

[LM87a] 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.

[LM87b] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[Lon93] D. E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, Carnegie-Mellon University, 1993.

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

[LS98] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.

[Mar89] F. Maraninchi. Argonaute: Graphical description, semantics, and verification of reactive systems by using a process algebra. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[MC99] A. S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In *Proceedings of the 20th International Conference on Application and Theory of Petri Nets*, 1999.

[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 (DAC '93)*, 1993.

[MP95] O. Maler and A. Pnueli. Timing analysis of asynchronous circuits using timed automata. In P. E. Camurati and H. Eveking, editors, *Correct Hardware Design and Verification Methods*, volume 987 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[MWBS88] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD-88)*, pages 6–9, 1988.

[Orl84] J. Orlin. Some problems in dynamic and periodic graphs. In W.R. Pulleyblank, editor, *Progress in Combinatorial Optimization*, pages 215–225. Academic Press, Orlando, Florida, 1984.

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

[PC98] E. Pastor and Cortadella. Efficient encoding schemes for symbolic analysis of Petri nets. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE98)*, 1998.

[PCP99] E. Pastor, J. Cortadella, and M. A. Peña. Structural methods to improve the symbolic analysis of Petri nets. In *Proceedings of the 20th International Conference on Application and Theory of Petri Nets*, 1999.

[PMRM94] M. Pankert, O. Mauss, S. Ritz, and H. Meyr. Dynamic data flow and control flow in high level DSP code synthesis. In *Proceedings of the 1994 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP-94)*, volume 2, pages 449–452, 1994.

[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.

[QS81] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.

[RB94] I. Radivojević and F. Brewer. Ensemble representation and techniques for exact control-dependent scheduling. In *Proceedings of the 7th International Symposium on High-Level Synthesis*, pages 60–65, 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.

[RK88] S. K. Rao and T. Kailath. Regular iterative algorithms and their implementations on processor arrays. *Proceedings of the IEEE*, 6:259–282, March 1988.

[RK97a] J. Ruf and T. Kropf. Symbolic model checking for a discrete clocked temporal logic with intervals. In H. F. Li and D. K. Probst, editors, *Advances in Hardware Design and Verification*, pages 146–163. Chapman & Hall, 1997.

[RK97b] J. Ruf and T. Kropf. A new algorithm for discrete timed symbolic model checking. In O. Maler, editor, *Hybrid and Real-Time Systems*, volume 1201 of *Lecture Notes in Computer Science*, pages 18–32. Springer-Verlag, 1997.

[RK98] J. Ruf and T. Kropf. Using MTBDDs for composition and model checking of real-time systems. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design*, volume 1522 of *Lecture Notes in Computer Science*, pages 185–202. Springer-Verlag, 1998.

[RS97] J.-F. Raskin and P.-Y. Schobbens. State clock logic: A decidable real-time logic. In O. Maler, editor, *Hybrid and Real-Time Systems*, volume 1201 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[Rud93] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD-93)*, pages 42–47, 1993.

[RZE+99] K. Richter, D. Ziegenbein, R. Ernst, L. Thiele, and J. Teich. Representation of function variants for embedded system optimization and synthesis. In *Proceedings of the 36th Design Automation Conference (DAC '99)*, pages 517–522, 1999.

[SF96] T. Sasao and M. Fujita, editors. *Representations of Discrete Functions*. Kluwer Academic Publishers, 1996.

[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.

[SLWS98] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Quasi-static scheduling of embedded software using free-choice Petri nets. In *Proceedings of the Workshop on Hardware Design and Petri Nets (HPWN '98)*, 1998.

[SLWS99] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Quasi-static scheduling of embedded software using equal conflict nets. In *Proceedings of the 20th International Conference on Application and Theory of Petri Nets (ATPN '99)*, 1999.

[SMTS99] K. Strehl, C. Moraga, K.-H. Temme, and R. Stanković. Fuzzy decision diagrams for the representation, analysis, and optimization of rule bases. Technical Report TIK-77, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, September 1999.

[SMTS00] K. Strehl, C. Moraga, K.-H. Temme, and R. Stanković. Fuzzy decision diagrams for the representation, analysis, and optimization of rule bases. In *Proceedings of the 30th International Symposium on Multiple-Valued Logic (ISMVL 2000) (to appear)*, Portland, Oregon, May 23–25, 2000.

[SRMB98] S. Schulz, J. W. Rozenblit, M. Mrva, and K. Buchenrieder. Model-based codesign. *Computer*, 31(8):60–67, August 1998.

[SSR89] R. Saracco, J. R. W. Smith, and R. Reed. *Telecommunications systems engineering using SDL*. North-Holland, Elsevier, 1989.

[ST98] K. Strehl and L. Thiele. Symbolic model checking of process networks using interval diagram techniques. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-98)*, pages 686–692, San Jose, California, November 8–12, 1998.

[ST99a] K. Strehl and L. Thiele. Interval diagram techniques for symbolic model checking of Petri nets. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE99)*, pages 756–757, Munich, Germany, March 9–12, 1999.

[ST99b] K. Strehl and L. Thiele. Interval diagram techniques and their applications. In *Proceedings of the 8th International Workshop on Post-Binary ULSI Systems*, pages 23–24, Freiburg im Breisgau, Germany, May 19, 1999. Invited paper.

[Str99] K. Strehl. Interval diagrams: Increasing efficiency of symbolic real-time verification. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA '99)*, pages 488–491, Hong Kong, December 13–15, 1999.

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

[STZE99] K. Strehl, L. Thiele, D. Ziegenbein, and R. Ernst. Scheduling hardware/software systems using symbolic techniques. Technical Report TIK-67, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, January 1999.

