Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl für Regelungstechnik
Prof. Dr.-Ing. G. Roppenecker   Prof. Dr.-Ing. Th. Moor

Methods of Supervisory Control: A Software Implementation

Als Master Thesis
vorgelegt von

Bernd Opitz

Betreuer:
Dr.-Ing. Klaus Schmidt

Betreuer:
Prof. Dr.-Ing. Th. Moor

Ausgabedatum: 20.06.2005
Abgabedatum: 20.12.2005

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 20.12.05

(Bernd Opitz)

Contents

Zusammenfassung

Gegenstand dieser Arbeit ist die Entwicklung einer Softwarebibliothek für ereignisdiskrete Systeme (DES). Ereignisdiskrete Systeme besitzen einen diskreten Zustandsraum in dem Übergänge ereignisgesteuert stattfinden. Für diese Klasse von Systemen wurde Mitte der 80er Jahre von P.J. RAMADGE und W.M. WONHAM eine Regelungstheorie entwickelt, die sog. RW Supervisory Control Theory (SCT). Mit dieser können ereignisdiskrete Systeme als endliche Automaten modelliert werden. Dabei werden interagierende Teile eines Systems als einzelne Automaten modeliert und zu einem größeren Gesamtautomaten zusammengefügt, für den mittels einer vorgegebenen Spezifikation eine Steuerung berechnet werden kann.

Bei der Modellierung großer Systeme mit mehreren Nebenläufigkeiten ergibt sich hierbei das Problem, dass die Größenordnung des Zustandsraumes nicht mehr von Rechnersystemen erfasst werden kann und damit die Berechnung einer Steuerung unmöglich wird. Zur Lösung des Problems der sogenannten Zustandsraumexplosion existieren in der Literatur verschiedene Ansätze, von denen einer von KLAUS SCHMIDT am Lehrstuhl für Regelungstechnik der Universität Erlangen-Nürnberg entwickelt wurde [Sch05b].

In dieser Arbeit wurde ein Software-Modell eines Automaten zur Modellierung und Steuerung von ereignisdiskreten Systemen erstellt. Dabei wurden zunächst zwei abstrakte Automatenmodelle auf ihre Eignung für Automatenalgorithmen untersucht. Für das universeller einsetzbare Datenmodell wurde eine Spezifikation seiner Datenstruktur durchgeführt, die als Klasse `Generator` mit Hilfe abstrakter Datentypen (ADT) implementiert wurde. Nach der Umsetzung von Algorithmen für Automaten und reguläre Sprachen für die `Generator` Klasse wurde eine für die SCT spezielle Er-

weiterung um steuerbare Ereignisse diskutiert und als erweiterte Klasse `cGenerator` implementiert. Basierend auf der `cGenerator` Klasse wurde der Algorithmus zur Berechnung einer Steuerung für die SCT umgesetzt. Weiterhin wurden alle erforderlichen Algorithmen zur Synthese von hierarchischen Steuerungen für dezentrale ereignisdiskrete Systeme gemäß [Sch05b] implementiert.

Der praktische Teil der Arbeit bestand dabei in einer Implementierung des Automaten Software-Modells mittels der Programmiersprache C++ und der darauf basierenden Umsetzung der Algorithmen. Ergänzend wurde eine textbasierte Applikation entwickelt, mit der die implementierten Algorithmen getestet werden können. Das Ergebnis der praktischen Arbeit, die C++ Klassenbibliothek LIBFAUDES, wird unter der GNU Lesser General Public License (LGPL) im World Wide Web zur Verfügung gestellt.

Chapter 1

Introduction

The Control theory of discrete event system is an action of research. This thesis covers the framework provided by P.J. RAMADGE and W.M. WONHAM in [RW89] with the extensions to hierarchical and decentralized control provided by KLAUS SCHMIDT in his Phd Thesis [Sch05b].

In the late 80th RAMADGE and WONHAM introduced a framework regular languages and finite automata for modelling and controling discrete event systems (DES). Besides Petrinets this is the most important approach to the supervisory control of DES. A DES is modeled as a finite automaton. a given regular language specification, a controller can be computed such that the closed-loop system fulfills the specification. This concept of monolithic supervisor computation is denoted as "RW Supervisory Control Theory" (SCT).

An ongoing research topic is the computation of supervisors for real world large-scale systems like manufacturing systems. These have many concurrent activities whose combinations all have to be modeled as single states in the RW supervisory control theory. This leads to a huge number of states, the so called "state space explosion", which cannot be handled by today's computer technology. Many concepts have been introduced for solving the state space explosion problem. The common feature of these approaches is the use of vertical or horizontal system structure in hierarchical or decentralized approaches. Using this system structure, the computation of the overall system is avoided. [Sch05b] combines both ideas in an approach for the hierarchical control of decentralized DES. Concurrent activies in decentralized system models are abstracted to their shared behavior on a higher level. This results in a multi level control hierarchy with a single supervisor on the highest level, controlling the abstracted behavior of the overall system. This highest-level supervisor can be implemented efficiently using decentralized supervisors in the lower levels.

This thesis deals with the aspects of implementing a generally applicable software library for supervisory control of DES that can be easily extended to different approaches. The RW supervisory control theory is used as the base where other frameworks can be put on. As an extension, the library provides all algorithms required for [Sch05b]. Data models and algorithms are treated language independent for general object-orientated programming. The practical work consists of developing such a library with the object orientated programming language C++. Knowledge of the RW theory which is completely described in [Won04], is assumed. Basic data types used in the thesis are described in standard computer science literature like [AHU+83].

At present, there already exist some software packages for supervisory control of discrete event systems. Well known are TCT [TCT], the UMDES SOFTWARE LIBRARY [UMD] and SUPREMICA [Sup]. TCT provides a console application for the monolithic supervisor synthesis. UMDES SOFTWARE LIBRARY by the University of Michigan is comparable with the library developed as practical work in this thesis. Both libraries do not exactly cover the same theoretical frameworks but implement RW supervisory control theory as a common base.

DESUMA, a java based graphical user interface for [UMD] by the University of Michigan and Mount Allison University, and SUPREMICA by Chalmers University of Technology both provide a feature-rich graphical environment for basic supervisory control theory. There may be other less known or publicly unavailable software packages which deal with the computation of supervisors for DES. By now none of the these packages implement a theoretical framework preventing the state space explosion problem. Also there is no public available source code for any of the packages.

This work is an effort to provide an open source software library with a free license[1] for

[1] The practical result of this thesis, called the faudes library is publicly available at [fau] under the

supervisory control of DES. The library has been developed with both high performance and extensibility in mind. The public release of the library sources shall encourage external researchers to use the library in their own projects or even contribute code for further development of the library. The library provides a automaton base class for automaton and language operations, an extened automaton model for RW supervisory control theory and as an extension the algorithms for [Sch05b].

The outline of this work is as follows: After a short introduction to the RW supervisory control theory in Chapter 2, a software model for finite automata is developed in Chapter 3. the automata model, basic algorithms for regular languages and finite automata are discussed in Chapter 4. Chapter 5 describes the extension of the automata model to fit the requirements of supervisory control theory, followed by the algorithm for computing supervisors. Chapter 6 deals with efficient implementations of the algorithms for the theoretical framework given in [Sch05b]. However, the development of a data structure for the hierarchical decentralized architecture is beyond the scope of this work and is covered in [Sch05a]. Finally the thesis concludes with some thoughts about further development and usage of the created software library.

LGPL [LGP] license.

Chapter 2

Basics of Supervisory Control of Discrete Event Systems

This chapter summarizes the most important results of the supervisory control theory (SCT) of discrete event systems and thus provides a basis for the algorithms implemented in the following chapters. A thorough description of the standard framework for SCT introduced by P.J. RAMADGE and W.M. WONHAM (RW) is given in [Won04]. A less formal introduction to SCT can be found in [CL99].

2.1 Regular Languages and Finite Automata

In RW supervisory control theory, the behavior of DES is modeled by finite automata and regular languages. For the further discussion some definitions are required. At first the terms alphabet, string and formal language are defined. An alphabet is a set of symbols, a string is a sequence of symbols and a language is a set of strings. Then deterministic and nondeterministic finite automata are introduced, followed by the terms generated language, marked language, blocking and regular language.

An *alphabet*, usually denoted Σ is a finite set of distinct symbols $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$. A *string* is a sequence $\sigma_{i1}\sigma_{i2}\ldots\sigma_{ik}$, $\sigma_{ij} \in \Sigma$, $k \in \mathbb{N}$ of symbols.

Σ^+ denotes the set of finite symbol sequences of the form $\sigma_{i1}\sigma_{i2}...\sigma_{ik}$ where $\sigma_{ij} \in \Sigma$ for $i = 1, \ldots, k$. The *empty string* ϵ denotes the empty sequence containing no symbols, where $\epsilon \notin \Sigma$. $\Sigma^* := \{\epsilon\} \cup \Sigma^+$ denotes the so called *Kleene-Closure* of the alphabet Σ.

The concatenation of $s \in \Sigma^*$ and $t \in \Sigma^*$ is written $st \in \Sigma^*$. The string s is denoted a *prefix*, t is denoted a *suffix* of st.

Definition 2.1.1 (Formal Language [HU79]). A *formal language* L over an alphabet Σ, also called *language* over Σ, is any subset $L \subseteq \Sigma^*$.

The definition includes both the *empty language* $\varnothing$ and Σ^* itself. There is a difference between the empty language $\varnothing$ and the string with no symbols ϵ. The *prefix-closure* $\overline{L} := \{s \in \Sigma^* \mid \exists t \in \Sigma^* \text{ s. t. } (st \in L)\}$ consists of all the prefixes of all strings in L, $L \subseteq \overline{L}$. The *active event set* after the string s is defined as $\Sigma(s) := \{\sigma \mid s\sigma \in L\}$.

Finite automata are introduced as a modeling framework to represent and manipulate languages. While most of the RW control theory is described in the language framework, finite automata are used to model the logical behavior of DES in practice.

Definition 2.1.2 (Automaton [HU79]). An automaton is a 5-Tuple, $G := (X, \Sigma, \delta, X_0, X_\mathrm{m})$ consisting of

- X: the set of states,
- Σ: the set of events,
- δ: the transition function is a partial function $\delta : X \times \Sigma \to 2^X$ only defined on a subset of Σ in any state $x \in X$,
- X_0: the set of initial states $X_0 \subseteq X$,
- X_m: the set of marked states $X_\mathrm{m} \subseteq X$.

An automaton is called *finite*, if the set of states is finite. If the set of initial states X_0 consists of a single state x_0 and the transition function is unique, $\delta : X \times \Sigma \to X$, the automaton is called *deterministic automaton*. Otherwise it is called[1] *nondeterministic automaton*. A nondeterministic automaton can always be transformed into a deterministic

[1]Note that the concept of ϵ-transitions in nondeterministic automata as found in [CL99] is not introduced here as states connected by ϵ-transitions can always be modeled as one single state.

one, if the set of states is finite. The corresponding algorithm is discussed in Chapter 4.

For convenience, δ is extended to a partial function on $X \times \Sigma^*$ by the recursive definition

- $\delta(x, \epsilon) := x$
- $\delta(x, s\sigma) := \delta(\delta(x, s), \sigma)$ for $s \in \Sigma^*$ and $\sigma \in \Sigma$

The *active event set function* $\Lambda := X \to 2^\Sigma$ with $\Lambda(x) := \{\sigma \mid \delta(x, \sigma)!\}$ maps every state to the respective set of events executable in this state. $\Lambda(x)$ is called the *active event set* at state x where $\delta(x_0, s)!$ says that the transition is defined.

An example of a finite deterministic automaton with five states that operatates on the alphabet $\Sigma = \{a, b\}$ is shown in figure 2.1.

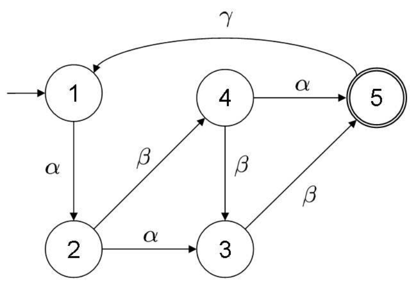

Figure 2.1: Example of a finite automaton

The generated language of an automaton G, $L(G)$ contains all strings in G starting in an initial state. The marked language $L_{\mathrm{m}}(G)$ contains all strings in G starting in an initial state and ending in a marked state.

Definition 2.1.3 (Generated and Marked Language [Won04]). For a given automaton $G := (X, \Sigma, \delta, X_0, X_{\mathrm{m}})$ the *generated language* is defined as $L(G) := \{s \in \Sigma^* \mid \delta(x_0, s)!\}$ and the *marked language* is defined as $L_{\mathrm{m}}(G) := \{s \in \Sigma^* \mid \delta(x_0, s) \in X_{\mathrm{m}}\}$.

There is no unique way to construct an automaton that marks a given language. However, an automaton that marks a language with a minimum set of states is called a *canonical recognizer* which is unique except for an isomorphism [HU79].

Definition 2.1.4 (Blocking and Nonblocking [CL99]). An automaton is called *blocking* if $\overline{L_{\mathrm{m}}(G)} \subset L(G)$ and *nonblocking* if $\overline{L_{\mathrm{m}}(G)} = L(G)$.

Blocking means there exists at least one string in the generated language of an automaton which cannot be extended to reach a marked state. In nonblocking automata from every string of the generated language there is a path to a marked state.

A state is *accessible* if it can be reached by a transition path from an initial state. In contrast, a state is *coaccessible* if there is a transition path from the state to a marked state.

Any language can be marked by a automaton, but only finite automata can be stored in the memory of a computer. As there are languages which cannot be marked by a finite automaton, e.g. $L = \{a^n b^n \mid n \geq 0\}$, the languages that can be used to represent DES are restricted to the class of *regular languages*. It should be noted that some languages that cannot be marked by finite automaton can be respresented by petri nets with a finite transition structure.

Definition 2.1.5 (Regular Language[CL99])**.** A formal language is denoted *regular* if it can be marked by a finite automaton. The distinct class of languages that can be marked by finite automata is called the class of *regular languages*.

This is an important result as it means the behavior of DES modeled by finite automata can always be described with regular languages.

Note that both deterministic finite automata and nondeterministic finite automata are represented by the same class of languages as a finite nondeterministic automata can always be converted into a finite deterministic automata and every deterministic automaton is also a nondeterministic automaton.

2.2 Supervisory Control Theory

The principle of supervisory control is restricting the behavior of a discrete event system to a given specification. Let $G := (X, \Sigma, \delta, X_0, X_\mathrm{m})$ be an automaton that models the uncontrolled behavior of a discrete event system, namely the *Plant*. Let S be a supervisor for G. S controls G in the closed feedback loop shown in figure 2.2 by disabling events in the current state of G which would make the controlled system violate the specification.

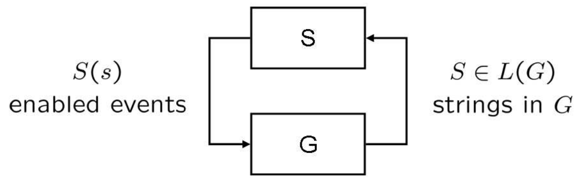

Figure 2.2: The supervisor S controls the plant G in a feedback loop.

As there may be events which cannot be directly influenced by a supervisor, the set of events is divided into two disjoint subsets: $\Sigma = \Sigma_{\mathrm{c}} \cup \Sigma_{\mathrm{uc}}$

Σ_{c} contains the subset of controllable events whose execution can directly be prevented by a supervisor, while Σ_{uc} yields the subset of uncontrollable events which cannot be prevented. Reasons for modeling events as uncontrollable are process control limitations, unpredictable machine breakdowns or sensor readings which are not the direct result of a given command, just to name some examples.

Definition 2.2.1 (Supervisor [WR87])**.** Formally a supervisor is a function mapping the language generated by G to Γ

$$S : L(G) \to \Gamma,$$

where $\Gamma = \left\{\gamma \in 2^{\Sigma} \mid \gamma \supseteq \Sigma_{\mathrm{uc}}\right\}$. Γ is called the set of all *control patterns.* A *control pattern* represents the set of events enabled by the supervisor and then contains all uncontrollable events.

A supervisor follows strings $s \in L(G)$ and restricts the active event set after the string s to $S(s) \cap \Sigma(s)$ which is called the set of *enabled events*.

The closed-loop system of the plant G and the supervisor S is written S/G meaning 'G under supervision of S'. S/G is a DES whose generated and marked language are defined as follows.

Definition 2.2.2 (Languages generated and marked by S/G [CL99])**.** The *language generated* by S/G is recursively defined by:

1. $\epsilon \in L(S/G)$
2. $[(s \in L(S/G))$ and $(s\sigma \in L(G))$ and $(\sigma \in S(s))] \Leftrightarrow [s\sigma \in L(S/G)]$

The *language marked* by S/G is defined by

$$L_{\mathrm{m}}(S/G) := L(S/G) \cap L_{\mathrm{m}}(G).$$

This means every sequence of symbols s that is already executed in the closed-loop system L/G can be extended by an event σ if and only if $s\sigma \in L(G)$ and σ is contained in the control pattern at string s. The language marked by the closed-loop system is then defined by language intersection of the closed-loop language with the marked language of G.

A specification language K can be implemented by a supervisor if it is controllable w.r.t. $L(G)$.

Definition 2.2.3 (Controllability [WR87]). A language $K \subseteq L(G)$ is *controllable* (*with respect to* G) if and only if

$$\overline{K}\Sigma_{\mathrm{uc}} \cap L(G) \subseteq \overline{K}.$$

This means, any prefix of K which is also in $L(G)$ followed by an uncontrollable event must still be a prefix of K.

Definition 2.2.4 (Set of Controllable Sublanguages [WR87]). The *set of controllable sublanguages* of $L(G)$, denoted $(L(G))$ is defined as

$$\mathcal{C}(L(G)) := \{H \subseteq L(G) \mid \overline{H}\Sigma_{\mathrm{uc}} \cap L(G) \subseteq \overline{H}\}.$$

Controllability is closed under union. As a consequence, if K_1 and K_2 are controllable, then $K_1 \cup K_2$ is controllable, too. Therefore, the union of the set of controllable sublanguages is also controllable. Then there must be a "largest" element in the set of controllable sublanguages that includes all other controllable sublanguages. This is defined as the *supremal controllable sublanguage*.

Definition 2.2.5 (Supremal Controllable Sublanguage [WR87]). Let $E \subseteq L(G)$ be a specification language. The *supremal controllable sublanguage* of E with respect to $L(G)$ is

$$\kappa_{L(G)}(E) := \bigcup \{K \in \mathcal{C}(L(G)) \mid K \subseteq E\}$$

$\kappa_{L(G)}$ is the union of all controllable sublanguages that agree with the specification E. A recognizer of $\kappa_{L(G)}(E)$ is a instance of a minimally restrictive supervisor for the DES G with the specification E.

Nonblocking control requires the definition of a further property:

Definition 2.2.6 ($L_m(G)$-Closure [CL99]). A language K is $L_{\mathrm{m}}(G)$-closed if

$$K = \overline{K} \cap L_{\mathrm{m}}(G).$$

This means that every string in K is prefix of a string in $L_{\mathrm{m}}(G)$.

Theorem 2.2.1 (Nonblocking Controllability Theorem [WR87]). Let $G = (X, \Sigma, \delta, x_0, X_{\mathrm{m}})$ be a DES with $\Sigma_{\mathrm{uc}} \subseteq \Sigma$ as the set of uncontrollable events. Let $K \subseteq L_{\mathrm{m}}(G)$ be a sublanguage with $K \neq \emptyset$. A *nonblocking supervisor* S for G with $L_{\mathrm{m}}(S/G) = K$ and $L(S/G) = \overline{K}$ exists iff

1. K is controllable (with respect to $L(G)$)

2. K is $L_{\mathrm{m}}(G)$-closed

This finally leads to the basic problem of supervisory control of DES [CL99].

Let $G = (X, \Sigma, \delta, x_0, X_{\mathrm{m}})$ be a discrete event system with the events Σ and the uncontrollable events $\Sigma_{\mathrm{uc}} \subseteq \Sigma$. Let $L_{\mathrm{am}} \subseteq L_{\mathrm{m}}(G)$ be the admissible marked language of G, which is assumed to be $L_{\mathrm{m}}(G)$-closed. A *nonblocking* supervisor S has to be found such that:

1. $L_{\mathrm{m}}(S/G) \subseteq L_{\mathrm{am}}$

2. $L_{\mathrm{m}}(S/G)$ is "as large as possible"

For solving this problem, it is necessary to compute a nonblocking supervisor S that is minimally restrictive. This is achieved by choosing S such that

$$L(S/G) = \overline{\kappa_{L(G)}(L_{\mathrm{am}})} \text{ and } L_{\mathrm{m}}(S/G) = \kappa_{L(G)}(L_{\mathrm{am}})$$

as long as $\kappa_{L(G)}(L_{\mathrm{am}}) \neq \emptyset$ which is the basic concept of the SCT.

