# Counter-Constrained Finite State Machines: Modelling Component Protocols with Resource-Dependencies[*]

Ralf Reussner

Distributed Systems Technology Center (DSTC) Pty Ltd
Monash University, Melbourne, Australia
`rreussner@dstc.monash.edu.au`

**Abstract**

This report deals with the specification of software component protocols (i.e., the set of service call sequences). The contribution of this report is twofold: (a) We discuss specific requirements of real-world protocols, especially in the presence of components wich make use of limited resources. (b) We define *counter-constrained finite state machines* (CC-FSMs), a novel extension of finite state machines, specifically created to model protocols having dependencies between services due to their access to shared resources. We provide a theoretical framework for reasoning and analysing CC-FSMs. Opposed to finite state machines and other approaches, CC-FSMs combine two valuable properties: (a) CC-FSMs are powerful enough to model realistic component protocols with resource allocation, usage, and de-allocation dependencies between methods (as occurring in common abstract datatypes such as stacks or queues) and (b) CC-FSMs have a decidabile equivalence- and inclusion problem as proved in this report by providing algorithms for efficient checking equivalence and inclusion. These algorithms directly lead to efficient checks for component interoperability and substitutability.

**Keywords:** software component protocols, finite state machine extension, decidable inclusion check, interoperability, substitutability.

[*]This research was undertaken while the author was with the Universität Karlsruhe (TH), Germany. It was supported by the National German Science Foundation (Deutsche Forschungsgemeinschaft (DFG), Graduiertenkolleg No. 209).

# Contents

**1 Introduction** **3**

**2 Requirements for Component Protocol Models** **4**
- 2.1 Component Protocols . . . . . 4
- 2.2 Requirements for Component Protocol Models . . . . . 5
  - 2.2.1 Interoperability and Substitutability Checks . . . . . 5
  - 2.2.2 Expressive Power . . . . . 6
  - 2.2.3 Other Requirements . . . . . 7

**3 Specifying Protocols with State-Machines** **7**
- 3.1 Using Finite State Machines for Protocols Specification . . . . 7
- 3.2 Counter-Constrained Finite State Machines . . . . . 9
- 3.3 Finite State Machines . . . . . 9
- 3.4 Equivalence Check and Minimisation of Finite State Machines 12
- 3.5 Counter-Constrained Finite State Machines . . . . . 14
- 3.6 Decidability of Equivalence and Inclusion . . . . . 15
- 3.7 Cross-Product-Automaton and Shuffle-Automaton . . . . . 27
- 3.8 Characterisation of the Languages Recognised by CC-FSMs . 29
  - 3.8.1 Relationship to the Chomsky Hierarchy . . . . . 30
  - 3.8.2 Recognition of Dyck-languages . . . . . 30
- 3.9 Modelling with Counter-Constraints . . . . . 31
- 3.10 Limitations . . . . . 32

**4 Related Work** **33**
- 4.1 Other Models of Component Protocol Specification . . . . . 33
- 4.2 Other Extensions of State Machines with a Decidable Inclusion or Equivalence Problem . . . . . 34

**5 Conclusions** **35**

# 1 Introduction

The strict organisational and personnel separation of component development and component deployment is seen as a prerequisite for an independent component market [46]. As a consequence of that separation, information on the proper deployment of the component must be transported from the component developer to the component user. It is of practical relevance that information on the proper deployment of a component is not only part of its documentation but is also part of the component's interface to make proper deployment checkable by the middleware. Unfortunately, current commercial middleware component models do not provide enough semantical information on the proper component usage, hence, proper component deployment is impossible to check statically. As a result, assembling systems or reconfiguring systems still introduce many errors which remain undetected during composition time (that is, when you actually expect errors), but are detected later by the system's user, often causing system-breakdowns, unavailability and financial loss. Usually, it is hard to back-trace which construction or reconfiguration step caused the later occurring error. (This situation is similar to debugging code: the location where the effects of a bug occur is often not the location where the bug actually resides. Tracing down the exact location of the bug often constitutes the major effort while debugging.)

Because current commercial component models fail to provide sufficient component models supporting configuration-time interoperability and substitutability checks, research was undertaken into richer interface specifications. Contracts [27] can also be considered as a richer interface specification, hence as an early anticipation of problems arising because of insufficient interface information.

Like in this paper, the primary emphasis of research in enriched interface models was laid on including component protocol specification within component interfaces (e.g., [1, 23, 32, 35, 37, 50, 53]). Mostly, the term component protocol denotes the set of sequences of calls to services supported by a component (i.e., the *provides protocol* as part of the provides interface) [23, 32]. Besides this, also the necessity of specifying the sequences of calls to external services (i.e., the *requires protocol* as part of the requires interface) is pointed out (e.g., [53] or more explicitly and generally in [6]).