[TKY+98] S. Taşıran, S. P. Khatri, S. Yovine, R. K. Brayton, and A. Sangiovanni-Vincentelli. A timed automaton-based method for accurate computation of circuit delay in the presence of cross-talk. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design*, volume 1522 of *Lecture Notes in Computer Science*, pages 149–166. Springer-Verlag, 1998.

[TSZ+99] L. Thiele, K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich. *FunState*—an internal design representation for codesign. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-99)*, pages 558–565, San Jose, California, November 7–11, 1999.

[TTN+98] L. Thiele, J. Teich, M. Naedele, K. Strehl, and D. Ziegenbein. *FunState*—functions driven by state machines. Technical Report TIK-33, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, January 1998.

[TTS00] L. Thiele, J. Teich, and K. Strehl. Regular state machines. *Journal of Parallel Algorithms and Applications*, Special Issue on Advanced Regular Array Design *(to appear)*, 2000.

[TZB98] J. Teich, E. Zitzler, and S. S. Bhattacharyya. Buffer memory optimization in DSP applications—an evolutionary approach. In *Parallel Problem Solving from Nature—PPSN V*, volume 1498 of *Lecture Notes in Computer Science*, pages 885–894. Springer-Verlag, 1998.

[vdB94] M. von der Beeck. A comparison of statecharts variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. Springer-Verlag, 1994.

[VNG95] F. Vahid, S. Narayan, and D. D. Gajski. SpecCharts: A VHDL frontend for embedded systems. *IEEE Transactions* *on Computer-Aided Design of Integrated Circuits and Systems*, 14(6):694–706, 1995.

[WG82] I. Watson and J. Gurd. A practical data flow computer. *Computer*, 15(2):51–57, February 1982.

[Yam96] S. Yamane. The verification technique of real-time systems using probabilities. In *Proceedings of the 3rd International Workshop on Real-Time Computing Systems and Applications (RTCSA '96)*, 1996.

[YHTM96] T. Yoneda, H. Hatori, A. Takahara, and S. Minato. BDDs vs. zero-suppressed BDDs: For CTL symbolic model checking of Petri nets. In *Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 435–449. Springer-Verlag, 1996.

[ZER+98] D. Ziegenbein, R. Ernst, K. Richter, J. Teich, and L. Thiele. Combining multiple models of computation for scheduling and allocation. In *Proceedings of the 6th International Workshop on Hardware/Software Codesign (Codes/CASHE '98)*, pages 9–13, 1998.

[Zha95] H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. *Proceedings of the IEEE*, 83(10):1374–1396, October 1995.

[ZRE+98] D. Ziegenbein, K. Richter, R. Ernst, J. Teich, and L. Thiele. Representation of process mode correlation for scheduling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-98)*, pages 54–61, 1998.

Acronyms

ATM	asynchronous transfer mode
BDD	binary decision diagram
CDD	clock difference diagram
CTL	computation tree logic
DBM	difference bounds matrix
DD	decision diagram
FIFO	first-in, first-out
FSM	finite state machine
FuDD	fuzzy decision diagram
GCI	greatest common interval
IDD	interval decision diagram
ILP	integer linear programming
IMD	interval mapping diagram
MDD	multi-valued decision diagram
MTBDD	multi-terminal binary decision diagram
NDD	numerical decision diagram
PAD	predicate action diagram
RSM	regular state machine
SDF	synchronous dataflow
SDL	specification and description language
SMV	*common symbolic model checking tool*
SPI	system property intervals

Symbols

A	for all paths
$\mathit{BackIm}(\mathbf{S}, \mathbf{T})$	back-image operation
$\mathbf{C}$	set of embedded components
$C.e$	invocation of event e in embedded component C
δ	transition relation
Δx	state distance
$\mathbf{E}$	edge set
E	there exists a path
$\mathbf{F}$	set of functions
F	eventually
f_{+}	shift function
$f_{=}$	assign function
$f_{x_i^{\mathbf{I}}}$	cofactor of f with respect to literal $x_i^{\mathbf{I}}$
G	always
$\mathbf{G}_{qq}$	staying condition of q
$\mathbf{G}_{qq'}$	transition guard from q to q'
$\mathbf{I}$	interval *or* set of input ports
$\mathbb{I}$	set of all intervals
$\mathbf{I}(\mathbf{P}_i)$	interval cover with respect to $\mathbf{P}_i$
$\mathbf{I}_A$	action interval
$\mathbf{I}_P$	predicate interval
$\mathit{Im}(\mathbf{S}, \mathbf{T})$	image operation

ITE	If-Then-Else operator
M	finite state machine
M in s	in-state operator
N	dataflow network
$\mathbf{O}$	set of output ports
Φ	time forward projection
p/a	transition label with predicate p and action a
$\mathbf{P}_i$	domain set of variable with index i
$PreIm(\mathbf{S}, \mathbf{T})$	inverse image operation
$\mathbf{Q}$	state space
$q\#$	number of tokens in queue q
$q\$k$	value of kth token in storage unit q
$\mathbf{Q}_f$	range set of function f
$R_{qq'}$	reset function from q to q'
$\mathbf{S}$	state set *or* set of storage units
$s(x)$	characteristic function of state set
τ	function latency
$\mathbf{T}$	transition relation
$t(x, x')$	characteristic function of transition relation
TN_0, TN_1	0-, 1-terminal node
U	until
$\mathbf{V}$	node set
X	next time
$x_1 \xrightarrow{t} x_2$	state transition t from x_1 to x_2
$x_i^{\mathbf{I}}$	literal of x_i with respect to $\mathbf{I}$
$\mathbf{Z}$	clock space