Chapter 3

Automaton Data Model

As described in Chapter 2, the supervisory control theory (SCT) uses finite automata to model the behavior of discrete event systems. While small didactic examples with only few states can be handled without computational support, this is not possible for most real world DES. Especially manufacturing systems with concurrent activities introduce complexity far beyond the scope of being modeled and processed by hand. Here software is required to handle the computation. This chapter deals with the development of an appropriate data model for finite automata, that fits well for the methods used in the supervisory control theory. In Section 3.1, two basic automata data models are introduced. These models are evaluated with commonly used SCT methods in Section 3.2.

3.1 Introduction of Abstract Automata Models

The effectiveness of a software algorithm heavily depends on the underlying data structure. At the same time the design of a data structure depends on the algorithms operating on it. Therefore, it is necessary to identify typical data access patterns in supervisory control theory methods, to figure out a proper data model for the finite automaton defined in Chapter 2. Here the problem arises that the ”optimal” data structure can only be known after evaluating all supervisory control methods at first which is beyond the scope of this thesis. To get a good overview what kind of data model is appropriate, two different abstract automaton data models which represent the two main modeling concepts for finite automata are introduced. The corresponding data access patterns are investigated in the next section.

3.1.1 Linked List Model

At first, an approach which is derived from the directed graph representation of an automaton as e.g. shown in Figure 2.1 is analyzed.The automaton is considered as a set of states containing transitions that point to other states. The initial states and marked states are used as starting points. In addition every state has a flag determining if the state is marked. This results in the following abstract data types:

Automaton:

- Initial States: List of pointers to states
- Marked States: List of pointers to states
- Set of Events

State:

- State Identifier
- Marked State (Binary)
- Transitions: Set of transitions

Transition:

- Event
- Pointer to State

These data objects build a linked list-like model of an automaton. The automaton is accessed by its initial states or marked states which are represented by some data type that points to state objects. Every state object has an identifier representing the name or number of the state, a binary flag containing the marking status and a list of transition objects. Every transition object has an event and a pointer to a state associated with it. In addition, the set of events is stored, as the automaton may contain events not associated with any transition. This forms a structure as shown in Figure 3.1. The automaton consists of states pointing to each other via transition objects. It is assumed that the pointers are bidirectional since some SCT methods may require traversing an automaton backwards beginning at the marked states. States between initial and marked states are only accessible by traversing the linked pointers. Therefore, random access of states is impossible. The representation is similar to a double linked list that provides direct access to both ends while random access in the middle of the list is impossible. For this reason the stated automata data model is called a *linked list automaton data model*.

Figure 3.1 shows how states are linked to each other. As can be seen, the diagram resembles the directed graph of an automaton.

3.1.2 Set Based Model

The second data model is directly deduced from the automaton given in Definition 2.1.2. The automaton is modeled as a five tuple consisting of a state set, an event set, a set of transitions, a set of initial states and a set of marked states:

Automaton:

- Set of States
- Set of Events
- Set of Transitions
- Set of Initial States
- Set of Marked States

State / Initial State / Marked State: State Identifier

Event: Event Identifier

Transition: State - Event - State

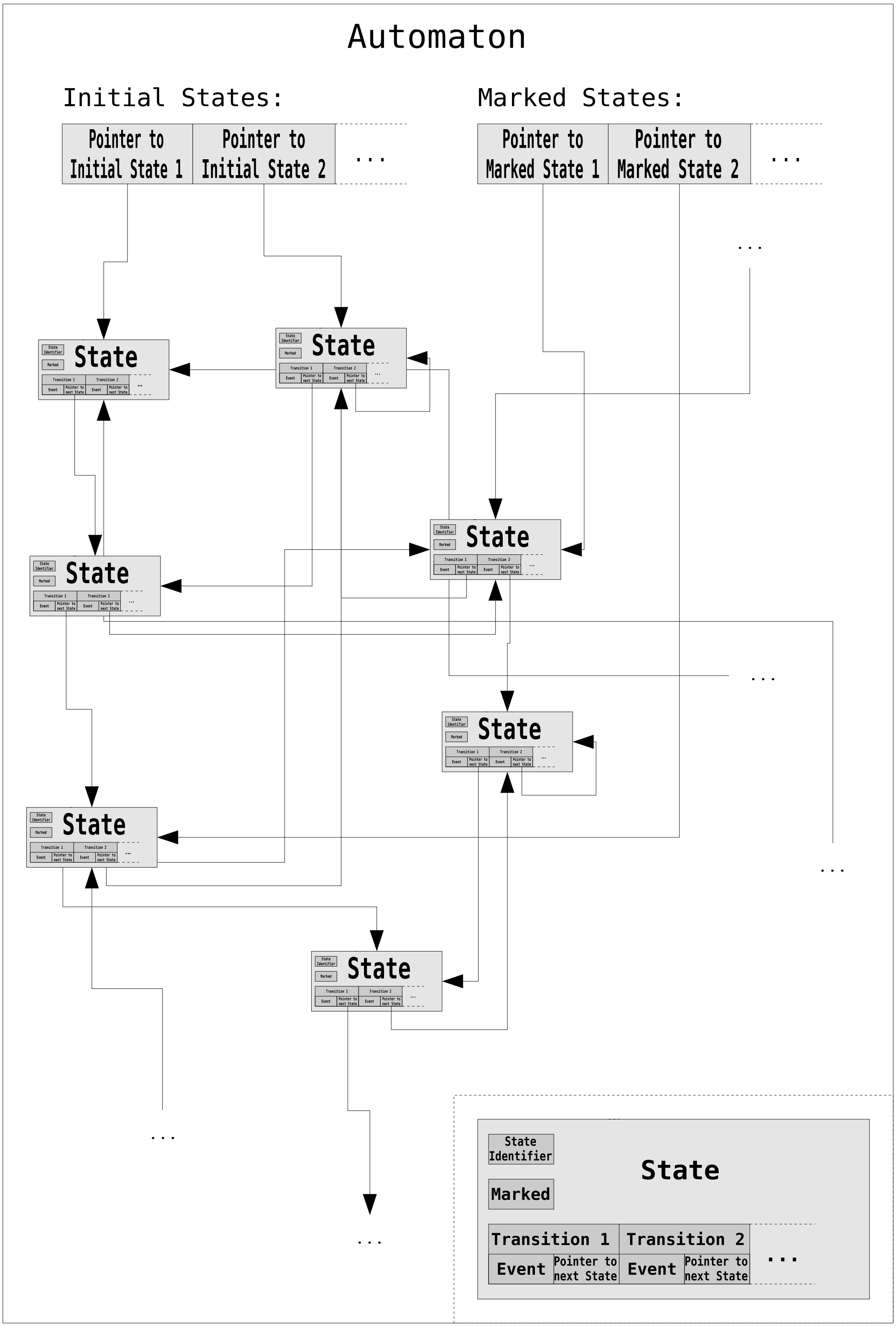

Figure 3.1: Linked states in the linked list automaton data model

In the context of the abstract set automata model the term *set* shall be specified as an amount of states, events or transitions whereas no specification is given for the sorting order. The set allows to determine if a specific state, event or transition is contained and allows retrieving all included elements in an unspecified order. Like in the algebraic set definition it may not contain duplicate elements in contrast to a so called *multiset*. Figure 3.2 shows the example automaton of Figure 2.1 in the context of a set based model.

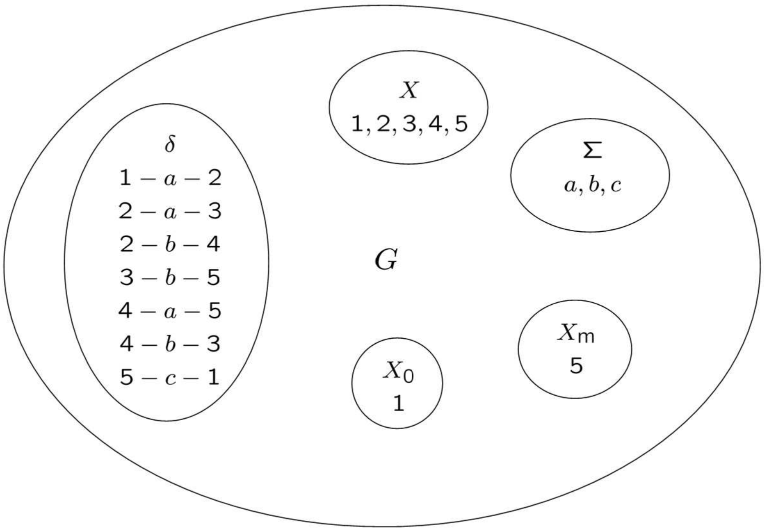

Figure 3.2: A set based model holds the example automaton of Figure 2.1.

3.2 Evaluation of Data Models

In this section, the abstract automaton data models introduced in the previous section are evaluated with methods used in DES modeling. The following three representative methods are investigated:

Parallel Composition: The main application of parallel composition is merging interacting components of a plant. In the RW theory all parts of a DES have to be combined to one monolithic automaton by parallel composition. This may lead to the state space explosion problem stated in Chapter 1. The method is similar to computation of the supremal controllable sublanguage used for supervisor computation. Both follow the transitions of two given automata in parallel beginning at their respective initial states while the pairs of parallel states build a new automaton.

Language Projection: The language projection is used in different decentralized [YL00, dQC00] and hierarchical [Sch05b, Led02, CC02] approaches.

Subset Construction: Conversion of a nondeterministic to a deterministic automaton is done by subset construction. Such a conversion is often required after executing other algorithms that most likely result in a nondeterministic automaton, e.g. language projection. Subset construction, along with state space minimization, is an algorithm that builds a new automaton by constructing power sets of the existing stateset. The order of memory complexitiy in subset construction is exponential in the number of states.

For each method the definition and an abstract algorithm is stated. A summary of general data access patterns and a short discussion of possible problems with each of the two introduced data models follows.

The tests of the automaton models with these methods show their general qualification for computation in SCT. While this is the most important criterion for choosing an automaton model, the raw performance of a model is important too. Therefore a performance evaluation of the models, data access patterns that typically occur in the iteration loops of SCT algorithms is included following up the discussion of the methods.

At first the automaton models are evaluated with the three stated algorithms.

3.2.1 Parallel Composition Method

Definition 3.2.1 (Parallel Composition)**.** The *parallel composition* of two finite automata $G_1 = (X_1, \Sigma_1, \delta_1, X_{0,1}, X_{\mathrm{m},1})$ and $G_2 = (X_2, \Sigma_{2,}\delta_2, X_{0,2}, X_{\mathrm{m},2})$ is defined as the following automaton:

$$G_1 \parallel G_2 := \left(X_{1\parallel 2}, \Sigma_1 \cup \Sigma_2, \delta_{1\parallel 2}, X_{0,1} \times X_{0,2}, X_{\mathrm{m},1\parallel 2}\right)$$

where

- $X_{1\|2} := \{(x_1 \in X_1, x_2 \in X_2) \mid \exists (x_{0,1}, x_{0,2}) \in X_{0,1} \times X_{0,2}, \exists s \in (\Sigma_1 \cup \Sigma_2)^*$ such that $\delta_{1\|2}((x_{0,1}, x_{0,2}), s) = (x_1, x_2)\}$

- $\delta_{1\|2}((x_{0,1}, x_{0,2}), \sigma) := \begin{cases} (\delta_1(x_1, \sigma), \delta_2(x_2, \sigma)) & \text{if } \sigma \in \Lambda_1(x_1) \cap \Lambda(x_2) \\ (\delta_1(x_1, \sigma), x_2) & \text{if } \sigma \in \Lambda_1(x_1) \setminus \Sigma_2 \\ (x_1, \delta_2(x_2, \sigma)) & \text{if } \sigma \in \Lambda_2(x_2) \setminus \Sigma_1 \\ \text{undefined} & \text{otherwise} \end{cases}$

- $X_{\mathrm{m},1\|2} := \{(x_1, x_2) \in X_{1\|2} \mid x_1 \in X_{\mathrm{m},1} \wedge x_2 \in X_{\mathrm{m},2}\}$

An example is shown by the parallel composition of the two automata

- $G_1 = (\{1,2,3\}, \{\alpha, \beta\}, \{(1,\alpha,2), (2,\alpha,3), (3,\beta,1)\}, \{1\}, \{1\})$ and

- $G_2 = (\{1,2\}, \{\beta, \gamma\}, \{(1,\gamma,2), (2,\beta,1)\}, \{1\}, \{1\})$

in figure 3.3.

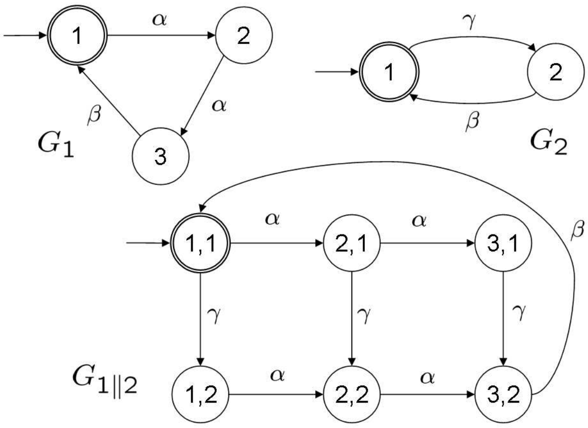

Figure 3.3: Parallel composition of automata G_1 and G_2

A basic algorithm that implements the parallel composition is given as follows.

Abstract Algorithm (Parallel Composition)**.** Input: Finite automata $G_1 = (X_1, \Sigma_1, \delta_1, X_{0,1}, X_{\mathrm{m},1})$ and $G_2 = (X_2, \Sigma_{2,}\delta_{2,}X_{0,2}, X_{\mathrm{m},2})$.

/ Starting at the initial state pairs, events in the shared alphabet $\Sigma_{shared} = \Sigma_1 \cap \Sigma_2$ are executed in parallel, while all other events $\Sigma - \Sigma_{shared}$ are executed asynchronously. */*
compute the set of initial states $X_{0,1} \times X_{0,2}$
/ Fill the waiting list. */*

$X_{waiting} := \{(x_1, x_{2,} (x_1, x_2)) \mid (x_1, x_2) \in X_{0,1} \times X_{0,2}\}$, $X_{1\|2} = \emptyset$, $X_{\mathrm{m},1\|2} = \emptyset$
/ The execution of events follows the scheme: */*
while $X_{waiting} \neq \emptyset$ **do**
    take a tuple $(x_1, x_{2,} (x_1, x_2)) \in X_{waiting}$ and remove
    for all $\sigma \in (\Lambda(x_1) \cup \Lambda(x_2))$ **do**
        for all $x_{1\|2} \in \delta_{1\|2}((x_1, x_2), \sigma)!$ **do**
            if $x_{1\|2} \notin X_{1\|2}$ **then**
                $X_{1\|2} = X_{1\|2} \cup \{x_{1\|2}\}$
                $X_{waiting} = X_{waiting} \cup \{(\delta_1(x_1, \sigma), \delta_2(x_2, \sigma), x_{1\|2})\}$
            end if
            add transition $((x_1, x_2), \sigma, x_{1\|2})$ to $G_{1\|2}$
        end for
    end for
    if $x_1 \in X_{\mathrm{m},1} \wedge x_2 \in X_{\mathrm{m},2}$ **then**
        $X_{\mathrm{m},1\|2} = X_{\mathrm{m},1\|2} \cup \{(x_1, x_2)\}$
    end if
end while

At first the initial state pairs are computed and put in the waiting set along with their respective states in G_1 and G_2. Then transitions in both automata are followed by executing shared events synchronously and unshared events asynchronously. This is carried out by retrieving a tuple of $(x_1, x_{2,} (x_1, x_2))$ from the set of waiting states $X_{waiting}$ and executing the transitions that link from the states in G_1 and G_2. Each time a new state pair is created it is put on the waiting list. Each state pair (x_1, x_2) that was put from the waiting set is marked if both states are marked in G_1 and G_2. The algorithm terminates when the waiting list is empty.

The symbolic notation of the algorithm operates on the automaton in Definition 2.1.2. A software implementation introduces access patterns to the automaton data model in place of the symbolic operations. Aside from primitive operations, the following set of data access patterns has to be taken into account for the concrete implementation of an automaton data model.

Data Access Patterns

- Direct access to the initial states is required to start the algorithm.

- A set of waiting states must be maintained. It contains tuples constisting of three states (or pointers to states) $x_1 \in G_1$, $x_2 \in G_2$ and $(x_1, x_2) \in G_{1\|2}$. The ordering of the set has no effect.

- Set inclusion tests for either Σ_{shared} or $\Sigma - \Sigma_{shared}$ are required for determining if an event is shared or not.

- For the determination of $(\Lambda(x_1) \cup \Lambda(x_2))$ the transitions of a given state have to be accessed. The order of the transitions is irrelevant at this point.

- Computing $\delta_{1\|2}((x_1, x_2), \sigma)$ requires determining if $\sigma \in \Lambda_1(x_1) \cap \Lambda(x_2)$, $\sigma \in \Lambda_1(x_1) \setminus \Sigma_2$ or $\sigma \in \Lambda_2(x_2) \setminus \Sigma_1$. For fast set inclusion test, the transitions of a given state must be ordered by their respective events.

- New transitions $\big((x_1, x_2), \sigma, \delta_{1\|2}((x_1, x_2), \sigma)\big)$ must be linked properly within $G_{1\|2}$. Therefore it must be determined if the states $\delta_{1\|2}((x_1, x_2), \sigma)$ have already been created, which may require a search through the complete automaton $G_{1\|2}$.

- The marked status of each new state $(x_1, x_2) \in G_{1\|2}$ must be checked for each pair $x_1 \in G_1$, $x_2 \in G_2$ that builds a new state in $G_{1\|2}$.

Now the set of data access patters can be evaluated in the data models introduced in Section 3.1.

Linked List Automaton Model

At first the linked list automaton model in Section 3.1.1 is evaluated:

- The inital states are directly accessible for starting the algorithm.

- The set of waiting states $X_{waiting}$ can be maintained efficiently by using three stacks with pointers to states. The elements on top of the respective stacks build the tuple of $x_1 \in G_1$, $x_2 \in G_2$ and $(x_1, x_2) \in G_{1\|2}$.

- The set of shared / unshared events can be directly computed as every automaton provides direct access to its alphabet.

- Transitions for a given state can be directly retrieved in the respective state object. The transitions must be ordered by their events.

- When linking new transitions $\left((x_1, x_2), \sigma, \delta_{1\|2}((x_1, x_2), \sigma)\right)$ in $G_{1\|2}$, the pointer to the state objects for $\delta_{1\|2}((x_1, x_2), \sigma)$ is not known at first by nature of the linked structure. Here an additional supporting data structure that maps pairs $(x_1, x_2) \in X_{1\|2}$ to state pointers is required. The map must be sorted to minimize search time. Without such an additional map $G_{1\|2}$ must be searched for $\delta((x_1, x_2), \sigma)$. The search may require traversing the complete automaton and should be avoided because of that.

- Checking the marking status of newly created states in $G_{1\|2}$ is fast, since the respective pair of states $x_1 \in G_1$, $x_2 \in G_2$ directly holds its marking status.

Set Based Automaton Model

The set based data model, introduced in Section 3.1.2, requires different access patterns:

- The initial states are directly accessible for starting the algorithm.

- As in the linked list model the set of waiting states $X_{waiting}$ can be maintained efficiently, e.g. by using three stacks with state identifiers. The elements on top of the stack build the tuple of $x_1 \in G_1$, $x_2 \in G_2$ and $(x_1, x_2) \in G_{1\|2}$.

- Like the linked list model the set based model provides direct access to the set of events for computing Σ_{shared} or $\Sigma - \Sigma_{shared}$.

- For the retrieval of the transitions for a given state, the set of transitions must have an order. The order must provide fast access for finding transitions that contain a specific predecessor state. In addition, an ordering of the transitions by their respective events per state is required.

- When new transitions are created in $G_{1\|2}$, the state identifier of the target state $\delta_{1\|2}((x_1, x_2), \sigma)$ may not be known at first. If a state identifier scheme is used that has no direct mapping from a pair $x_1 \in G_1$, $x_2 \in G_2$ to the corresponding state identifier of $(x_1, x_2) \in G_{1\|2}$, an additional sorted data structure that provides such a map is required.

- Checking the marking status of newly created states in $G_{1\|2}$ is straight forward by querying the set of marked states in G_1 and G_2. For a fast inclusion test, the set of marked states must have an order.

Model Comparison

The parallel composition has shown that the specification of both basic models has to be extended for ordered transitions. In the linked list model transitions have to be ordered locally at each state by their events. In the set based model a global order of transitions predecessor states first and events second is required. With this extended specification, both models fulfill the requirements of the parallel composition. Both need additional data structures for building the transition relation of $G_{1\|2}$. A further requirement is a sorted set of marked states for fast set inclusion tests. As conclusion, there is no preference for one of the models by the parallel composition method.

3.2.2 Language Projection Method

Definition 3.2.2 (Natural Projection [Won04])**.** For an alphabet $\Sigma_0 \subseteq \Sigma$ the *natural projection* $p_0 : \Sigma^* \to \Sigma_0^*$ is recursively defined as follows:

- $p_0\left(\epsilon\right) := \epsilon$
- $p_0\left(\sigma\right) := \begin{cases} \epsilon \text{ if } \sigma \notin \mathit{\Sigma}_o \\ \sigma \text{ if } \sigma \in \Sigma_0 \end{cases}$
- $p_0\left(s\sigma\right) := p_0\left(s\right) p_0\left(\sigma\right)$ for $s \in \Sigma^*$, $\sigma \in \Sigma$

The natural projection $p_o\left(L\right)$ removes all symbols $\sigma \notin \Sigma_0$ from strings in the language L by concatenation of the remaining fragments that contain only symbols $\sigma \in \Sigma_0$. For convenience, the natural projection is simply denoted *projection*.

While the projection of a language is straightforward, projecting the generated language of an automaton by manipulation of the automaton itself is not trivial. An abstract algorithm that hides implementation details is stated as follows[1].

Abstract Algorithm (Projection)**.** Given a finite automaton $G = (X, \Sigma, \delta, X_0, X_\mathrm{m})$ and a projection alphabet $\Sigma_0 \subseteq \Sigma$.

/ The method is initialized by putting all initial states in the set of waiting states */*
$X_{waiting} = X_0$
/ Starting at the initial states all reachable states are explored for transition paths with events* $\sigma \in \Sigma_0$ **/*
while $X_{waiting} \neq \emptyset$ **do**
    take a state x from $X_{waiting}$
    put x in X_{path}
    while $X_{path} \neq \emptyset$ **do**
        take a x_p from X_{path}
        for all $\sigma \in \Lambda(x_p)$ **do**
            if $\sigma \in \Sigma_0$ **then**
                $\delta(x, \sigma) := \delta(x_p, \sigma)$
                put state $\delta(x_p, \sigma)$ in $X_{waiting}$
            else
                put successor state $\delta(x_p, \sigma)$ in X_{path}
                if $\delta(x_p, \sigma) \in X_\mathrm{m}$ **then**
                    $X_\mathrm{m} = X_\mathrm{m} \cup \{x\}$
                end if
            end if
        end for
    end while
    for all $\sigma \in \Lambda(x)$ **do**
        if $\sigma \notin \Sigma_0$ **then**
            remove the transition from δ
        end if
    end for
end while

Initiallization is done by filling the waiting set with the set of initial states. Then, the reachable part of the automaton is explored in the following manner. After retrieving the next state from the waiting set, transition paths are executed until they end in an event contained in Σ_0. This may also be the first event in a path. While executing a

[1]Note that this algorithm creates a nondeterministic automaton. The *Deterministic* method which is required for converting a projected deterministic automaton back into a deterministic one is introduced in Section 3.2.3.

transition path, transitions with projected events that link from states in the path, are relinked at the starting state. Previously unexplored states along the path that contain transitions with projected events are added to the waiting list. If a marked state is reached by such a transition path the starting state is marked. After transition paths are explored for a state from the waiting list, all transitions that link from this state and have an event not in Σ_0 are removed from the transition relation.

Figure 3.4 shows the automaton

$$G = (\{1,2,3\}, \{\alpha,\beta,\gamma,\mu\}, \{(1,\alpha,2), (2,\beta,1), (2,\mu,3), (3,\gamma,1)\}, \{1\}, \{1\})$$

being projected to the alphabet $\Sigma_{proj} = \{\alpha,\beta,\mu\}$. This results in the projected automaton G_{proj} where $L_{\mathrm{m}}(G_{proj}) = p_0(L_{\mathrm{m}}(G))$ and $L(G_{proj}) = p_0(L(G))$.

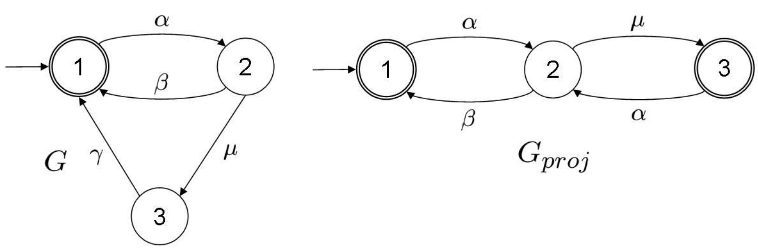

Figure 3.4: Projection of an automaton G to G_{proj} over the alphabet Σ_{proj}

As in the previous section, the abstract algorithm is evaluated for data access patterns that appear in a software implementation.

Data Access Patterns

- Direct access to the set of initial states is required.
- A simple unordered set of waiting states $X_{waiting}$ and locally unprocessed states X_{path} is required. For example both sets can be maintained by a stack.
- Set inclusion tests are required to determine whether an event σ is contained in Σ_0 or not. Therefore Σ_0 must be sorted.
- Adding transitions by the statement $\delta(x,\sigma) := \delta(x_p,\sigma)$ is trivial. This is in contrast

to parallel composition, where adding transitions to the new automaton requires traversing the automaton if no helper data structures are used.

- When deleting transitions, it has to be ensured that all states are still accessible in the data model after the deletion.

These data access patterns are investigated in context with the linked list automaton model and the set based automaton model.

Linked List Automaton Model

- Direct access to the initial states is given.
- Both sets $X_{waiting}$ and X_{path} have to be implemented as sets that hold pointers to states. Then fast access to the next state x_p is given.
- A supporting data structure is required for storing Σ_0 as a sorted set of events.
- Adding a transition $\delta(x,\sigma) := \delta(x_p,\sigma)$ simply is done by adding a new transition object at state x that points to state $\delta(x_p,\sigma)$ with event σ.
- Deleting transitions may have the effect that states and transitions linking from that states have no more link path to either an initial state and a marked state. These states and transitions are then lost in memory and cannot be accessed anymore. Although, this does not affect the projected language of the automaton, it can cause memory leaks in an implementation.

Set Based Automaton Model

- As in the linked list model, the initial states can be directly accessed to invoke the algorithm.
- $X_{waiting}$ and X_{path} can be implemented by a data structure that can hold an unordered set of state identifiers.
- A supporting data structure is required for storing Σ_0 as a sorted set of events.
- Adding transitions by $\delta(x,\sigma) := \delta(x_p,\sigma)$ is straightforward.

- Deleting transitions is trivial. It should be noted that "loosing" states by removing all transitions linking to them is not possible at all with this data model, because the whole transition relation is kept as single set.

The evaluations of the data models concludes in the model comparison.

Model Comparison

The projection algorithm has shown an important weakness in the linked list model. Without extensions, it cannot hold states that have no link path to either an initial state or a marked state. Although such automata are not covered by regular language theory, they may occur in the implementation of automata algorithms. In contrast, the set based model does not have this problem. It is well suited for the projection algorithm. For both models, a supporting data structure is required for doing set inclusion tests with the projection alphabet.

As a conclusion, both models can execute the algorithm, but the linked list model may lose states and transitions (that are not required to generate the projected language) in memory. It should be noted, that this is just one version of many possible language projection algorithms.

3.2.3 Data Model Evaluation by Subset Construction

The conversion of a nondeterministic finite automaton into a deterministic finite automaton is an important topic in automata theory. The method for the conversion is called *subset construction*. It is heavily used in [Sch05b] and other approaches.

Definition 3.2.3 (Subset Construction [HU79])**.** For a given nondeterministic finite automaton $G_{nd} = (X_{nd}, \Sigma, \delta_{nd}, X_{0,nd}, X_{\mathrm{m},nd})$ a deterministic finite automaton $G_d = (X_d, \Sigma, \delta_d, x_{0,d}, X_{\mathrm{m},d})$ can be constructed such that $L_{\mathrm{m}}(G_{nd}) = L_{\mathrm{m}}(G_d)$ where

- $X_d := \left\{S \subseteq 2^{X_{nd}} \mid S = x_{0,d} \vee \exists s \in \Sigma^* \text{ such that } S = \delta_d(x_{0,d}, s)\right\}$,

- $\delta_d(S,\sigma) = \bigcup_{x_{nd}\in S} \delta_{nd}(x_{nd},\sigma)$ where $S \subseteq 2^{X_{nd}}$,

- $x_{0,d} = X_{0,nd}$,

- $X_{\mathrm{m},d} = \{S \in X_d \mid \exists x_{nd} \in S$ such that $x_{nd} \in X_{\mathrm{m},nd}\}$.

This definition results in the following algorithm.

Abstract Algorithm (Subset Construction)**.** [Les95]: Given a nondeterministic finite state automaton $G_{nd} = (X_{nd}, \Sigma, \delta_{nd}, X_{0,nd}, X_{\mathrm{m},nd})$.

/ Initialization is done by defining the set of initial states of the nondeterministic automaton as the initial state of the deterministic automaton */*
$S_1 := X_{0,nd}$, $X_d := \{S_1\}$, $x_{0,d} := S_1$
$last = 1$
for $i = 1$; $i \leq last$; $i = i + 1$ **do**
    for all $\sigma \in \Sigma$ **do**
        / Create empty temporary set */*
        $S := \emptyset$
        for all $x \in S_i$ **do**
            $S = S \cup \{\delta_{nd}(x,\sigma)\}$ if $\delta_{nd}(x,\sigma)!$
        end for
        if $S \neq \emptyset$ **then**
            if $\exists S_k \in X_d$, $1 \leq k \leq last$ such that $S = S_k$ **then**
                Create transition $\delta_d(S_i,\sigma) := S_k$
            else
                $last = last + 1$
                $S_{last} := S$, $X_d = X_d \cup \{S_{last}\}$
                Create transition $\delta_d(S_i,\sigma) := S_{last}$
            end if
        end if
    end for
    if $\exists x \in S_i \mid x \in X_{\mathrm{m},nd}$ **then**
        $X_{\mathrm{m},d} = X_{\mathrm{m},d} + \{S_i\}$
    end if
end for

The new deterministic automaton is built of the nondetermistic one by constructing subsets (*power sets*) of X_{nd} which are linked by transitions. Initialization is done by putting the set of initial states of the deterministic automaton into a subset. From there on a new subset is constructed every time there is an event that causes at least one transition (with this event) from the subset which does not link back into the subset again. Then a new subset is created of the set of successor states of these transitions. This way, all transitions that link from a subset and contain the same event are mapped to a single transition in the deterministic automaton. Marking of states in the deterministic automaton is then done by marking every subset that contains at least one state which is marked in the nondeterministic automaton.

Again the significant data access patterns that occur in a software implementation are identified for both automata models introduced before.

Data access patterns

Unlike the evaluation of parallel composition and language projection, the evaluation of subset construction is based on existing work. In [Les95] efficient approaches to subset construction are developed. This includes both optimized algorithms and optimized data structures. Different algorithm implementations are suggested for different orders of the number of states (among other data model independent parameters that will not be covered here for the sake of simplicity). As large numbers of states are common in RW theory, only the relevant results for this case are presented here.

The algorithm holds two main areas in which most of the computational time is spent:

- The first is in the transition loop, where transitions containing the current event are searched at every state in the subset. This seems optimizable as most likely the transitions at a state will contain only few events compared to the overall alphabet. A multiway merge algorithm that solves this problem is available in [Les95], however, the algorithm is not covered here as its implementation is independent of a specific automaton data model.
- The second area is the test for set existence and equality in X_d after a new subset was created. Here additional data structures are required to build the deterministic automaton:
  - States in the resulting deterministic generator consist of power sets of the

states of the nondeterministic one ($X_d \subseteq 2^{X_{nd}}$). This requires a set data structure to hold a set of states. [Les95] suggests implementing subsets as heaps [Knu98]. A heap is an array-based implementation of a special type of binary search tree, that is complete up to the lowest level which may not be completely filled.

– According to [Les95], for every subset a set signature is computed that supports set inclusion tests in X_d. The signature for a power set may not be unique but in set comparisons it must limit the number of possible matches to few sets compared to the overall number. Finding a good algorithm for computation of a signature for a power set heavily depends on the exact data type stored in the set and is not covered at this point.

– Comparing power sets requires a hash table to look up all the sets matching a given signature. Then X_d is implemented as an array of subsets with an associated hash table for set inclusion tests. It can also be directly implemented as a hashed set. If an array is used to store the subsets the array index can be used directly to implement the Index i used in the abstract algorithm.

In combination with the proposed data structures, the algorithm introduces the following data access patterns:

- Initialization is done by creating a subset that contains all initial states of the nondeterministic generator. The subset is hashed by its set signature and put into X_d.

- For every newly constructed deterministic state an iteration over all events in Σ is required.

- For every event in the alphabet iteration at first an empty power set is created. Then all transitions that contain the current event are followed by evaluating every state in the power set for transitions with the event from that state. Pointers to the successor states are then stored in the new power set.

- If the newly created power set is still empty, processing stops at this point and continues with the next event in Σ.

- For nonempty power sets it has to be determined if it is already contained in X_d:
  - Compute the set signature for the new set.
  - Look up the matching subsets in the hash table.
  - If one of the subsets is equal to the new set use the existing one for linking the transition in the next step. In the other case add the new subset to the array of subsets, store its array index in the hash table and mark the set as waiting.
- Finally, a transition in the deterministic automaton is introduced by linking the subset of the deterministic predecessor state with the subset of the deterministic successor state via the current event in the event loop.
- A subset in X_d has to be marked if one of its nondeterministic states is marked.

There exist many ways to build the new deterministic automaton in the original data model. The most convenient idea seems to be constructing the deterministic automaton on the fly by linking each new subset in X_d with a newly created state in the deterministic automaton. Transitions and marked states are then established in the usual way while the subsets are only used for constructing new deterministic states and set inclusion tests. It should be noted, that this is only possible because states in the deterministic automaton (in the original data model) are always accessed through their corresponding subsets and never the other way. So only unidirectional links from the subsets to their associated states are required.

At next these data access patterns are evaluated with both automata models.

Linked List Automaton Model

- Starting the algorithm is straight forward by creating a power set containing pointers to the set of initial states of the nondeterministic automaton. In the new deterministic automaton a initial state is created and a link established from the power set to the state. The set is put in the waiting list.
- While in the algorithm the deterministic automaton consists of subsets, that are

linked by transitions and can be marked, the way of constructing a deterministic linked-list automaton is different. Here a new linked list automaton (with the natural state objects the model provides) is constructed on the fly.

Processing a waiting power set causes the following data access patterns:

- Iterating over the alphabet of the automaton is straightforward.

- For every event in the alphabet iteration, at first an empty power set is created. Then for each event in the alphabet iteration all states in the set are evaluated for transitions that contain the current event. Therefore a sorting of the transitions by their events is required. For each transition that matches the current event, a pointer to its target state is stored in the newly created power set. This is simply done by copying the state pointer, held by each transition.

  – If the newly created power set is still empty, processing stops at this point and continues with the next event in Σ.

  – For a nonempty power set it has to be determined if it is already contained in X_d. This is done by a set signature test. If subsets in X_d have the same signature, these subsets have to be compared against the new power set. If no matching set is found, a new state is created in the deterministic automaton and a link established between the subset and the state.

  – At last the transition has to be added in δ_d by linking the predecessor power set and the successor power set with the respective event. This is directly done by creating a transition between the corresponding state objects both subsets respectively point to.

- For each new subset in X_d, it has to be determined, if at least one of its nondeterministic states is marked. For this all the states where the subset entries point to have to be evaluated for their respective marking flag. If a nondeterministic state is marked, the state in the deterministic automaton is marked.

Set Based Automaton Model

- Algorithm initialization is simply done by copying the set of initial states of the deterministic automaton as a new power set into X_d. A new state is added to the set of states and the set of initial states of the deterministic automaton. A link between the power set and the state identifier of the created state has to be established. The new power set is added to the waiting list. From there on the waiting list is processed subset by subset.

Processing a subset includes the following data access patterns:

- Iteration over the alphabet is directly possible.

- A new empty power set is created for the current event. Then all transitions containing that event are followed for every state in the set. This is done by an iteration over the states in the (predecessor) subset. For each state the transitions from the state with the current event have to be searched in the set of transitions. This requires the set of transitions being sorted both by state and by event. For each matching transition the successor state is put into the new power set.

- For nonempty power sets, the procedure is nearly the same as in the linked list model. It has to be determined if the new set is equal to an existing subset in X_d. If the new set is unique, it is put into X_d and a new state identifier is created in the set of states of the deterministic automaton. Then a link is established between the subset and the state identifier.

- At last the transition in the deterministic automaton is added by adding a new transition object to the set of transitions. The new transition object contains the state identifier of the predecessor state, the event and the state identifier of the successor state.

- Every subset has to be tested for marking states to determine the marked status of its corresponding deterministic state. Therefore, the set of marked states of the nondeterministic automaton must be sorted.

Model Comparison

The evaluation of both data models has shown that the algorithmic implementation is straightforward for both data models up to the set inclusion tests for X_d with newly created power sets. Aside from small implementation details, namely the storage of either pointers to states or state identifiers in subsets, data access patterns for comparing new subsets with existing ones in X_d are identical for both automaton models. Again the transitions in the linked list automaton model were required to be sorted at each state while the set based model required the set of transitions being sorted by predecessor state and event.

The subset construction method concludes the data model evaluation by means of algorithms. At next the computational complexity of typical operations is investigated for both models.

3.2.4 Computational Complexity of Important Data Access Patterns

In this section the computational complexity of the most important data access patterns in the above algorithms is evaluated.

It is assumed that a sorted set is implemented as a balanced binary search tree [AHU+83, Knu98] with elements additionally being linked in order. In such a search tree finding an element never takes more than $O(\log n)$ steps, where n is the number of entries in the set. There may be other implementations for sorted sets that have a better average search complexity, but as most standard libraries of modern programming languages provide at least one implementation of a balanced binary search tree, this sorted set type is used for evaluation.

The previously analyzed algorithms have shown that the transitions have to be sorted. For the linked list model it is assumed, that transitions at each state are sorted in the order of their events first and successor states next. For the set based model an ordering by predecessor states, events and successor states of the set of transitions is assumed. In addition an ordering of the set of marked states is assumed for the set based model.

For a given automaton $G = (X, \Sigma, \delta, X_0, X_{\mathrm{m}})$ the number of states is denoted x, the number of transitions t and the number of (local) transitions that link from a given state t_l.

As the evaluation of the three methods above has shown, the most important access patterns deal with accessing the transition relation of an automaton. Therefore the typical transition access operations are evaluated. In addition the test for the marking status of a state is investigated.

Find all transitions that link from a given state

- Linked List Model
  - For a given state the transitions that link from this state can be directly accessed. Therefore the complexity for finding the first transition is $\mathcal{O}(1)$. Accessing the next transition at the state requires one step. So accessing all transitions linking from a state has the complexity $\mathcal{O}(t_L)$.
- Set Based Model
  - Searching for the first transition that matches a given state causes a maximum of $\log t$ steps. Therefore the complexity is $\mathcal{O}(\log t)$ for finding the first element. If more than one transition links from the state accessing the next transition causes one step. So accessing all transitions linking from the state is of complexity $\mathcal{O}(\log t + t_l)$ where $\log t + t_l < t$.

Find all transitions that link from a given state and contain a specific event

- Linked List Model
  - For a given state, the transitions linking from that state can be directly accessed. So a search of order $\mathcal{O}(\log t_l)$ is required. If the state holds more than one transition with the event, accessing the next transition requires one step. Let e be the number of transitions containing the event. Then the complexity of accessing all matching transitions is $\mathcal{O}(\log t_l + e)$ where $\log t_l + e < t_l$.

- Set Based Model
  - The search for the first element that matches the predecessor state and the event requires a maximum of $\log t$. All further transitions that hold the state and the event are incrementally accessible with one step each. Let e be the number of transitions containing the event. So the overall complexity is $\mathcal{O}(\log t + e)$ where $\log t + e < t$.

Find all transitions that link to a given state

Let t_r be the number of transitions that link to a given state.

- Linked List Model
  - According to the model definition, pointers to states are implemented bidirectional. Thus, the first transition linking to a state is directly accessible with $\mathcal{O}(1)$. Accessing the next transition linking to the state requires one step. This results in the overall complexity $\mathcal{O}(t_r)$.
- Set Based Model
  - The transition relation is sorted in the order predecessor state - event - successor state. From this it follows that finding all transitions linking to a state always requires t steps. The complexity then is $\mathcal{O}(t)$. This suggests a reordering of the transition relation by successor state, event and predecessor state if this access pattern required several times in an algorithm. Then the complexity for finding all matching transitions is $\mathcal{O}(\log t + e)$ as stated before.

Find all transitions that link to a given state and contain a specific event

Let t_r be the number of transitions that link to a given state.

- Linked List Model
  - Because of the bidirectional implementation of the state pointers in transitions, direct access to the transitions is possible. Then finding all transitions

that contain the event requires $\log t_r$ steps. Therefore the order of complexity is $\mathcal{O}(\log t_r)$ for finding all matching transitions.

- Set Based Model
  - The complexity is exactly the same as in the case of finding all transitions that link to a given state. Again a complete search through the transition relation with complexity $\mathcal{O}(t)$ is required here.

Marking Status Test

Testing the marking status of a given state is an important operation required in nearly all SCT algorithms.

- Linked List Model
  - Testing the marked status of a given state simply requires evaluating the marking flag at the state object and therefore is of order $\mathcal{O}(1)$.
- Set Based Model
  - To test a given state for its marking status a search in the sorted set of marked states is required. Let x_{m} be the number of marked states in the automaton. Then the complexity for determining the marked status is $\mathcal{O}(\log x_{\mathrm{m}})$.

Model Comparison

The evaluation has shown, that the linked list model takes big advantage of its transitions being directly accessible at each state. Therefore algorithms that traverse the linked transition structure of an automaton will be faster with a linked list automaton model than with a set based model. The marking status test for a state has also shown the linked list model requires less computational steps here.

However, it is important to note, that SCT algorithms often strongly depend on the efficiency of supporting data structures. For this reason the complexity evaluation of the access patterns can only be seen as a synthetical benchmark for the raw performance of the models. Therefore the results of the performance evaluation may only be used as an additional criterion in relation to the evaluation by algorithms.

3.2.5 Conclusion

Two basic automata data models have been evaluated with three methods commonly used in supervisory control theory. The evaluation started with the parallel composition method followed by the language projection method and ended with the subset construction. For every method an abstract algorithm was presented. Then the algorithm was used to explore the data access patterns that occur in a software implementation of the algorithm. These data access patterns were finally evaluated for both data models, followed by a short conclusion for each method. In addition important data access patterns were analyzed for their computational complexity with both models.

While the overall evaluation by SCT methods has shown that all three algorithms can be implemented with both data models, the evaluation of the projection method revealed an important weakness of the linked list model. The linked list model is unable to handle a fragmented transition relation or states that have no transition link to either the initial states or the marked states. Such an automaton cannot be generated by marking a given language, but may emerge in automata algorithms. Therefore the evaluation by algorithms has given a preference for the less restrictive set based automaton data model. In contrast, the complexity evaluation of data access patters has shown, that the linked list model requires less computational steps in important access patterns that can be often found in loops of SCT algorithms.

The evaluation has further shown that a more detailed specification is required for both automata models to fulfill the requirements of computation in SCT. For the linked list model, a sorting of the transitions by their respective events is required. This holds as well for the set based model, where the set of transitions must be sorted by predecessor state, events and maybe by successor state.

As a conclusion the linked list model provides faster computational performance but is restricted to a linked state structure while the set based model has no restrictions at all but provides less computational performance.

A different approach for the modelling of finite automata in software is using binary decision diagrams (BDDs). This is presented in [ZW01]. Binary decision diagrams enable a memory saving representation of finite automata. However, this is still an action of research, since only few supervisory control methods are known to be implemented by using BDDs. Therefore the implementation of algorithms for SCT using BDDs is not covered in this thesis.

3.3 Basic Automaton Implementation

The evaluation of the linked list automaton model and the set based automaton model has shown a preference for the set based model in terms of flexibility. As the main goal of this thesis is to provide a software library for RW control theory, that can be easily extended to other theoretical approaches, the abstract set based model is used as the basis for specifying the real implementation of an automaton.

At first the specification of the set based automaton model is developed in Section 3.3.1 by discussing important aspects regarding the implementation of event identifers and state identifers. To keep the implementation of the automaton class and the algorithms that operate on this class independent of the details of a specific programming language, abstract data types (ADT) are introduced in Section 3.3.2. These abstract data types are used in subsequent chapters when specifying the algorithms for all methods covered in the thesis. Section 3.3.3 then finally introduces the implementation of the object orientated automaton class along with its basic functions.

3.3.1 Automaton Model Specification

Several important aspects of implementing an automaton software model that is both well suited for computation and user friendly are discussed. In this section, the results the specification of the data model is given.

Event Identifiers and State Identifiers

At first the definition of the event identifiers and the state identifier data types, used in the abstract model is required for the real implementation. Both event identifiers and state identifier data types must be able to hold different entries in the order of millions. For fast set inclusion tests, comparing two identifiers should be an operation that requires as less computational steps as possible. Both requirements are very well fulfilled by using the natural integer data type of the used computer architecture for the representation of state and event identifiers. The natural integer data type of a computer architecture is usually specified by the data type `int`.

As DES usually are modelled by more than one automaton, for events the assigned numbers must be consistent in all automata, that share the events. This requires that a modelled event must be uniquely numbered within all automata sharing the event. In contrast state numbers have no relation between different automata, as the generated language of automata is independent of any state identifiers. So integer state numbers must only be unique within an automaton.

Such a representation of states and events by integer numbers is not very user friendly. Especially when modelling real world DES it is required to know, which real world events correspond to an event number in the model and what is the meaning of a state number in the real world system. Therefore, a mapping between symbolic names and integer numbers is required for both events and states. This is discussed at next.

Symbolic Event Names

If several automata operate on shared events, the set of shared events will be called an *event domain*. If an event is added to the alphabet of an automaton and the event is already known in the domain, the matching event number must be added to the alphabet of the automaton. From the user input, an event most likely will be added by a symbolic name, that corresponds to a specific integer number. Then the number of the event is not known at first because the automaton does not have any information about mappings between symbolic event names and event numbers.

This problem is solved by an entity outside the automata that assigns integer numbers to symbolic names and looks up the numerical index of a symbolic event name for all automata. The properties of such an entity are stated as follows:

- Each time a new event is added to any automaton alphabet in the same event domain, a new unique integer index is assigned to the event. This can be implemented by a counter initialized with 0 that is incremented each time a event is added where the incremented value is assigned as the integer number of the event.
- The entity holds a mapping of event numbers to the symbolic names of the events. For convenience a mapping of symbolic event names to the respective integer numbers can be provided to accelerate the number lookup for a given event symbol. Providing such a reverse mapping is uncritical for memory usage, as the number of events is usually small compared to the number of states or transitions in an automaton.
- Symbolic event names have to be unique.
- When an event is deleted from the alphabet of an automaton, it is not removed from the entity.

Such an entity that provides unique event indices with maps for looking up event numbers by symbolic event names as well as looking up symbolic event names by event numbers is denoted an *eventsymboltable*.

Symbolic State Names

Analogous to event numbers, states may also be assigned symbolic names. By nature of the state identifiers, symbolic state names must only be unique within an automaton, while several automata may contain states with different state numbers but sharing the same symbolic name.

Here, a possible performance problem emerges from the execution of algorithms that construct a new automaton input from existing automata. As an example, the parallel composition method builds a new automaton, where each state in the resulting automaton represents a state in each of the two input automata. If the symbolic state names in the resulting automaton should represent the symbolic state names in the input automata, then for each state in the new automaton a string operation is required that constructs a symbolic state name the corresponding two state names in the input automata. This slows down the computation process and increases memory usage. Another example for the problem is subset construction, where states in the deterministic automaton consist of subsets of states of the nondeterministic automaton. So symbolic state names in the deterministic automaton must contain the names of all the states in their corresponding subset of nondeterministic states.

A solution for this problem, that addresses the decrease of computational performance as well as the increasing memory usage is difficult to find, as there is no direct mapping of a integer state number in the output automaton to the integer state numbers in the input automata. However, several approaches are possible to solve the problem partially:

- Symbolic state names for the resulting automaton are computed depending on the size of the input automata. The size can be specified e.g. by the number of states or transitions.

- Each implementation of an algorithm that constructs a new automaton has a binary parameter that turns on the computation of state names in the resulting automaton. This leaves the decision of computing symbolic state names to the user. Also the automaton model may have such a binary flag. Then, if one of the input automata of an algorithm has the flag set to not compute state names, this holds for the constructed automaton, too.

- Some algorithms allow computing symbolic names at the end of the algorithm without any performance tradeoffs. This is possible for every algorithm that requires holding a data structure, that maps states in the new automaton to states in the input automata until the algorithm finishes. Then at least the computational performance of the algorithm is not decreased if state name computation is not requested.

- Additional data structures can be used to store mappings of state numbers in new automata to state numbers in input automata. Then the symbolic state name of an automaton can be computed on demand. This also has the advantage, that the formatting of the symbolic state name string can be adjusted on request, which may be an interesting feature in a graphical user interface. Also this enables to rename states in the new automata that are constructed by an algorithm, while still being able to resolve the corresponding states in the input automata of the algorithm. The problem here consists in developing a data structure that can hold state mappings for different algorithms.

All approaches have in common that they do not really solve the problem. As a conclusion the best solution seems to be a combination of the last three approaches by providing additional data structures to store state mappings for the most important algorithms (that create new automata) and additionally allow the direct computation of symbolic state names in each algorithm on user demand.

Storage of Symbolic State Names

It was already shown that events require an entity outside any automaton model to manage the relation between symbolic event names and event numbers. For states such an extern entity is not directly required, as state numbers in different automata have no relation to each other. To develop an appropriate specification for the storage of symbolic state names, the advantages and disadvantages of storing symbolic state names locally in each automaton are investigated.

Local storage of symbolic state names within an automaton is easy to implement. This just requires each entry of the ordered set of states to have an additional link pointing to a string object, that contains the symbolic name of the state. The string is empty, if a state has no symbolic object associated. Another implementation could be to have an additional map object in the automaton that maps state indices to symbolic names. A map from symbolic names to state numbers should only be created when reading a file, as a complete search through all symbolic names for finding its index is fast enough for user interactions and holding.

Therefore, storing symbolic state names locally in an automaton is a good solution for directly computing state names in algorithms. This approach requires an automaton to have an integer state index counter for assigning a unique state number when creating a new state in the automaton.

Mapping state numbers in constructed automata to their corresponding state numbers in the original automata then additionally requires storing a pointer to the automaton with each with a state number, which increases memory usage. Therefore a solution that doesn't require accessing an automaton for looking up the symbolic name of a state is better suited at this point.

Using additional data structures for mapping states in resulting automata to states in input automata of an algorithm is appropriate for a software library, that is developed for general use in different supplementary software projects. For being able to implement such additional data structures, a different approach is required for the storage of symbolic state names. Here the mapping of state numbers to symbolic names must be implemented outside the automaton, similar to the handling of symbolic names for event numbers.

Therefore, a solution to handle symbolic state names by an entity located outside of automata objects is specified as follows:

- State numbers have to be globally unique. To enforce this restriction, an integer type counter can be used to always assign the next larger integer value to a newly created state in an automaton in the event domain. The counter is initialized with 0 and incremented any time a new state number is assigned. Each time a new state is added to an automaton, the next available state number is requested from the entity and associated with the state. As the highest possible value of an unsigned integer on a 32bit computer architecture is 4294967296, there are enough free indices to assign[1].
- For storing symbolic state names, a mapping of integer numbers to symbolic state names is required. The same name may be associated with different state numbers, as long as the numbers correspond to states in different automata. A search for the number of a symbolic name is done by a complete search through the map

[1]For very special applications, the integer range may not be large enough. Then there is always the option to choose a integer type of double `int` length.

with restricting the result to state numbers in the respective automaton. This is fast enough for user interactions in any case. An additional mapping of symbolic names to numbers is only required for reading large automata from the filesystem, as resolving each symbolic state name in the file by a search through the complete map may require too much computation.

- If a state in an automaton is deleted, the state must be removed from the map. If a whole automaton is deleted then all state numbers in the automaton must be deleted in the map.

- As some algorithms may require operations on duplicates of an automaton or on a duplicated part of the automaton, it must be possible to create temporary copies of an automaton that contain the same state numbers. The deletion of the automaton copy may involve automatic state name deletion. Therefore a copy counter is required for each automaton so only when the last instance of an automaton is deleted, symbolic state names are removed from the map.

An entity that provides globally unique state numbers and a mapping of state numbers to symbolic names is denoted a *statesymboltable*.

With the results of this section the data structure of the automaton model can be specified.

Specification of the Automaton Data Structure

The data structure of the 5-tuple automaton $G := (X, \Sigma, \delta, X_0, X_\mathrm{m})$ is specified as follows:

- The set of states X is implemented as a sorted set of integer indices that are unique within an event domain.

- The alphabet Σ is implemented as a sorted set of integer indices that are unique within an event domain.

- The transition relation δ consists of a sorted set of unique transition objects. Each transition object holds the integer index of a predecessor state, the integer index of an event and the integer index of a successor state. The set is sorted by predecessor state first, event next and last by successor state. The transition objects may only contain state indices of X and event indices of Σ.

- The set of initial states X_0 is implemented as a sorted set of integer indices that must be contained in X.
- The set of marked states X_m is implemented as a sorted set of integer indices that must be contained in X.

In addition, the automaton data structure contains the following objects:

- A pointer (or reference) to an *event symbol table* object as specified before.
- A pointer (or reference) to a *state symbol table* object as specified before.
- A integer type counter for copys of the automaton object. The counter is initialized with 0 and incremented or decremented each time a unique copy of the automaton object is created or deleted. If an automaton object is deleted and its copy counter is 0, then the state symbol table is cleared from state indices contained in X. The copy counter has to be shared by all copies of an automaton object.

Every time an event is added to Σ by its symbolic name the event symbol table is used to look up its integer index that will be stored. If the symbolic event name is unknown in the event symbol table a new integer index is created and stored in the symbol table. When a new state is added to X, a new unique index is requested from the state symbol table that is then stored in X. New states can be optionally added by specifying a symbolic state name. Then a new unique index is created for the name and stored in X.

At next abstract data types are used to describe the implementation of the automaton class.

3.3.2 Abstract Data Types

According to the results of Section 3.2, the set of transitions and the set of marked states must be sorted. This requires the specification of an abstract data type (ADT) for a sorted set. In addition the ADTs map, vector and stack are introduced for the specification of the automaton model in the next section and the notation of algorithms in subsequent chapters. For all ADTs an implementation can be found in nearly all standard libraries of modern object orientated programming languages[1]. If a programming language does not provide ADTs, abstract implementations of the most important ADTs can be found in [AHU+83, Knu98]. An ADT that is used to store an amount of data is also often called a *container*.

In the algorithm implementations the methods of an ADT will be denoted by the ADT object as the first variable.

Before specifying the container ADTs, the `Iterator` ADT is introduced. An iterator is an abstract object that holds a position in a container ADT by hiding implementation details of the container. Most containers have methods that return iterators to access the data at a position. Iterators can also be incremented to access the next position. The (virtual) next position after the last element is specified by the End method, if defined for the container.

Pair

The Pair ADT specifies a pair of two values. A pair by a `Pair<TypeFirst, TypeSecond>` statement, where `TypeFirst` specifies the type of the first value, while `TypeSecond` specifies the type of the second value. The values stored in a `Pair` are directly accessed by specifying the `Pair` object in conjunction with `first` or `second` separated by a dot.

Set

The ADT `Set` specifies an ordered set of values. A set is created by a `Set<Type>` statement, where `Type` specifies the data type stored in the set. The data type must

[1]However only implementations that store values in their natural data type should be used. If an implementation of an ADT stores values encapsulated in abstract objects this requires a cast of the object into the real type at each access, which may slow down computation speed enormously.

have a defined order. A set may not have any duplicates. Incrementing a set iterator causes the iterator to move to the next element in the sorted range.

Usually a set is implemented as a balanced binary search tree. Then accessing an element requires a maximum of $\log n$ steps, where n is the number of elements in the set.

The primary methods of a set are:

- `Insert(element)`
  Inserts an element.

- `Erase(element)`
  Erases an element.

- `Begin`
  Returns an iterator to the first element or to the end of the set if the set is empty.

- `End`
  Returns an iterator to the end of the set.

- `Find(element)`
  Searches for an element and returns its position in the set by an iterator.

- `Exists(element)`
  Searches for an element and returns `True` or `False`.

- `Size`
  Returns the number of elements in the set.

- `Empty`
  Returns `True` or `False` if the set is empty or not.

- `Clear`
  Clears all set entries.

Map

The ADT `Map` specifies an ordered set of keys where each key has an associated value. A map is created by a `Map<KeyType,ValueType>` statement, where `KeyType` specifies the data type of the search keys and `ValueType` specifies the data type of the associated values. The search key data type must have a defined order and there may be no duplicate keys. The set only allows direct search for keys, not for values. Incrementing a map iterator causes the iterator to move to the next element in the sorted range.

Analogous to the `Set` ADT, a `Map` usually is usually implemented as balanced binary search tree with the computational complexity $\mathcal{O}(\log n)$ for accessing a set with cardinality n.

The primary methods of a map are:

- `Insert(key,value)`
  Inserts a key with an associated value.
- `Erase(key)`
  Erases a key with its associated value.
- `Begin`
  Returns an iterator to the first element or to the end of the map if the map is empty.
- `End`
  Returns an iterator to the end of the map.
- `Find(key)`
  Searches for a key and returns its position in the map by an iterator.
- `Exists(key)`
  Searches for a key and returns `True` or `False`.
- `Size`
  Returns the number of elements in the map.
- `Empty`
  Returns `true` or `false` if the map is empty or not.

- `Clear`
  Clears all set entries.

- `Lookup(key)`
  Returns the value of the key at a specific position.

In addition, value pairs in a `Map` object can be set and retrieved by the `[]` operator and be used like the `Pair` ADT introduced before. For the symbolic notation of algorithms the set of keys in a map will be denoted by the statement `Keys()`.

Vector

The ADT `Vector` specifies a resizeable, unsorted array, in which elements are accessible by their positional index. A `vector` is created by a `Vector<Type>` statement, where `Type` specifies the data type that is stored in the vector. To access the data at an index the name of the vector variable is specified with the positional index attached within the `[]` operator by $object\,[index]$. The `Vector` type provides fast random access, but inserting new elements in the middle of the vector requires all following elements being copied to higher positions. Incrementing or decrementing a vector iterator causes the iterator to move to the next or previous positional index.

Aside from accessing elements by the `[]` operator, the primary methods of a vector are:

- `Begin`
  Returns an iterator to the first index position. If the vector is empty the end of the vector is returned.

- `End`
  Returns an iterator to the end of the vector.

- `Push_Back(element)`
  Inserts a new value at the end of the vector.

- `LastIndex`
  Returns the position index of the last stored element (the size of the vector may be bigger).

- `Size`
  Returns the current size of the vector.
- `Empty`
  Returns `True` or `False` if the vector is empty or not.
- `Resize`
  Resizes the vector to a given number of element positions. This may delete existing elements if the vector is downsized.
- `Clear`
  Removes all elements in the vector and sets its size to zero.

Stack

The `Stack` ADT specifies a stack of elements where only the the top element is accessible. A stack is created by a `Stack<Type>` statement, where `Type` specifies the data type that is hold in the stack. The stack does not provide iterators.

The primary methods of a stack are:

- `Push`
  Puts an element on top of the stack.
- `Pop`
  Retrieves the topmost element and deletes it from the stack.
- `Get`
  Retrieves the data value of the topmost element.
- `Empty`
  Returns `True` or `False` if the list is empty or not.
- `Clear`
  Removes all elements from the stack.

Using the ADTs defined above, the automaton data model implementation is introduced.

3.3.3 Basic Generator Class Implementation

In this section the implementation of the specified automaton data model as an object orientated class is presented. The class is called *Generator* as also other types of automata are known that do not generate and mark a regular language.

At first further ADTs are introduced. The `EventSet` ADT holds an ordered set of event indices of the type integer and implements the alphabet object of an automaton as specified in Section 3.3.1. In contrast the `StateSet` ADT holds an ordered set of different state indices of the type integer. For a more general representation the type integer is defined as `Idx`, as every event and every state is specified to have an unique index within an event domain. Then an `EventSet` holds events of type `Idx` and a `StateSet` holds states of type `Idx`. To support symbolic event names and state names the ADT classes `EventSymbolTable` and `StateSymbolTable` will be used. Transitions are represented by objects of the class `Transition`, which is stored in the ADT `TransSet`. This holds a sorted set of `Transition` objects. A sorting order for the transitions can be specified on creation of a `TransSet` object. The `Generator` finally consists of a composition of objects of these classes.

EventSet

The `EventSet` ADT consists of a `Set<Idx>` and contains an additional pointer to an `EventSymbolTable` object that ensures that event indices and symbolic event names are handled consistently in a domain. In algorithms it will be denoted by the alphabet symbol Σ. Its basic methods are:

- `InsEvent(event)`
  Puts an event in the set. The event can be specified by either `Idx` or symbolic name.

- `DelEvent(event)`
  Removes an event from the set. The event can be specified by either `Idx` or symbolic name.

- `ExistsEvent(event)`
  Tests if a event is included in the set. The event can be specified by either `Idx` or symbolic name.

In algorithms the methods on `EventSet` objects will be stated in set notation by $\cup$, $\setminus$ and $\in$.

StateSet

The `StateSet` ADT consists of a `Set<Idx>` and contains an additional pointer to a `StateSymbolTable` object which provides unique state indices and symbolic state names for an event domain. In algorithms the `StateSet` will be denoted by the symbol X. Its basic methods are:

- `InsState(), InsState(name)`
  Gets the next larger state index from the `StateSymbolTable` object and puts it in the set. The name parameter is optional and can associate a symbolic name with the new state index.

- `DelState(state)`
  Deletes an index from the set. If the index had a symbolic name associated its entry is removed from the `StateSymbolTable` object. The state parameter may by specified by either `Idx` or symbolic name. In the latter case a name lookup of all indices in the set may be required to find the corresponding state index.

- `ExistsState(state)`
  Tests if a state is included in the set. The state parameter may by specified by either `Idx` or symbolic name. In the latter case a name lookup of all indices in the set may be required to find the corresponding state index.

Analogous to the EventSet, in algorithms operations on `StateSet` objects will be stated in set notation by $\cup$, $\setminus$ and $\in$.

To support subset construction another method is required:

- `Signature()`
  This method computes a set signature to accelerate algorithms that compare sets of states. In [Les95] a proposal is given for computing such a set signature. In the practical implementation of the library the simple SIGNATURE function stated in Figure 3.5 is used.

function SIGNATURE(X)
  `Idx` $sig := 0$
  `int` $i := 1$
  for all $x \in X$ **do**
    $sig := sig + x * i$
    $i := i + 1$
  end for
  return sig
end function

Figure 3.5: Simple set signature

Transition

Transitons are stored in objects of the ADT `Transition`. A `Transition` object contains three members of type `Idx`. The predecessor state is stated by `x1`, the event by `ev` and the successor state by `x2`, which also corresponds to the algorithmic notation. Access to the members will be denoted by transition object name and member, separated by a dot.

TransSet

The `TransSet` ADT consists of a `Set<Transition>`. The set order can be specified by an optional parameter `TransSet<SortType>` at the creation of a `TransSet`, where SortType is self explanatory. The sort types of Sortx1evx2, Sortx1x2ev, Sortevx1x2, Sortevx2x1, Sortx2x1ev or Sortx2evx1 are used. When no order is specified, the default order Sortx1evx2 is used. In algorithms a TransSet is specified by the transition relation symbol δ. Its primary methods are:

- SetTransition(x1, ev, x2), SetTransition(t)
  Adds a new Transition object to the set, either be specifying the indices of the states and the event or by specifying an existing Transition object t.
- DelTransition(x1, ev, x2), DelTransition(t)
  Removes a transition from the set.
- Transitions(x1)
  Returns an iterator to access all transitions with predecessor state x1 in order. The method may only be called if Sortx1evx2 or Sortx1x2ev was specified as set order.
- Transitions(x1,ev)
  Returns an iterator to access all transitions with predecessor state x1 and event ev in order. The method may only be called for a Sortx1evx2 set ordering.
- TransitionsByx2(x2)
  Returns an iterator to access all transitions with successor state x2 in order. The method is only valid for a Sortx2evx1 and Sortx2x1ev set ordering.
- TransitionsByx2ev(x2,ev)
  Return an iterator to access all transitions with successor state x2 and event ev in order. The method is only valid for a Sortx2evx1 set ordering.

In terms of TransitionsByx2 and TransitionsByx2ev for every sorting order access methods are defined to retrieve iterators for parts of a respectively sorted set of transitions.

In algorithms the methods are stated by providing the respective TransSet object as a parameter in front of the method parameters, e.g.

$$t \in \texttt{Transitions}(\delta, \texttt{x1})$$

denotes an iteration over all transitions of δ that have a predecessor state x1.

EventSymbolTable

The EventSymbolTable ADT is not directly used in algorithms, but required by EventSet objects for the management of symbolic event names. The associations between indices and symbolic names is hold by two maps. A Map<Idx,string>[1] object holds associations from indices to names and a Map<string,Idx> object the associations from names to indices. The type string represents the symbolic name type of a programming language. A Idx type counter is used to provide the next unused unique event index. It has the following main methods:

- InsEvent(name)
  This returns a new unique event index and stores the association in its internal maps. Events have always to be added by providing a symbolic name.

- Name(index)
  Returns the symbolic name for an existing index.

- Index(name)
  Returns the event index for a symbolic name.

Once created indices stay as long as the EventSymbolTable object exists. The methods of an EventSymbolTable object usually will be called indirectly by an EventSet, when calling an EventSet method.

StateSymbolTable

The StateSymbolTable ADT is the counterpart to the EventSymbolTable and provides similar services for states. In difference it only holds a Map<Idx,string> to store mappings from state indices to symbolic names while providing no reverse

[1]For optimized performance also a Vector<string> can be used if Idx is compatible with the Vector index type. However the more general Map solution is used here.

lookup. It also contains an `Idx` type counter to provide unique state indices. Its main methods are:

- `InsState(), InsState(name)`
  This returns a new unique state index and optionally stores an associated symbolic name in the internal map.
- `DelState(index)`
  This searches for the index in the map and deletes its entry. If the index has no name associated nothing happens.
- `Name(index)`
  Returns the symbolic name for a state index.

The methods of the statesymboltable usually will be only called by `StateSet` objects that request a new state index or look up the symbolic name of a state index.

Generator Class

With the stated ADTs the `Generator` class that implements a software model of the 5-tuple automaton $G := (X, \Sigma, \delta, X_0, X_\mathrm{m})$ defined in Definition 2.1.2 is developed as shown in Figure 3.6. Note that the $\star$ at the `EventSymbolTable` and `StateSymbolTable`, which denotes a pointer to an object outside of the `Generator`. Within an event domain all `Generator` objects point to the same `EventSymbolTable` and `StateSymbolTable` object.

While in algorithms, all operations are stated by directly using the objects contained in a `Generator`, a programming language implementation of the `Generator` class usually encapsulates all inner objects. The practical part of this thesis follows these programming guidelines.

Generator
StateSet X
EventSet Σ
TransSet δ
StateSet X_0
StateSet X_{m}
EventSymbolTable* e
StateSymbolTable* s

Figure 3.6: `Generator` class

Chapter 4

Algorithms for Regular Languages and Finite Automata

In this chapter, algorithms for regular languages and finite automata are implemented with the `Generator` class and the abstract data types introduced above.

The notation of algorithms is as follows:

- Function parameters are denoted G^{index} instead of G_{index} while both means the same.
- Every function parameter called by reference[1], is denoted by a $_{\&}$ after the parameter name.
- The methods of ADTs are stated in `Typewriter` font. The first parameter of each method is always the object itself.
- Multiple statements within one line are separated by a semicolon.
- The subscript of a generator variable corresponds to the subscript of its members. E.g. X_{sub}, Σ_{sub}, δ_{sub}, $X_{0,sub}$ and $X_{\mathrm{m},sub}$ correspond to the generator G_{sub}.

[1]This means that the function operates on the object itself and not on a copy of the object.

4.1 Language Operations

4.1.1 Parallel Composition

The parallel composition is a language operation as well as an automaton method. It was already defined in Definition 3.2.1. The algorithm implemented for the `Generator` class is stated as follows.

Algorithm 4.1 (Parallel Composition)**.** The parallel composition of two given automaton $G_1 = (X_1, \Sigma_1, \delta_1, X_{0,1}, X_{\mathrm{m},1})$ and $G_2 = (X_2, \Sigma_2, \delta_2, X_{0,2}, X_{\mathrm{m},2})$ constructs the automaton $G_{1\|2} := \left(X_{1\|2}, \Sigma_1 \cup \Sigma_2, \delta_{1\|2}, X_{0,1} \times X_{0,2}, X_{\mathrm{m},1\|2}\right)$ such that $L\left(G_{1\|2}\right) = L\left(G_1\right) \| L\left(G_2\right)$ and $L_{\mathrm{m}}\left(G_{1\|2}\right) = L_{\mathrm{m}}\left(G_1\right) \| L_{\mathrm{m}}\left(G_2\right)$. The synchronous product of two languages is defined in [Won04, CL99].

The following algorithm is a proposal by T. .

function PARALLEL(`Generator` $G^1_{\&}$, `Generator` $G^2_{\&}$, `Generator` $G^{1\|2}_{\&}$, `map<pair <Idx,Idx>,Idx>` $RCMap_{\&}$)
    / Local variables */*
    `Stack<Pair<Idx,Idx>>` $X_{waiting}$ */* Stack of state index pairs */*
    `Pair<Idx,Idx>` $p_{current}$, p_{new} */* State pairs */*
    `Idx` x_{tmp} */* Temporary state index */*
    `EventSet` $\Sigma_{shared} := \Sigma_1 \cap \Sigma_2$ */* The shared alphabet */*
    / Initialization */*
    for all $(x_1, x_2) \in X_{0,1} \times X_{0,2}$ **do**
        `Push(`$X_{waiting}$`,` (x_1, x_2)`)`
    end for
    / Start */*
    while $X_{waiting} \neq \emptyset$ **do**
        $p_{current} :=$ `Pop(`$X_{waiting}$`)`
        / Iteration over all transitions at current state in G^1 */*
        for all $t_1 \in$ `Transitions(`δ_1`,` $p_{current}$`.first)` **do**
            / If the event of the current transition is not shared */*
            if t_1`.x1` $\notin \Sigma_{shared}$ **then**
                $p_{new} := (t_1$`.x2`$, p_{current}$`.second`$)$
                if $p_{new} \notin$ `Keys(`$RCMap$`)` **then**
                    `push(`$X_{waiting}$`,` p_{new}`)`
                    $x_{tmp} :=$ `InsState(`$X_{1\|2}$`)`

```
                $RCMap\,[p_{new}] := x_{tmp}$
            **else**
                $x_{tmp} := RCMap\,[p_{new}]$
            **end if**
            SetTransition($\delta_{1\|2}$, $RCMap\,[p_{current}]$, $t_1$.ev, $x_{tmp}$)
            /* If the event of the current transition is shared */
        **else**
            **for all** $t_2 \in$ Transitions($\delta_2$, $t_1$.ev) **do**
                $p_{new} := (t_1.\mathtt{x2}, t_2.\mathtt{x2})$
                **if** $p_{new} \notin$ Keys($RCMap$) **then**
                    push($X_{waiting}$, $p_{new}$)
                    $x_{tmp} :=$ InsState($X_{1\|2}$)
                    $RCMap\,[p_{new}] := x_{tmp}$
                **else**
                    $x_{tmp} := RCMap\,[p_{new}]$
                **end if**
                SetTransition($\delta_{1\|2}$, $RCMap\,[p_{current}]$, $t_1$.ev, $x_{tmp}$)
            **end for**
        **end if**
    **end for**
    /* Iteration over all transitions at current state in $G^2$ */
    **for all** $t_2 \in$ Transitions($\delta_2$, $p_{current}$.second) **do**
        /* If the event of the current transition is unshared */
        **if** $t_2$.ev $\notin \Sigma_{shared}$ **then**
            $p_{new} := (p_{current}.\mathtt{first}, t_2.\mathtt{x2})$
            **if** $p_{new} \notin$ Keys($RCMap$) **then**
                Push($X_{waiting}$, $p_{new}$)
                $x_{tmp} :=$ InsState($X_{res}$)
                $RCMap\,[p_{new}] := x_{tmp}$
            **else**
                $x_{tmp} := RCMap\,[p_{new}]$
            **end if**
        **end if**
    **end for**
    /* Mark states in $G^{1\|2}$ */
    **for all** $x_1 \in X_{\mathrm{m},1}$ **do**
        **for all** $x_2 \in X_{\mathrm{m},2}$ **do**
            **if** $(x_1, x_2) \in$ Keys($RCMap$) **then**
                $X_{\mathrm{m},1\|2} := X_{\mathrm{m},1\|2} \cup \{RCMap\,[(x_1, x_2)]\}$
```

            end if
        end for
    end for
end while
end function

Note: The map $RCMap$, given as a reference parameter in the function, can be used for constructing state names of $G_{1\|2}$ on demand as proposed in Section 3.3.1. If this is not required a convenience PARALLEL function can be created that hides the parameter.

4.1.2 Projection

The natural projection is defined in Definition 3.2.2. The algorithm resembles the abstract algorithm provided in Chapter 3.

Algorithm 4.2 (Projection)**.** Given a finite automaton $G = (X, \Sigma, \delta, X_0, X_{\mathrm{m}})$ and a projection alphabet $\Sigma_0 \subseteq \Sigma$. The function PROJECT applies the projection $p_0 : \Sigma^* \to \Sigma_0^*$ directly on the automaton G. In addition, a helper function COMPACCREACH (compute accessible reach) is given that computes the states that are reachable by invisible transitions (meaning t.`ev` $\notin \Sigma_{proj}$) from a given state.

Let G_0 be the resulting automaton of a function call PROJECT(G, Σ_0). Then $L(G_0) = p_0(L(G))$ and $L_{\mathrm{m}}(G_0) = p_0(L_{\mathrm{m}}(G))$.

function PROJECT(`Generator` $G_\&$, `EventSet` $\Sigma_\&^0$)
    / local variables */*
    `StateSet` X_{reach} */* reachable states */*
    `Stack<Idx>` $X_{waiting}$*/* waiting list */*
    `StateSet` X_{done} */* processed states */*
    `Idx` $x_{current}$ */* current state */*
    / start */*
    for all $x \in X_0$ **do**
        `Push(`$X_{waiting}$`,` x`)`
    end for
    while $X_{waiting} \neq \emptyset$ **do**
        $x_{current} :=$ `Pop(`$X_{waiting}$`)`
        $X_{done} := X_{done} \cup \{x_{current}\}$

/* *compute accessible reach local paths*[1] */
COMPACCREACH(G, Σ^0, X_{reach}, $x_{current}$)
for all $t \in$ `Transitions(`δ`,`$x_{current}$`)` **do**
  if t`.ev` $\notin \Sigma^0$ **then**
    `DelTransition(`δ`,`t`)`
  end if
end for
for all $x_{reach} \in X_{reach}$ **do**
  for all $t \in$ `Transitions(`δ`,`x_{reach}`)` **do**
    if t`.ev` $\in \Sigma^0$ **then**
      `SetTransition(`$x_{current}$`,`t`.ev,`t`.x2)`
      if t`.x2` $\notin X_{done}$ **then**
        `Push(`$X_{waiting}$`,`t`.x2)`
      end if
    end if
  end for
  /* *if locally reachable state is marked, mark* $x_{current}$ */
  if $x_{reach} \in X_{\mathrm{m}}$ **then**
    $X_{\mathrm{m}} := X_{\mathrm{m}} \cup \{x_{current}\}$
  end if
end for
end while
end function

The COMPACCREACH function is defined as follows.

function COMPACCREACH(`Generator` $G_{\&}$, `EventSet` $\Sigma^0_{\&}$, `StateSet` $X^{reach}_{\&}$, `Idx` x_{start})
  /* *local variables* */
  `Stack<Idx>` $X_{waiting}$
  `Idx` $x_{current}$
  /* *start* */
  `Push(`$X_{waiting}$`,`x_{start}`)`
  while $X_{waiting} \neq \emptyset$ **do**
    $x_{current}$:= `Pop(`$X_{waiting}$`)`
    for all $t \in$ `Transitions(`δ`,`$x_{current}$`)` **do**
      if t`.ev` $\notin \Sigma^0 \wedge t$`.x2` $\notin X^{reach}$ **then**
        `Push(`$X_{waiting}$`,`t`.x2)`

[1] A local path is a sequence of low-level events in $\Sigma - \Sigma^0$.

$X^{reach} := X^{reach} \cup \{t.\mathtt{x2}\}$
end if
end for
end while
end function

4.1.3 Inverse Projection

The *inverse projection* is defined as follows.

Definition 4.1.1 (Inverse Projection [Won04]). For an alphabet $\Sigma_0 \subseteq \Sigma$ the *inverse projection* $(p_0)^{-1} : \Sigma_0^* \to 2^{\Sigma^*}$ is

$$(p_0)^{-1}(t) := \{s \in \Sigma^* \mid p_0(s) = t\}$$

for $t \in \Sigma_0^*$.

The inverse projection of a language $L_0 \in \Sigma_0^*$ is

$$(p_0)^{-1}(L_0) := \{s \in \Sigma^* \mid \exists t \in L_0 \text{ s.t. } p_0(s) = t\}.$$

Algorithm 4.3 (Inverse Projection). Given a finite automaton $G_0 = (X_0, \Sigma_0, \delta, X_{0,0}, X_{\mathrm{m},0})$ and a alphabet $\Sigma_0 \subseteq \Sigma$. The generated language of the automaton is extended to alphabet Σ directly in the automaton by the function INVPROJECT. Let G be the resulting automaton of a function call INVPROJECT(G, Σ). Then $L(G) = (p_0)^{-1}(L(G_0))$ and $L_\mathrm{m}(G) = (p_0)^{-1}(L_\mathrm{m}(G_0))$.

function INVPROJECT(`Generator` $G^0_\&$, `EventSet` $\Sigma_\&$)
    / local variables */*
    `EventSet` $\Sigma_{new} := \Sigma \setminus \Sigma^0$
    / start */*
    for all $x \in X_0$ **do**
        for all $\sigma \in \Sigma_{new}$ **do**
            `SetTransition(`δ_0`,`x`,`σ`,`x`)`
        end for
    end for
end function

4.2 Automata Operations

4.2.1 Accessible

The *accessible* operation removes all states, that cannot be reached by a transition path starting at an initial state from an automaton. Formally [CL99],

- $\text{ACCESSIBLE}(G) := (X_{acc}, \Sigma, \delta_{acc}, X_0, X_{\mathrm{m},acc})$, where
- $X_{acc} := \{x \in X \mid \exists s \in \Sigma^*, \exists x_0 \in X_0 \text{ s.t. } \delta(x_0, s) = x\}$,
- $X_{\mathrm{m},acc} := X_\mathrm{m} \cap X_{acc}$,
- $\delta_{acc} := \delta|_{X_{acc} \times \Sigma \to X_{acc}}$.

Algorithm 4.4 (Accessible Automaton)**.** Given an automaton $G = (X, \Sigma, \delta, X_0, X_\mathrm{m})$. The ACCESSIBLE function returns `True` if the automaton is nonempty and no states had to be removed, otherwise `False`. Internally the set of reachable states is computed by the function ACCESSIBLESET while ACCESSIBLE only removes the set difference and handles the return value. ACCESSIBLESET uses the recursive function CHECKACCESSIBLE to compute the set of reachable states. The ACCESSIBLESET function is also used in other algorithms. The accessible operation has no effect on $L(G)$ and $L_\mathrm{m}(G)$.

function ACCESSIBLE(`Generator` $G_\&$)
  / local variables */*
  `StateSet` $X_{not\_accessible}$
  / start */*
  if $X = \emptyset$ **then**
    return `False`
  end if
  $X_{not\_accessible} := X \setminus$ ACCESSIBLESET(G)
  if $X_{not\_accessible} \neq \emptyset$ **then**
    for all $x \in X_{not\_accessible}$ **do**
      $X := X \setminus \{x\}$
      for all $t \in$ `Transitions(`δ`,`x`)` **do**
        `DelTransition(`δ`,`t`)`
      end for
      $X_0 := X_0 \setminus \{x\}$

$X_{\mathrm{m}} := X_{\mathrm{m}} \setminus \{x\}$
end for
return `False`
else
return `True`
end if
end function

The ACCESSIBLESET function returns a `StateSet` containing the reachable states in an automaton.

function ACCESSIBLESET(`Generator` $G_{\&}$)
/ local variables */*
`StateSet` X_{acc}
/ start */*
for all $x \in X_0$ **do**
CHECKACCESSIBLE(G, X_{acc}, x)
end for
return X_{acc}
end function

function CHECKACCESSIBLE(`Generator` $G_{\&}$, `StateSet` $X^{acc}_{\&}$, `Idx` x^{start})
if $x^{start} \notin X^{acc}$ **then**
$X^{acc} := X^{acc} \cup \{x^{start}\}$
for all $t \in$ `Transitions(`δ`,` x^{start}`)` **do**
CHECKACCESSIBLE(G, X^{acc}, t`.x2`)
end for
end if
end function

4.2.2 Coaccessible

Corresponding to the *accessible* operation the *coaccessible* operation removes all states from an automaton, that do not have a transition path to a marked state. Formally [CL99],

- COACCESSIBLE(G) $:= (X_{coacc}, \Sigma, \delta_{coacc}, X_{0,coacc}, X_{\mathrm{m}})$, where

- $X_{coacc} := \{x \in X \mid \exists s \in \Sigma^*, \exists x_{\mathrm{m}} \in X_{\mathrm{m}} \text{ s.t. } \delta(x_{\mathrm{m}}, s) \in X_{\mathrm{m}}\}$,

- $X_{0,coacc} := \{x_0 \in X_0 \mid x_0 \in X_{coacc}\}$,

- $\delta_{coacc} := \delta|_{X_{coacc} \times \Sigma \to X_{coacc}}$.

Algorithm 4.5 (Coaccessible Automaton)**.** Given an automaton $G = (X, \Sigma, \delta, X_0, X_{\mathrm{m}})$. At first the set of coaccessible states is computed by the function COACCESSIBLESET, that itself calls the recursive function CHECKCOACCESSIBLE. COACCESSIBLE returns `True` if all states in the automaton are coaccessible and `False` if not. While $L(\text{COACCESSIBLE}(G)) = \overline{L_{\mathrm{m}}(G)}$ the operation has no effect on $L_{\mathrm{m}}(G)$.

```
function COACCESSIBLE(Generator G_&)
    /* local variables */
    StateSet X_not_coaccessible
    /* start */
    if X = ∅ then
        return False
    end if
    X_not_coaccessible := X \ COACCESSIBLESET(G)
    if X_not_coaccessible ≠ ∅ then
        for all x ∈ X_not_coaccessible do
            X := X \ {x}
            for all t ∈ Transitions(δ, x) do
                DelTransition(δ, t)
            end for
            X_0 := X_0 \ {x}
            X_m := X_m \ {x}
        end for
        return False
    else
        return True
    end if
end function
```

The COACCESSIBLESET function returns a `StateSet` containing the states that can reach a marked state via a transition path.

function COACCESSIBLESET(Generator $G_{\&}$)
  /* *local variables* */
  StateSet X_{coacc}
  TransSet<Sortx2evx1> $\delta_r = \delta$ /* *build x2,ev,x1 sorted set of transitions* */
  /* *start* */
  for all $x \in X_{\mathrm{m}}$ **do**
    CHECKCOACCESSIBLE(G, δ_r, X_{coacc}, x)
  end for
  return X_{coacc}
end function

function CHECKCOACCESSIBLE(Generator $G_{\&}$, TransSet<Sortx2evx1> $\delta^r_{\&}$, StateSet $X^{coacc}_{\&}$, Idx x^{start})
  if $x^{start} \notin X^{coacc}$ **then**
    $X^{coacc} := X^{coacc} \cup \{x^{start}\}$
    for all $t \in$ Transitions(δ^r, x^{start}) **do**
      CHECKACCESSIBLE(G, X^{coacc}, t.x1)
    end for
  end if
end function

4.2.3 Trim

Trim is a convenience method for computing the accessible and coaccessible part of an automaton. An automaton is *trim* if all states in the automaton are accessible and coaccessible. Formally, the trim operation is defined as

- TRIM := ACCESSIBLE(G) $\wedge$ COACCESSIBLE(G) .

Algorithm 4.6 (Trim Automaton)**.** Given an automaton $G = (X, \Sigma, \delta, X_0, X_{\mathrm{m}})$. The TRIM function removes states by calling the ACCESSIBLE and the COACCESSIBLE function. If no state is removed `True` is returned, else the result is `False`.

```
function TRIM(Generator G_&)
    /* local variables */
    Bool is_accessible
    Bool is_coaccessible
    /* start */
    is_accessible := ACCESSIBLE(G)
    is_coaccessible := COACCESSIBLE(G)
    if is_accessible = True ∧ is_coaccessible = True then
        return True
    else
        return False
    end if
end function
```

4.2.4 Determine

The *determine* method computes the deterministic automaton for a nondeterministic one by subset construction, according to Definition 3.2.3.

The implementation of the method is straightforward. In Section 3.2.3 two main areas of computation in subset construction have been pointed out. The set inclusion tests in the set of subsets by a hash table and supporting set signatures is directly implemented in the stated algorithm. The optimization by multiway merge according to [Les95] to compute the deterministic successor states of a subset is illustrated at the end of the section.

Algorithm 4.7 (Deterministic Automaton)**.** For a given nondeterministic automaton $G_{nd} = (X_{nd}, \Sigma, \delta_{nd}, X_{0,nd}, X_{\mathrm{m},nd})$ the DETERMINE function computes the deterministic automaton $G_d = (X_d, \Sigma, \delta_d, x_{0,d}, X_{\mathrm{m},d})$ such that $L(G_d) = L(G_{nd})$ and $L_\mathrm{m}(G_d) = L_\mathrm{m}(G_{nd})$. In addition the function DETERMINISTIC is provided, that checks a given automaton $G = (X, \Sigma, \delta, X_0, X_\mathrm{m})$ for nondeterminism.

The DETERMINE function is implemented by two additional call-by-reference parameter $PowerStates$ and $DetStates$ that hold information for the implementation of the algorithms in [Sch05b]. This will be discussed in Chapter 6.

```
function DETERMINE(Generator $G^{nd}_{\&}$, Generator $G^{d}_{\&}$, Vector <StateSet>
  $PowerStates_{\&}$, Vector<Idx> $DetStates_{\&}$ )
    /* local variables */
    StateSet $X_{new}$ /* holds newly constructed subsets */y
    Idx $x_{new}$
    Map<int,Vector<int> > $HashMap$ /* implements a hash table */
    int $sig$ /* set signature */
    int $i$, $j$ /* vector indices */
    Bool $new$
    /* start */
    $\Sigma_d := \Sigma_{nd}$
    /* initialization */
    $x_{new}$ := InsState($X_d$)
    for all $x \in X_{nd,0}$ do
        $X_{new} := X_{new} \cup \{x\}$
    end for
    $sig$ := Signature($X_{new}$)
    PushBack($PowerStates$, $X_{new}$);PushBack($DetStates$, $x_{new}$)
    /* creates new map entry and directly adds vector element */
    PushBack($HashMap[sig]$, Lastindex($PowerStates$))
    /* iteration over all vector entries */
    for $i := 0$; $i \leq$ LastIndex($PowerStates$); $i := i + 1$ do
        $X_{new} := \emptyset$
        for all $\sigma \in \Sigma_{nd}$ do
            for all $x \in PowerStates[i]$ do
                for all $t \in$ Transitions($\delta_{nd}$, $x$, $\sigma$) do
                    $X_{new} := X_{new} \cup t$.x2
                end for
            end for
            $sig$ := Signature($X_{new}$)
            $new$ := True
            if $sig \in$ Keys($HashMap$) then
                /* iteration over vector of $PowerStates$ vector indices */
                for all $j \in HashMap[sig]$ do
                    if $X_{new} = PowerStates[j]$ then
                        $new$ := False
                        break; /* end for all loop */
                    end if
                end for
```

        end if
        if new = `True` **then**
            x_{new} := `InsState`(X_d)
            `PushBack`($PowerStates$, X_{new});`PushBack`($DetStates$, x_{new})
            `PushBack`($HashMap[sig]$, `Lastindex`($PowerStates$))
            for all $x \in X_{new}$ **do**
                if $x \in X_{\mathrm{m},nd}$ **then**
                    $X_{\mathrm{m},d} := X_{\mathrm{m},d} \cup x_{new}$
                    break; /* *end for all loop* */
                end if
            end for
        end if
        /* *introduce transition in* G^d */
        `SetTransition`(δ_d, $DetStates[i]$, σ, x_{new})
    end for
end for
end function

The function DETERMINISTIC only tests if an automaton is deterministic.

function DETERMINISTIC(`Generator` $G_{\&}$)
    /* *local variables* */
    `Idx` $x_{last} := 0$
    `Idx` $\sigma_{last} := 0$
    /* *start* */
    if `Size`(X_0) > 1 **then**
        return `False`
    else
        for all $t \in \delta$ **do**
            if $x_{last} = t$.`x1` $\wedge$ $\sigma_{last} = t$.`ev` **then**
                return `False`
            end if
            $x_{last} := t$.`x1`; $\sigma_{last} := t$.`ev`
        end for
    end if
    return `True`
end function

Optimization by Multiway Merge

As already noted the algorithm can be optimized by a multiway merge to compute the successor states of a deterministic state. In line 23 an iteration over all events is done. For each nondeterministic state in the subset, the transitions from the state driven by the current event have to be computed in the two nested loops in the lines 24 to 28. This causes a computational expensive search in the set of transitions in an inner loop.

The search can be avoided by placing an iterator for each state in the subset at the first transition of that state in the set of transitions. Then a multiway merge of the transitions at which the iterators currently point to is applied. This results in two vectors, a vector of events with event indices in ascending order and a vector of corresponding successor states. The vector of events is partitioned, where each corresponding partition element in the successor state vector builds a new deterministic state that will be processed according to lines 29 to 52. As partitioning the resulting vectors is trivial this is not included. The multiway merge algorithm is given as follows.

`Idx` σ_{last}
`Idx` x_{last}
`Vector<Idx>` $EventVector$
`Vector<Idx>` $StateVector$
/ vector of `TransSet` iterators */*
`Vector<TransSet::Iterator>` $Iterators$
/ place an iterator at each state of the subset in δ */*
for all $x \in PowerStates\,[i]$ **do**
    `Iterator` $it :=$ `Find(`δ`,`x`)`
    if $it \neq$ `End(`δ`)` **then**
        `PushBack(`$Iterators$`,`it`)`
    end if
end for
while $Iterators \neq \emptyset$ **do**
    find $it \in Iterators$ such that it`.ev` is the smallest
    $x_{last} :=$ it`.x1`
    $\sigma_{last} :=$ it`.ev`
    loop
        if $it =$ `End(`δ`)` $\vee$ it`.x1` $\neq x_{last}$ **then**
            $Iterators := Iterators \setminus \{it\}$

```
        else if it.ev = σ_last then
            PushBack(EventVector, it.ev)
            PushBack(StateVector, it.x2)
            it++ /* set iterator to next transition */
        else
            break; /* exit loop */
        end if
    end loop
end while
```

The algorithm initialization for a subset is shown in Figure 4.1. It is assumed the lexical ordering of the events resembles the ordering of their indices. At first the transition $5 - a - 9$ is read at $it\,[0]$. Then the next iterator with the lowest event, $it\,[1]$ is chosen. After successively reading two transitions $8 - a - 2$ and $8 - a - 8$, $it\,[0]$ is chosen again. Then the transitions $5-b-8$ and $5-b-15$ are read. After reading $8-b-13$ at $it\,[1]$ this iterator is removed. The algorithm continues with $11 - b - 5$, $5 - c - 3$ and the removal of iterator $it\,[0]$. At last the rest of the transitions at $it\,[2]$ is read, $11-c-6$ and $11-c-10$.

Iterator $it[0]$ $\longrightarrow$	$5 - a - 9$
	$5 - b - 8$
	$5 - b - 15$
	$5 - c - 3$
Iterator $it[1]$ $\longrightarrow$	$8 - a - 2$
	$8 - a - 8$
	$8 - b - 13$
Iterator $it[2]$ $\longrightarrow$	$11 - b - 5$
	$11 - c - 6$
	$11 - c - 10$

Figure 4.1: Initialization of multiway merge of transitions from a subset

4.2.5 State Space Minimization

The state space minimization method constructs a *canonical recognizer* for a deterministic finite automaton. As already stated in Section 2.1, the resulting set of states of the minimized automaton is unique up to an isomorphism [HU79].

Algorithm 4.8 (State Space Minimization)**.** For a given deterministic automaton $G = (X, \Sigma, \delta, X_0, X_\mathrm{m})$ the StateMin function computes the canonical recognizer $G_{min} = (X_{min}, \Sigma, \delta_{min}, x_{0,min}, X_{\mathrm{m},min})$, such that $L(G_{min}) = L(G)$ and $L_\mathrm{m}(G_{min}) = L_\mathrm{m}(G)$. This function directly implements the algorithm provided in [Hop71, AHU+83] with a complexity of $\mathcal{O}(|\Sigma| \cdot |X| \cdot \log |X|)$ to compute the canonical recognizer.

function StateMin(`Generator` $G_\&$, `Generator` $G_\&^{min}$)
 / local variables; naming corresponds to [AHU+83] */*
 `Vector<StateSet>` B */* blocks */*
 `int` i, j
 `Set<int>` $Waiting$
 `TransSet<Sortevx2x1>` $\delta_r := \delta$ */* ev,x2,x1 sorted transition relation */*
 `StateSet` $X_{current}$, $X_{inverse}$, $X_\cap$, X_-
 `Map<Idx, int>` $IndexMap$ */* maps blocks to new states */*
 `Idx` x_{new}
 / start */*
 Accessible(G) */* ensure G contains only accessible states */*
 if `Size(`X`)` ≤ 1 **then**
  G^{min} := `Copy(`G`)`
  return
 end if
 $\Sigma_{min} := \Sigma$
 / set up blocks */*
 $i := 0$
 if `Size(`X`)` $-$ `Size(`X_m`)` > 0 **then**
  `PushBack(`B`,` $X \setminus X_\mathrm{m}$`)`
  $Waiting := Waiting \cup \{i\}$
  $i := i + 1$
 end if
 `PushBack(`B`,` X_m`)`
 $Waiting := Waiting \cup \{i\}$
 $i := i + 1$
 while $Waiting \neq \emptyset$ **do**
  pick $i \in Waiting$; $Waiting := Waiting \setminus \{i\}$
  $x_{current} := B[i]$
  / compute $X_{inverse} := f^{-1}(B[i])$ for each $\sigma \in \Sigma$ */*
  for all $\sigma \in \Sigma$ **do**
   $X_{inverse} := \emptyset$
   for all $x \in X_{current}$ **do**

for all $t \in$ `TransitionsByevx2`[1]`(`δ_r`,`σ`,`x`)` **do**
$X_{inverse} := X_{inverse} \cup \{$`t.x1`$\}$
end for
end for
if $X_{inverse} \neq \emptyset$ **then**
for $j = 0; j <$ `Size(`B`)`, $j := j + 1$ **do**
$X_\cap := B[j] \cap X_{inverse}$
$X_- := B[j] \setminus X_\cap$
if $X_\cap = \emptyset \vee X_- = \emptyset$ **then**
continue /* *next for iteration* */
end if
`PushBack(`B`,`$X_\cap$`)`
$B[j] := X_-$/* *replace old block* */
if $j \in Waiting$ **then**/* *mark both waiting* */
$Waiting := Waiting \cup \{$`LastIndex(`$B$`)`$\}$
else/* *mark only smaller as waiting* */
if `Size(`$X_\cap$`)` $>$ `Size(`X_-`)` **then**
$Waiting := Waiting \cup \{$`LastIndex(`$B$`)`$\}$
else
$Waiting := Waiting \cup \{j\}$
end if
end if
end for
end if
end for
end while
/* *build minimized generator* */
for $i := 0, i \leq$ `LastIndex(`B`)`, $i := i + 1$ **do**
$x_{new} :=$ `InsState(`X_{min}`)`
for all $x \in B[i]$ **do**
$IndexMap[x] := i$
if $x \in X_0$ **then**
$X_{0,min} := X_{0,min} \cup \{x_{new}\}$
end if
if $x \in X_{\mathrm{m},min}$ **then**
$X_{\mathrm{m},min} := X_{\mathrm{m},min} \cup \{x_{new}\}$
end if

[1] `TransitionsByevx2` is defined according to Section 3.3.3 as a object method that returns a `TransSet<Sortevx2x1>` iterator.

```
        end for
        for all t ∈ δ do
            SetTransitions(δ_min, IndexMap[t.x1], t.ev, IndexMap[t.x2])
        end for
    end for
end function
```

Chapter 5

Automaton Extension and Algorithms for Supervisory Control

In this chapter, the automaton model specified in Chapter 3 is extended for the requirements of supervisory control by the introduction of uncontrollable events in Section 5.1. Then the extended model is applied to the `Generator` automaton class in Section 5.2 which results in the `cGenerator` class. At last the algorithm for computing the supremal controllable and nonblocking sublanguage is stated, with an additional section about implementing a function that tests controllability.

5.1 Introduction of Events Properties

When modelling DES in RW control theory the set of events Σ in a DES is divided into two disjoint sets, the set of controllable events Σ_c and the set of uncontrollable events Σ_{uc}. This was already shown in Section 2.2. It is assumed that the division is consistent for all automata. Therefore an event cannot be controllable in one automaton and uncontrollable in another automaton.

This is an important result for modelling DES in software as it means the controllability properties of events can be stored globally for all events in an system. As other theory approaches may introduce further event properties that are consistent within a system, a general approach is required to hold a set of event properties for all events in an event domain.

Therefore an entity that holds the controllability property of events and is extensible to store further event properties is proposed as follows:

- Each property is stored as a set of binary values. Such a set is usually called a *bitset*. The required number of bits to store a property is one for binary properties like controllability, but may be bigger if a property can take more than two values. To efficiently access the bitset, it is implemented by the natural integer type of the computer architecture. Then a 32bit computer architecture can hold up to 32 event properties. If more bit values are required the double long integer can be used or any other data type that allows bit by bit manipulation.

- The mapping of events to their respective bitset is implemented by a `Map<Idx, int>`[1] where the key holds the event index and the value stores the bitset.

Bits in a bitset are enabled by combining the current value of the bitset with an `int` that has all bits, that are to be enabled by the logical `AND` operation set to `1` . In contrast, bits are disabled by combining the current bitset value with an `int` having all bits set to `0` that are to be disabled by the logical `OR` operation. To retrieve a set of bits of the bitset, it is combined with an `int` where all bits that are to be retrieved are set to `1` by the `AND` operation. Bits which are not requested are set to `0` while the requested bits still have their value `0` or `1`.

As an example, the controllability property may be represented by the first bit in an `int`. An event is modelled controllable if the controllable bit is set to `1` and uncontrollable if `0`. Then defining an event `e` to be controllable requires a `e OR 0x0000001` operation and defining it to be uncontrollable a `e AND 0xFFFFFFF0` operation in the octal integer notation. The controllability property can be retrieved by a `e AND 0x0000001` operation, which blends out all other bits. Determining the value then simply is done by comparing the retrieved value with `0x00000001` or `0x00000000`.

[1]Note that this can also be implemented by a `Vector<int>` if the `Vector` index type is compatible with `Idx` for performance optimizations. However the more general `Map` solution is proposed here, as a `Vector` always requires to allocate as much element blocks in memory as required to access the largest `Idx` by a vector index . In contrast a `Map` can also hold only bitsets for a subset of the events.

The entity that holds the controllability property of all events within an event domain is called an *eventflagtable*, representing the flag character of a bit. The implementation as an ADT is presented in the next section.

5.2 Controllable Generator Class Implementation

The `Generator` class introduced in Section 3.3.3 has no implementation of controllable events. In this section, the class is extended to a controllable `Generator` class that suits the requirements of supervisory control. An *eventflagtable* entity located outside of the data structure of an automaton was proposed to provide the controllability property for events. The implementation of this entity as the ADT `FlagTable` will be shown at first, followed by the extended automaton class `cGenerator`.

FlagTable

The `FlagTable` ADT is created by a `FlagTable<TBitsetType>` statement and holds a `Map<Idx,TBitsetType>` object internally. This more general implementation of the specified *eventflagtable* entity by a variable type for storing the bitset is well suited for different approaches in theory and can be used to assign attributes to events and states. For the RW control theory a `FlagTable<int>` is used as proposed before.

The `FlagTable` provides the following methods:

- `SetFlags(index,flags)`
  This method enables flags in the bitset by the logical `OR` operation. The first parameter contains the `Idx` identifier and the second holds a `TFlagType` having all bits that are to be enabled set to `1`.

- `ClrFlags(index,flags)`
  This method disables flags in the bitset by the logical `AND` operation on the inverse argument. The first parameter specifies the `Idx` identifier. The second holds a `TBitSetType`, where all flags that are to be disabled are set to `0` while all other have to be set to `1`.

- `GetFlags(index,flags,defaultflags)`
  The method retrieves bits of a bitset by the logical `AND` operation. Like in the other methods, the first parameter specifies the `Idx` type identifier. The second parameter holds the set of bits that are to be retrieved. Bits of interest have to be set to `1`, the remaining bits are set to `0`. The third parameter contains the default bitset value, that is returned if no bitset was stored in the map for the given identifier.

- `ExistsFlag(index)`
  This method will be called, to test if the internal map contains a bitset for a identifier. The `index` parameter specifies the `Idx` type identifier. `True` is returned, ff a bitset exists for the identifier, else `False`.

With the `FlagTable` ADT, the extension of the `Generator` class for supervisory control can now be implemented.

cGenerator

The object class that implements an automaton with controllable and uncontrollable events is called `cGenerator` (controllable Generator). In addition to the `Generator` class, it contains a pointer to a `FlagTable<int>` object outside the class that holds the controllability properties of events within an event domain. The class implements the same 5-tuple automaton $G := (X, \Sigma, \delta, X_0, X_\mathrm{m})$ as the `Generator` class with an additional division of Σ in the two disjoint sets Σ_c and Σ_uc that is modelled by the `FlagTable` class.

The `cGenerator` class provides methods to support the handling of controllable events like `SetControllable(index)`, `SetUncontrollable(index)` to set the controllability property of an event or `IsControllable(index)` to test the controllability property of an event. As the implementation of methods for controllable events is straightforward this is not covered here.

In algorithms a `cGenerator` object will be stated by the symbol G. The alphabet of uncontrollable events is implicitly included by the statement of a `cGenerator` object.

`cGenerator`
`StateSet` X
`EventSet` Σ
`TransSet` δ
`StateSet` X_0
`StateSet` X_m
`EventSymbolTable*` e
`StateSymbolTable*` s
`EventFlagTable*` f

Figure 5.1: `cGenerator` class

5.3 Nonblocking Supremal Controllable Sublanguage

The supremal controllable sublanguage as well as the nonblocking control theorem are defined in Definition 2.2.5 as the union of all controllable sublanguages that agree with a specification language $E \subseteq L(G)$. This algorithm computes the solution to the basic problem of supervisory control as stated in Section 2.2.

At first the algorithm computes the parallel composition of a plant G and a specification G_{spec} with $L(G_{spec}) = E$ which results in the supervisor automaton G_{sup}. Then in a loop two operations are executed until a fixpoint is reached:

- The supremal controllable sublanguage of G_{sup} with respect to G is computed. This requires identifying bad states, where strings followed in G and G_{sup} in parallel leave G_{sup} via an uncontrollable event. Then all transitions and states are removed from G_{sup} that can reach the bad states via uncontrollable events.

- The TRIM operation is executed on G_{sup}.

A fixpoint is reached, when no string of $L(G)$ leaves $L(G_{sup})$ via an uncontrollable event and G_{sup} is trim. This is also the case, if all states in G_{sup} have been removed and G_{sup} consists of the empty language $\emptyset$. Then computing a supervisor for the specification G_{spec} is not possible at all.

The algorithm is implemented by the function SUPCONNB that computes the nonblocking supremal controllable sublanguage and calls PARALLEL and then executes SUPCON and TRIM in a loop. SUPCON computes the supremal controllable sublanguage and calls the function REMOVEUCBACKWARDS (remove uncontrollable backwards) to remove states and transitions in G_{sup}. The PARALLEL function is stated in Section 4.1.1 and TRIM is introduced in Section 4.2.3. Remarks on optimizing the algorithm are given at the end of the section .

Algorithm 5.1 (Supremal Controllable Sublanguage)**.** Given a finite automaton of type `cGenerator` $G := (X, \Sigma, \delta, X_0, X_\mathrm{m})$ with a set of uncontrollable events $\Sigma_\mathrm{uc} \subseteq \Sigma$ and a specification automaton of type `Generator` $G_{spec} := (X_{spec}, \Sigma_{spec}, \delta_{spec}, X_{0,spec}, X_{\mathrm{m},spec})$. Then a supervisor automaton $G_{sup} := (X_{sup}, \Sigma_{sup}, \delta_{sup}, X_{0,sup}, X_{\mathrm{m},sup})$ is computed that either solves the basic supervisory control problem or contains the empty language $\emptyset$. A reference parameter `Map<Idx,Pair<Idx,Idx>>` $RCMap$ is used to hold the mapping of states $x_{sup} \in X_{sup}$ to combined states $(x, x_{spec}) \in X \times X_{spec}$.

Require: $\Sigma = \Sigma_{spec}$

function SUPCONNB(`cGenerator` $G_\&$, `Generator` $G_\&^{spec}$, `Map<Idx, Pair<Idx, Idx>>` $RCMap_\&$, `cGenerator` $G_\&^{sup}$)
    / local variables */*
    `Bool` $is\_controllable$, is_trim
    / start */*
    PARALLEL(G, G^{spec}, G^{sup}, $RCMap$)
    repeat
        $is\_controllable :=$ SUPCON(G, G^{sup}, $RCMap$)
        $is\_trim :=$ TRIM(G^{sup})
    until $is\_controllable \wedge is\_trim$
end function

The SUPCON function follows strings in G and G_{sup} and calls REMOVEUCBACKWARDS if a bad state is found.

function SUPCON(`cGenerator` $G_\&$, `Generator` $G_\&^{sup}$, `Map<Idx, Pair<Idx,`

`Idx>>` $RCMap_{\&}$)

/ local variables */*

`Stack<Idx>` $X_{waiting}$, $X_{waiting,sup}$ */* waiting list */*

`StateSet` $X_{discovered}$ */* discovered states in X_{sup} */*

`StateSet`X_{bad} */* forbidden states in X_{sup} */*

`Idx` $x_{current}$, $x_{current,sup}$

`TransSet<Sortx2evx1>` $\delta_r := \delta$ */* x2,ev,x1 sorted set of transitions */*

/ start */*

/ initialize waiting list */*

for all $x \in X_0$ **do**

  for all $x_{sup} \in X_{0,sup}$ **do**

    `Push(`$X_{waiting}$`,` x`)`

    `Push(`$X_{waiting,sup}$`,` x_{sup}`)`

  end for

end for

while $X_{waiting} \neq \emptyset$ **do**

  $x_{current}$:= `Pop(`$X_{waiting}$`)`

  $x_{current,sup}$:= `Pop(`$X_{waiting,sup}$`)`

  / process all transitions at current state in G */*

  for all $t \in$ `Transitions(`δ`,` $x_{current}$`)` **do**

    if t`.ev` $\in \Sigma_{\text{uc}} \wedge t$`.ev` $\notin \Lambda\,(\delta_{sup}, x_{current,sup})$ **then**

      REMOVEUCBACKWARDS(G^{sup}, δ_r, X_{bad}, $x_{current,sup}$)

      break */* continue with next pair on waiting list */*

    else

      for all $t_{sup} \in$ `Transitions(`δ_{sup}`,` $x_{current,sup}$`,` t`.ev)` **do**

        / add successor states to waiting list if undiscovered */*

        if $x_{current,sup} \notin X_{discovered}$ **then**

          `Push(`$X_{waiting}$`,` t`.x2)`

          `Push(`$X_{waiting,sup}$`,` t_{sup}`.x2)`

        end if

        / if successor state is not forbidden add to backward transitions */*

        if t_{sup}`.x2` $\notin X_{bad}$ **then**

          `SetTransition(`δ_r`,` t_{sup}`)`

        else if t`.ev` $\in \Sigma_{\text{uc}}$ **then**

          REMOVEUCBACKWARDS(G^{sup}, δ_r, X_{bad}, $x_{current,sup}$)

          t := `End(`δ`,` $x_{current}$`)` */* exit outer loop on next iteration */*

          break; */* exit inner loop */*

        end if

      end for

end if
end for
$X_{discovered} := X_{discovered} \cup \{x_{current,sup}\}$
end while
if $X_{forbidden} = \emptyset$ **then**
return `True`
else
return `False`
end if
end function

The function REMOVEUCBACKWARDS removes all states and transitions backwards that could make G_{sup} reach a bad state, i.e. along uncontrollable events.

function REMOVEUCBACKWARDS(`cGenerator` $G_{\&}^{sup}$, `TransSet< Sortx2evx1>` $\delta_{\&}^{r}$, `StateSet` $X_{\&}^{bad}$, `Idx` x^{start})
/ local variables */*
`Stack<Idx>` $X_{waiting}$
`Idx` $x_{current}$
/ start */*
`Push(`$X_{waiting}$`,` x^{start}`)`
$X^{bad} := X^{bad} \cup \{x^{start}\}$
while $X_{waiting} \neq \emptyset$ **do**
$x_{current} :=$ `Pop(`$X_{waiting}$`)`
/ remove transitions containing uncontrollable events backwards */*
for all $t \in$ `TransitionsByx2(`δ^{r}`,` $x_{current}$`)` **do**
if t`.ev` $\in \Sigma_{\mathrm{uc}} \wedge t$`.x1` $\notin X^{bad}$ **then**
`Push(`$X_{waiting}$`,` t`.x1)`
$X^{bad} := X^{bad} \cup \{t$`.x1`$\}$
end if
end for
/ remove $x_{current}$ from G^{sup} */*
$X_{sup} := X_{sup} \setminus \{x_{current}\}$
for all $t \in$ `Transitions(`δ_{sup}`,` $x_{current}$`)` **do**
`DelTransition(`δ_{sup}`,` t`)`
end for
$X_{0,sup} := X_{0,sup} \setminus \{x_{current}\}$
$X_{\mathrm{m},sup} := X_{\mathrm{m},sup} \setminus \{x_{current}\}$
end while

end function

Optimization Notes

Two optimizations are possible for computing a supervisor by the SupConNB function:

- The Parallel function in SupConNB can be exchanged by a function that performs the parallel composition by not following transitions from bad states. Instead bad states and the states that can be directly reached by transitions from bad states are stored in a set of bad states as in the SupCon function.
- If both G and G_{spec} are known to be deterministic, a parallel composition function can take advantage of the transition ordering in δ and δ_{spec} by following transitions in both automata in parallel instead of using two nested loops, like in the Parallel function. This is possible because both automata operate on the same alphabet which is generally not the case in the Parallel function.

5.4 Controllability

The definition of *controllability* is given in Definition 2.2.3. The algorithm for a function Controllable that tests the controllability of a given specification G_{spec} with respect to an automaton G is very similar to the SupCon function. The only required change is, that no state may be deleted. Instead the Controllable function must return `False` when a state is found, where G leaves the specification automaton G_{spec} by an uncontrollable event and `True` if no such state is found.

Chapter 6

Algorithms for Nonblocking Hierarchical Control

In [Sch05b], an extension to RW theory for the hierarchical control of decentralized DES is provided. Different from the RW control theory, this approach takes advantage of the decentralized structure of a system and enables the control of large scale systems, e.g. manufacturing systems. The decentralized system components are modelled as finite automata and build the lowest level of a multi-level control hierarchy. Then interacting components are abstracted to their common behavior and merged on a higher level of the control hierarchy. The abstraction is done by the natural projection of the alphabet on the lower level to the set of common events, referred to as the high-level alphabet. The main objective of the approach is a reduction of the state space. Each component on a level of the hierarchy is controlled by an own local supervisor and a supervisor on the next hierarchical level. The supervisor on the highest level then controls the whole system.

Formally, the low-level system is an automaton G and the high-level automaton G^{hi} is computed using the projection $p^{\mathrm{hi}} : \Sigma^* \rightarrow \left(\Sigma^{\mathrm{hi}}\right)^*$ with $L\left(G^{\mathrm{hi}}\right) := p^{\mathrm{hi}}\left(L\left(G\right)\right)$ and $L_{\mathrm{m}}\left(G^{\mathrm{hi}}\right) := p^{\mathrm{hi}}\left(L_{\mathrm{hi}}\left(G\right)\right)$. The tuple $\left(G, G^{\mathrm{hi}}\right)$ is called a projected system.

For nonblocking control and hierarchical consistency of the modelled system several conditions have to be fulfilled. These are *marked string acceptance*, *locally nonblocking*, *liveness*, *marked string controllability* and *mutual controllability*. Abstract algorithms for verification of the conditions are also provided in [Sch05b].

In this chapter, the algorithms for verification of *marked string acceptance* and *locally nonblocking* are adopted to the automaton model implemented in this thesis. The algorithms for verifying *Liveness*, *marked string controllability* and *mutual controllability* are included in Appendix A.1.

The *locally nonblocking* algorithm is stated as an example, how to implement an additional data structure, to support the visualization of states failing the condition in a graphical environment. The other algorithms are stated in versions that only verify the respective condition, without providing such additional data structures. However, the algorithm implementations in the practical part of this thesis all provide data structures for implementing such a visualization of failed states.

In [Sch05a] an improvement to the algorithm for verification of the locally nonblocking condition is provided that is incorporated in the implementation. [Sch05a] also provides data structures for an implementation of the control hierarchy which is not covered in this thesis.

At first a convenience method for deterministic language projection is presented followed by the introduction of entry states.

Deterministic Projection and Entry States

In [Sch05b] an automaton on a lower level in the control hierarchy G is abstracted to an automaton on a higher level G^{hi} by a deterministic projection of Σ to Σ^{hi}. The deterministic projection is applied by successively calling the function PROJECT, introduced in Section 4.1.2, and DETERMINE, introduced in Section 4.2.4. Therefore a convenience function is stated as follows.

1: **function** DETPROJECT(`Generator` $G_{\&}$, `EventSet` Σ^{hi}, `Generator` $G^{\mathrm{hi}}_{\&}$, `Map<Idx, StateSet>` $EntryStateMap_{\&}$)
2:     */* local variables */*
3:     `Generator` G_{tmp}
4:     `Vector<StateSet>` $PowerStates$

`Vector<Idx>` $DetStates$
`int` i
/ start */*
$G_{tmp} := G$
`Project(`G_{tmp}`,`Σ^{hi}`)`
`Determine(`G_{tmp}`,`G^{hi}`,`$PowerStates$`,`$DetStates$`)`
for $i := 0; i <$ `Size(`$PowerStates$`);` $i := i + 1$ **do**
  $EntryStateMap\,[DetStates\,[i]] := PowerStates\,[i]$
end for
end function

Note that $L\left(G^{\mathrm{hi}}\right) = p^{\mathrm{hi}}\left(L\left(G\right)\right)$ and $L_{\mathrm{m}}\left(G^{\mathrm{hi}}\right) = p^{\mathrm{hi}}\left(L_{\mathrm{m}}\left(G\right)\right)$.

This convenience function provides the abstraction of an automaton G on a lower level of the control hierarchy to an automaton G^{hi} on a higher level by a deterministic projection of Σ to Σ^{hi}. A map, denoted $EntryStateMap$, from states in the high level automaton to the corresponding subsets of states in the low-level automaton is created as a by-product. These subsets are sets of *entry states* of the low-level automaton, which are required for the algorithmic verification of the conditions mentioned above. The formal definition of entry states is given as follows.

Definition 6.0.1 (Entry States [Sch05b])**.** Let $G = (X, \Sigma, \delta, X_0, X_{\mathrm{m}})$ be an automaton that is abstracted to an automaton $G^{\mathrm{hi}} = \left(X^{\mathrm{hi}}, \Sigma^{\mathrm{hi}}, \delta^{\mathrm{hi}}, X^{0,\mathrm{hi}}, X^{\mathrm{m,hi}}\right)$ by a projection $p^{\mathrm{hi}} : \Sigma^* \rightarrow \left(\Sigma^{\mathrm{hi}}\right)^*$. For each state $x^{\mathrm{hi}} \in X^{\mathrm{hi}}$ a set of low-level entry states $X_{\mathrm{en,x^{hi}}}$ is defined as

$$X_{\mathrm{en,x^{hi}}} := \left\{x \in X \mid x = \delta\left(x_0, s_{\mathrm{en}}\right) \text{ for } s_{\mathrm{en}} \in L_{\mathrm{en,s^{hi}}}\right\} \subseteq X$$

with $L_{\mathrm{en,s^{hi}}} := \left\{s \in L\left(G\right) \mid p^{\mathrm{hi}}\left(s\right) = s^{\mathrm{hi}} \wedge \nexists s' < s \text{ s.t. } p^{\mathrm{hi}}\left(s'\right) = s^{\mathrm{hi}}\right\} \subseteq \Sigma^*$ which is denoted the set of *entry strings* [dCCK02].

This means when following strings in the low-level automaton and the corresponding projected strings in the high-level automaton, the entry states in the low level automaton correspond to these states in the high level automaton, where the same sequences of high level events have been executed with a minimum length of the low level string.

6.1 Verification of Marked String Acceptance

We assume that a deterministic projected system with $\left(G, G^{\mathrm{hi}}\right)$ is given. Marked string acceptance guarantees, that all low-level strings $s \in L(G)$ which reach the low level entry states $X_{\mathrm{en,x^{hi}}} \in X$ of a marked high level state $x^{\mathrm{hi}} \in X^{\mathrm{hi}}$, pass a marked low level state before a high level event is generated. This condition ensures, that transitions executed in the low level automaton always reach a marked state when the high level automaton reaches a marked state, too.

The definition of marked string acceptance is given in [Sch05b].

The following algorithm corresponds to the abstract algorithm in [Sch05b]. However, the computation of entry states is not required, as this already is a result of the deterministic projection.

Algorithm 6.1 (Marked String Acceptance)**.** Given the projected system as stated above with an entry state map $EntryStateMap$ that is the result of the deterministic projection function DETPROJECT. The function MARKEDSTRINGACCEPTANCE returns `True` if marked string acceptance is guaranteed for the system and `False` if not. The helper function CHECKLOCALMSA checks the property for a single entry state.

function MARKEDSTRINGACCEPTANCE(`Generator` $G^{\mathrm{lo}}_{\&}$, `Generator` $G^{\mathrm{hi}}_{\&}$, `Map <Idx,StateSet>` $EntryStateMap_{\&}$)
    / local variables */*
    `StateSet` X_{done}
    `Bool` $result$
    / start */*
    for all $x_{\mathrm{m,hi}} \in X_{\mathrm{m,hi}}$ **do**
        for all $x_{entry} \in EntryStateMap\left[x_{\mathrm{m,hi}}\right]$ **do**
            $result :=$ `CheckLocalMSA(`G^{lo}, G^{hi}, X_{done}, x_{entry}`)`
            if $result =$ `False` **then**
                return `False`
            end if
        end for
    end for
    return `True`
end function

CHECKLOCALMSA returns `True` if marked string acceptance is guaranteed for the given entry state and `False` if not.

function CHECKLOCALMSA(`Generator` $G_{\&}^{\mathrm{lo}}$, `Generator` $G_{\&}^{\mathrm{hi}}$, `StateSet` $X_{\&}^{done}$, `Idx` x^{entry})
    / local variables */*
    `Stack<Idx>` $X_{waiting}$
    `Idx` $x_{current}$
    / start */*
    if $x^{entry} \notin X_{done}$ **then**
        `Push(`$X_{waiting}$`,` x^{entry}`)`
    end if
    while $X_{waiting} \neq \emptyset$ **do**
        $x_{current} :=$ `Pop(`$X_{waiting}$`)`
        if $x_{current} \notin X_{\mathrm{m,lo}}$ **then**
            for all $t_{lo} \in$ `Transitions(`δ_{lo}`,` $x_{current}$`)` **do**
                if t_{lo}`.ev` $\in \Sigma_{\mathrm{hi}}$ **then**
                    return `False`
                end if
                if t_{lo}`.x2` $\notin X_{done}$ **then**
                    `Push(`$X_{waiting}$`,` t_{lo}`.x2)`
                end if
            end for
        end if
    end while
    return `True`
end function

Note that the algorithm can be extended to compute the low level entry states, if a transition with a high level event is discovered before reaching a marked state by following invisible transitions backwards. This suggests implementing an additional data structure, that can store the associations between the bad entry states, the transitions that fail the marked string test and the corresponding high level state. Such an extended algorithm with an additional data structure is implemented in the practical part of this thesis.

6.2 Verification of the Locally Nonblocking Condition

Given a deterministic projected system (G, G^{hi}). The locally nonblocking condition verifies, that no local transition path in G starting at a entry state reaches a state, that has not paths to all high-level events that are reachable in the high-level automaton. If this is the case then the low level automaton is *locally blocking* and the system can get stuck in the lower hierarchy of the system model.

The definition of the locally nonblocking condition is given in [Sch05b].

The algorithm of the MARKEDSTRINGACCEPTANCE function was stated in a simplified version[1], that computes only a boolean result. This algorithm is a detailed example for the computation of a more verbose result in conjunction with an additional data structure, that can be used for the visualization of the result in a graphical user interface.

Algorithm 6.2 (Locally Nonblocking)**.** Given the projected system as stated above with an entry state map $EntryStateMap$ that is the result of the deterministic projection function DETPROJECT. The function LOCALLYNONBLOCKING then returns `True` if the locally nonblocking condition is fulfilled and `False` if not. The helper functions LOWEXITSTATES, REACHABLEEVENTS and COMPACCREACH are introduced as stated below.

function LOCALLYNONBLOCKING(`Generator` $G^{\mathrm{lo}}_{\&}$, `Generator` $G^{\mathrm{hi}}_{\&}$, `Map<Idx, StateSet>` $EntryStateMap_{\&}$, `Vector<LnbBlockingStates>` $BlockingResult$)
    / local variables */*
    `Bool` $result$:= `True`
    `TransSet<Idx, StateSet>` $\delta_{\mathrm{lo},r}$ */* reverse sorted low level transition relation */*
    `EventSet` $\Sigma_{\mathrm{hi},active}$ */* high-level active event set */*
    `StateSet` $X_{\mathrm{lo},exitstates}$ */* low-level local exit states */*
    `StateSet` $X_{\mathrm{lo},blockingexitstates}$ */* low-level locally blocking exit states */*
    `StateSet` $X_{\mathrm{lo},blocking}$ */* low-level locally blocking states */*
    `StateSet` $X_{\mathrm{lo},accreach}$ */* low-level local accessible reach */*
    `StateSet` $X_{\mathrm{lo},coaccreach}$ */* low-level local coaccessible reach */*
    `StateSet` $X_{\mathrm{lo},nonblockingexitstates}$ */* low-level locally nonblocking exit states */*
    / start */*
    / iteration over all high-level states */*
    for all $x_{\mathrm{hi}} \in X_{\mathrm{hi}}$ **do**
        / clear local sets */*

[1] of the implementation in the practical part of the thesis

$\Sigma_{\text{hi},active} := \emptyset$; $X_{\text{lo},exitstates} := \emptyset$; $X_{\text{lo},blockingexitstates} := \emptyset$; $X_{\text{lo},blocking} := \emptyset$
$X_{\text{lo},accreach} := \emptyset$; $X_{\text{lo},coaccreach} := \emptyset$; $X_{\text{lo},nonblockingexitstates} := \emptyset$
/ compute active high-level event set accumulate and low level exit states */*
for all $t_{\text{hi}} \in$ `Transitions(`δ_{hi}`,` x_{hi}`)` **do**
$\Sigma_{\text{hi},active} := \Sigma_{\text{hi},active} \cup \{t_{\text{hi}}\texttt{.ev}\}$
LowExitStates(G^{hi}, $\delta_{\text{lo},r}$, $EntryStateMap$, $t_{\text{lo}}\texttt{.x2}$, $X_{\text{lo},exitstates}$)
end for
if $\Sigma_{\text{hi},active} \neq \emptyset$ **then**
continue; */* proceed with next high-level state */*
end if
/ check if local exit states reach all high-level events */*
for all $x_{\text{lo},exit} \in X_{\text{lo},exitstates}$ **do**
if $\Sigma_{\text{hi},active} \subset$ ReachableEvents(G^{lo}, G^{hi}, $x_{\text{lo},exit}$) **then**
$X_{\text{lo},blockingexitstates} := X_{\text{lo},blockingexitstates} \cup \{x_{\text{lo},exit}\}$
end if
end for
/ compute locally nonblocking exit states */*
$X_{\text{lo},nonblockingexitstates} := X_{\text{lo},exitstates} \setminus X_{\text{lo},blockingexitstates}$
/ accumulate accessible reach of entry states */*
for all $x_{entry} \in EntryStateMap\,[x_{\text{hi}}]$ **do**
CompAccReach(G^{lo}, Σ_{hi}, $X_{\text{lo},accreach}$, x_{entry})
end for
/ accumulate coaccessible states for nonblocking exit states */*
for all $x_{exit} \in X_{\text{lo},blockingexitstates}$ **do**
CompCoaccReach($\delta_{\text{lo},r}$, Σ_{hi}, $X_{\text{lo},coaccreach}$, x_{exit})
end for
/ compute locally blocking states */*
$X_{\text{lo},blocking} := X_{\text{lo},accreach} \setminus X_{\text{lo},coaccreach}$
if $X_{\text{lo},blocking} \neq \emptyset$ **then**
$result :=$ `False` */* locally nonblocking test fails */*
/ add blocking states with high-level state and low-level entry states to result */*
`PushBack(`$BlockingResult$`,`
`LnbBlockingStates(`x_{hi}`,` $EntryStatesMap\,[x_{\text{hi}}]$`,` $X_{\text{lo},blocking}$`))`
end if
end for
return $result$
end function

The LowExitStates function computes the low-level exit states for a high-level state.

The definition of local exit states is given by the algorithm.

function LowExitStates(Generator $G^{\mathrm{hi}}_{\&}$, TransSet<Sortx2evx1> $\delta^{\mathrm{lo},r}_{\&}$, Map <Idx, StateSet> $EntryStateMap_{\&}$, Idx x^{hi}, StateSet $X^{\mathrm{lo},exitstates}$)
  / local variables */*
  for all $x_{entry} \in EntryStateMap\left[x^{\mathrm{hi}}\right]$ **do**
    for all $t_{\mathrm{lo}} \in$ TransitionsByx2($\delta^{\mathrm{lo},r}$, x_{entry}) **do**
      if t_{lo}.ev $\in \Sigma_{\mathrm{hi}}$ **then**
        $X^{\mathrm{lo},exitstates} := X^{\mathrm{lo},exitstates} \cup \{t_{\mathrm{lo}}.\mathtt{x1}\}$
      end if
    end for
  end for
end function

The set of high-level events, that can be reached from a low-level state is computed by the ReachableEvents function.

function ReachableEvents(Generator $G^{\mathrm{lo}}_{\&}$, Generator $G^{\mathrm{lo}}_{\&}$, Idx x^{lo})
  / local variables */*
  Stack<Idx> $X_{waiting}$
  EventSet $\Sigma_{reachable}$
  StateSet X_{done}
  Idx $x_{current}$
  / start */*
  Push($X_{waiting}$, x^{lo})
  $X_{done} := X_{done} \cup \left\{x^{\mathrm{lo}}\right\}$
  while $X_{waiting} \neq \emptyset$ **do**
    $x_{current}$:= Pop($X_{waiting}$)
    for all $t_{\mathrm{lo}} \in$ Transitions(δ_{lo}, $x_{current}$) **do**
      if t_{lo}.ev $\in \Sigma_{\mathrm{hi}}$ **then**
        $\Sigma_{reachable} := \Sigma_{reachable} \cup \{t_{\mathrm{lo}}.\mathtt{ev}\}$

      else if t_{lo}.x2 $\notin X_{done}$ **then**
        / if low-level event and not already in done set */*
        Push($X_{waiting}$, t_{lo}.x2)
        $X_{done} := X_{done} \cup \{t_{\mathrm{lo}}.\mathtt{x2}\}$
      end if
    end for
  end while
  return $\Sigma_{reachable}$
end function

Corresponding to the CompAccReach function introduced in Section 4.1.2, that computes the accessible reach of invisible transitions, the CompCoaccReach computes the coaccessible reach. A `Sortx2evx1` sorted set of transitions is given for following transitions backwards.

function CompCoaccReach(`TransSet<Sortx2evx1>` $delta_{\&}^{r}$, `EventSet` $\Sigma_{\&}^{\mathrm{hi}}$, `StateSet` $X_{\&}^{coaccreach}$, `Idx` x^{lo})
    / local variables */*
    `Stack<Idx>` $X_{waiting}$
    `Idx` $x_{current}$
    / start */*
    `Push(`$X_{waiting}$`,` x^{lo}`)`
    $X_{coaccreach} := X_{coaccreach} \cup \left\{x^{\mathrm{lo}}\right\}$
    while $X_{waiting} \neq \emptyset$ **do**
        $x_{current} :=$ `Pop(`$X_{waiting}$`)`
        for all $t_{\mathrm{lo}} \in$ `TransitionsByx2(`δ^{r}`,` $x_{current}$`)` **do**
            if t_{lo}`.ev` $\notin \Sigma^{\mathrm{hi}} \wedge t_{\mathrm{lo}}$`.x1` $\notin X^{coaccreach}$ **then**
                `Push(`$X_{waiting}$`,` t_{lo}`.x1)`
                $X^{coaccreach} := X^{coaccreach} \cup \{t_{\mathrm{lo}}$`.x1`$\}$
            end if
        end for
    end while
end function

For post processing the resulting locally blocking states a data structure `Vector< LnbBlockingStates>` is introduced.

The type `LnbBlockingStates` is defined as follows:

```
struct LnbBlockingStates {
  Idx HiState /* high level state */
  StateSet LowEntries /* low level entry states */
  StateSet LowBlocking /* low level blocking states */
};
```

By holding the high level state and the low level entry states for each set of locally blocking states a graphical environment can provide a detailed visualization of the result. In the practical part of the thesis, such a supporting data structure for storing failed states is implemented for each of the verification methods in [Sch05b].

Chapter 7

Conclusions and Outlook

In SCT, computation requires efficient data structures and algorithms. In this thesis, a software model of an automaton is developed followed by the implementation of algorithms applied to the software model. The practical part of the thesis consists of an implementation of a software library for SCT in C++.

As a basis for further discussion the basic concepts of finite automata and regular language theory are introduced in Chapter 2, followed by an outline of the main results of RW control theory.

The software model of a automaton is developed in Chapter 3. At first two abstract data models of an automaton are introduced, the linked-list model and the set based model. These abstract models are evaluated by three algorithms, parallel composition, language projection and subset construction. In addition, the computational complexity of several important data access patterns that occur in inner loops of algorithms is shown. The evaluation concludes in a decision for the set based model as the evaluation by the projection algorithm has shown a weakness in the linked-list model. In the subsequent sections important implementation aspects of a set based automaton model are discussed. The resulting data structure specification is then implemented as the `Generator` automaton class abstract data types (ADTs) introduced before.

Following the implementation of the `Generator` class, algorithms for regular languages and finite automata are stated in Chapter 4. This includes the language methods parallel composition, projection and the automata algorithms determine and state space minimization, computing the equivalent deterministic automaton for a nondeterministic one and a canonical recognizer for a deterministic automaton, respectively.

In Chapter 5, the automaton data structure is extended to the requirements of supervisory control by specification of another ADT that can hold properties of events. For this specification, a more general implementation is stated, that can also be used to extend the `Generator` class to other approaches. The extended automaton model is then implemented as the `cGenerator` class. Algorithms for monolithic supervisory control, implemented by the `cGenerator` class are stated at the end of the chapter. Here the algorithmic solution of the basic supervisory control problem is presented.

Chapter 6 introduces implementations of algorithms for hierarchical control of decentralized systems according to the approach provided in [Sch05b]. A by-product of the deterministic projection, a map of high-level states to their low-level entry states is used to effectively implement the algorithms.

The practical part, the so called *FAUDES* software library is the main result of this thesis. It provides flexible set based data structures for modelling DES, that can be extended to different theory approaches. All basic automata and language algorithms are implemented. Also an algorithm for efficient monolithic supervisor computation and controllability test in the RW control theory is included. In addition all verification algorithms for the hierarchical and decentralized approach provided in [Sch05b] are implemented, with data structures for the visualization of the results. Further on the library provides advanced features for the use as a base library in a graphical user interface. Beside the implementation of exception handling it provides extensible and user friendly data structures on top of the base classes for automata memory management in large applications. The library is also designed to implement mappings between the state indices of automata, that have been computed in functions and the state indices of function input automata by the usage of globally unique state indices. In addition a simple console application was developed as an example for using the data structures provided in the library and for testing purposes.

This software library will be provided to the public for free use under the GNU Lesser General Public License. It shall encourage external researchers to adopt the library for their own approaches and contribute to the further development.

Acknowledgement:
I would like to thank Klaus Schmidt and Prof. Th. Moor for providing me this interesting topic and for the freedom of research in developing the software library and writing the thesis. I also want to thank Alexander Dreweke and Gerd Hofmann for always answering my programming language related questions and the Studienprojekt for testing my code and for much fun at work. Finally I would like to thank my girlfriend Katja for her constant love and moral support during the time this thesis was written.

Bibliography

[AHU$^+$83] Alfred V. Aho, John E. Hopcroft, Jeffrey Ullman, J. D. Ullman, and J. E. Hopcroft. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[CC02] A.E.C. Da Cunha and J.E.R. Cury. Hierarchically Consistent Controlled Discrete Event Systems. *IFAC World Congress*, 2002.

[CL99] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer, 1999.

[dCCK02] Antonio E. C. da Cunha, Jose E. R. Cury, and Bruce H. Krogh. An Assume-Guarantee Reasoning for Hierarchical Coordination of Discrete Event Systems. *Workshop on Discrete Event Systems*, 2002.

[dQC00] M. H. de Querioz and J. E. R. Cury. Modular Control of Composed Systems. *American Control Conference*, 2000.

[fau] FAU Discrete Event Systems Library. `http://www.rt.e-technik.uni-erlangen.de/FGdes/`.

[Hop71] John E. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[Knu98] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[Led02] R.J. Leduc. *Hierarchical Interface Based Supervisory Control*. PhD thesis, 2002.

[Les95] Ted Leslie. Efficient Approaches to Subset Construction. Master's thesis, Computer Science, University of Waterloo, 1995.

[LGP] GNU Lesser General Public License. `http://www.gnu.org/copyleft/lesser.html`.

[RW89] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77:81–89, 1989.

[Sch05a] Berno Schlein. Erstellung einer Softwareumgebung fuer Anwendungen aus der Supervisory Control Theory: Hierarchischer und Dezentraler Entwurf, 2005.

[Sch05b] Klaus Schmidt. *Hierarchcal Control of Decentralized Discrete Event Systems, Theory and Application*. PhD thesis, 2005.

[Sup] Supremica. `http://www.supremica.org`.

[TCT] TCT. `http://www.control.utoronto.ca/people/profs/wonham/wonham.html`.

[UMD] UMDES Software Library. `http://www.eecs.umich.edu/umdes/toolboxes.html`.

[Won04] W.M. Wonham. Supervisory Control of Discrete-Event Systems, 2004.

[WR87] W. M. Wonham and P. J. Ramadge. On the Supremal Controllable Sublanguage of a Given Language. *SIAM Journal of Control and Optimization*, 25:637–659, 1987.

[YL00] T. Yoo and S. Lafortune. A Generalized Framework for Decentralized Supervisory Control of Discrete Event Systems. *Workshop on Discrete Event Systems*, 2000.

[ZW01] Z. Zhang and W. Wonham. STCT: an efficient algorithm for supervisory control design, 2001.

Appendix A

Additional Methods

A.1 Algorithms for Nonblocking Hierarchical Control

A.1.1 Verification of Liveness

The *liveness* condition guarantees that each state has at least one transition. It is defined in [Sch05b].

Algorithm A.1 (Liveness)**.** Given an automaton $G = (X, \Sigma, \delta, X_0, X_\mathrm{m})$. The function LIVENESS returns `True` if the automaton is live and `False` if not. The states failing the liveness condition are stored in the reference parameter X_{failed}.

function LIVENESS(`Generator` $G_\&$, `StateSet` $X_\&^{failed}$)
  for all $x \in X$ **do**
    if `Transitions(`δ`,`x`)` $= \emptyset$ **then**
      $X^{failed} := X^{failed} \cup \{x\}$
    end if
  end for
  if $X^{failed} = \emptyset$ **then**
    return `True`
  else
    return `False`
  end if
end function

A.1.2 Verification of Marked String Controllability

The definition of the marked string controllability condition for deterministic projected systems $\left(G, G^{\text{hi}}\right)$ is given in [Sch05b]. It guarantees that if no high-level event is feasible in a marked high-level state of G^{hi}, then the low-level system G still can be controlled such that nonblocking occurs.

Algorithm A.2 (Marked String Controllability)**.** Given a deterministic projected system $\left(G, G^{\text{hi}}\right)$ and an entry state map $EntryStateMap$ that is the result of the deterministic projection by the function DETPROJECT. The MARKEDSTRINGCONTROLLABILITY returns `True` if the marked string controllability condition holds and `False` if not. The helper function LOCALAUTOMATON is called to compute a local automaton starting from a specified entry state. Local automata are defined in [Sch05b].

function MARKEDSTRINGCONTROLLABILITY(`cGenerator` $G^{\text{lo}}_{\&}$, `cGenerator` $G^{\text{hi}}_{\&}$, `Map<Idx,StateSet>` $EntryStateMap_{\&}$)
    / local variables */*
    `Bool` $result :=$ `True`
    `cGenerator` $G_{\text{lo},spec} := G^{\text{lo}}$
    `Bool` $gotucevent$ */* "got an uncontrollable event" */*
    `cGenerator` $G_{\text{lo},local}$
    `cGenerator` $G_{\text{lo},sup}$
    / start */*
    TRIM($G_{\text{lo},spec}$)
    for all $x_{\text{m,hi}} \in X_{\text{m,hi}}$ **do**
        $gotucevent :=$ `False`
        for all $t_{\text{hi}} \in$ `Transitions(`δ_{hi}`,` $x_{\text{m,hi}}$`)` **do**
            if t_{hi}`.ev` $\in \Sigma_{\text{uc,hi}}$ **then**
                $gotucevent :=$ `True`
                break */* exit for all loop */*
            end if
        end for
        if $gotucevent =$ `False` **then**
            for all $x_{entry} \in EntryStateMap\,[x_{\text{m,hi}}]$ **do**
                / compute local automaton */*
                LOCALAUTOMATON(G^{lo}, Σ_{hi}, x_{entry}, $G_{\text{lo},local}$)
                / build supervisor from local automaton */*
                SUPCONNB($G_{\text{lo},local}$, $G_{\text{lo},spec}$, $G_{\text{lo},sup}$)

            if $X_{\mathrm{lo},sup} = \emptyset$ **then**
                $result :=$ `False`
            end if
        end for
    end if
end for
return $result$
end function

The LocalAutomaton function computes a local automaton by following transitions with events t`.ev` $\notin \Sigma^{\mathrm{hi}}$ in the low level automaton.

function LocalAutomaton(`cGenerator` $G^{\mathrm{lo}}_{\&}$, `EventSet` $\Sigma^{\mathrm{hi}}_{\&}$, `Idx` x^{entry}, `cGenerator` G^{local})
    / local variables */*
    `Idx` $x_{current}$
    `Stack<Idx>` $X_{waiting}$
    / start */*
    $\Sigma_{local} := \Sigma$
    $X_{local} := X_{local} \cup \{x^{entry}\}$
    $X_{0,local} := X_{0,local} \cup \{x^{entry}\}$
    `Push(`$X_{waiting}$`,` x^{entry}`)`
    while $X_{waiting} \neq \emptyset$ **do**
        $x_{current} :=$ `Pop(`$X_{waiting}$`)`
        for all t_{lo}`.ev` $\notin$ `Transitions(`δ_{lo}`,` $x_{current}$`)` **do**
            if t_{lo}`.ev` $\notin \Sigma_{\mathrm{hi}}$ **then**
                if t_{lo}`.x2` $\notin X_{local}$ **then**
                    `Push(`$X_{waiting}$`,` t_{lo}`.x2)`
                    $X_{local} := X_{local} \cup \{t_{\mathrm{lo}}$`.x2`$\}$
                end if
                `SetTransition(`δ_{local}`,` t_{lo}`)`
            end if
        end for
        if $x_{current} \in X_{\mathrm{m,lo}}$ **then**
            $X_{\mathrm{m},local} := X_{\mathrm{m},local}\ \{x_{current}\}$
        end if
    end while
end function

A.1.3 Verification of Mutual Controllability

The definition of the mutual controllability condition is given in [Sch05b]. It requires decentralized subsystems of a composed system to agree on the occurrence of shared uncontrollable events.

Algorithm A.3 (Mutual Controllabilty)**.** Given two automata $G_1 = (X_1, \Sigma_1, \delta_1, X_{0,1}, X_{\mathrm{m},1})$ and $G_2 = (X_2, \Sigma_2, \delta_2, X_{0,2}, X_{\mathrm{m},2})$ with the uncontrollable events $\Sigma_{\mathrm{uc},1} \subseteq \Sigma_1$ and $\Sigma_{\mathrm{uc},2} \subseteq \Sigma_2$. The function MUTUALCONTROLLABILITY returns `True` if the automata fulfill the mutual controllability condition and `False` if not.

function MUTUALCONTROLLABILITY(`cGenerator` $G^1_{\&}$, `cGenerator` $G^2_{\&}$)
    / local variables */*
    `EventSet` $\Sigma_{12} := \Sigma_1 \cap \Sigma_2$
    `EventSet` $\Sigma_{\mathrm{uc},12} := \Sigma_{\mathrm{uc},1} \cap \Sigma_{\mathrm{uc},2}$
    `Bool` $result_{12}$
    `Bool` $result_{21}$
    `cGenerator` G_{tmp}
    / start */*
    if $\Sigma_{\mathrm{uc},12} = \emptyset$ **then**
        return `True`
    end if
    /* $p_{21}^{-1}\left(p_{12}\left(L\left(G^2\right)\right)\right)$ */
    DETPROJECT(G^2, Σ_{12}, G_{tmp})
    INVPROJECT(G_{tmp}, Σ_1)
    $result_{12} :=$ CONTROLLABLE(G_{tmp}, G^1, $\Sigma_{\mathrm{uc},12}$)
    /* $p_{12}^{-1}\left(p_{21}\left(L\left(G^1\right)\right)\right)$ */
    DETPROJECT(G^1, Σ_{12}, G_{tmp})
    INVPROJECT(G_{tmp}, Σ_2)
    $result_{21} :=$ CONTROLLABLE(G_{tmp}, G^2, $\Sigma_{\mathrm{uc},12}$)
    if $result_{12} =$ `True` $\wedge\, result_{21} =$`True` **then**
        return `True`
    else
        return `False`
    end if
end function

List of Figures