To be useful, component protocol models have to fulfil certain practical requirements, such as powerful expressiveness (to model practically relevant protocols). Furtheron, protocol models must have formal properties, e.g., providing efficient algorithms for checking interoperability. Unfortunately, none of the existing interface models fulfils these practical *and* formal requirements. Generally, the dilemma is to define a model, which, on the one hand, is powerful enough to model realistic, widely used protocols, like the provides protocol of a stack, and, on the other hand, is simple enough to possess a decidable inclusion problem. For example, finite state machines (FSMs) have an efficient algorithm to check inclusion, but they are not capable of modelling the provides protocol of a stack. The same is true for linear timed logic (LTL). Opposed to that, push-down automata obviously can model a stack, but the inclusion is not decidable for this automata class (e.g., [16]). (Other examples are given in sections 2 and 4.)

In this report we present counter-constrained finite state machines (CC-FSMs), a new extension of finite state machines which fulfils the above mentioned requirements. Our model is mainly influenced by state machines and algorithms for their analysis and Krämer's work on synchronising interfaces [23, 20]. The theoretical framework presented in section 3 originates mainly from the author's German dissertation [39]. A slightly simplified version (but without proofs) is given in [40].

CC-FSMs allow to model relevant component protocols (as specified in section 2), and to perform efficient checks for interoperability and substitutability. The state machine model itself is defined in section 3. This section also contains a theoretical framework which is used to derive an efficient algorithm for testing the inclusion of protocols. Furtheron, the languages recognised by the state machine model are characterised briefly. Section 4 discusses related work in the area of interface models in general, and of state machine based approaches in particular. Section 5 concludes and presents limitations and issues for future work in the area of interface models. We will use the term automaton and state machine interchangeably.

## 2 Requirements for Component Protocol Models

### 2.1 Component Protocols

Nowadays, all industrial middleware-platforms use so-called signature-list-based interface models, i.e., interface models like CORBA IDL [33] or JAVA interfaces [21] which contain the list of services (methods, functions) provided or required by the component. (Note that JAVA does not model requires interfaces.)

The result of a successful interoperability or substitutability check with these signature-list-based interface models is that one can exclude run-time errors like "invoked method not existing". These checks assume that all methods of a component are available in all states of the component (i.e., are always callable). Practice shows that this is rarely the case. Most often, some services are only callable after the successful completion of others. E.g., a file most be opened before one can read or write its content. Finally, it must be closed. As a result, not all sequences of calls are valid.

The set of valid call sequences is the *provides protocol.* The set of all required call sequences is the *requires protocol.* The benefit of including these protocols within component interfaces is that by a successful interoperability or substitutability check with protocol-modelling interfaces run-time errors like "invoked method not supported in this state" are excluded.

Note that this definition of component protocol is not concerned with the *network* protocol used to invoke services (e.g., TCP/IP used by a CORBA remote procedure call).

In the following, let's identify the provides interface and the provides protocol and, likewise, the requires interface and the requires protocol, because the statements on substitutability and interoperability checks hold for arbitrary interface models, not only for protocols. Let denote $prov_C$ the provides protocol of a component C and $req_C$ the requires protocol of a component C.

## 2.2 Requirements for Component Protocol Models

### 2.2.1 Interoperability and Substitutability Checks

During the construction of new component based software architectures, one usually connects components (which are supposed to interact). As a result, their interoperability have to be checked. (Dealing systematically with component adaptation is another concern for interface models, see subsection on "Other Requirements".) During system reconfiguration (including re-engineering legacy software into a component based architectures), components are substituted by other (i.e., when updating with newer versions, or replacing one component by an assembly of others, etc). To detect common errors during these system (re-)configuration steps, a component interface model must have the following specific properties:

- it must model the provided functionality *and* the required functionality of the component (i.e., the provides interface and the requires interface must be modelled). If the requires protocol is not modelled, one cannot check whether a used component offers the functionality and protocol required by components using it.

- interoperability of components must be checkable: Assuming component $A$ uses services of component $B$, one has to check whether $req_A \subseteq prov_B$.

- substitutability of components must be checkable: Assuming component $A$ is to be substituted for component $B$, one has to check whether $prov_B \supseteq prov_A \wedge req_B \subseteq req_A$.

These last two requirements for protocol analysis boil down to the need of an efficient inclusion test. (We can reduce the inclusion test to an equivalence check and the construction of the intersection, because we can use (for arbitrary sets $A, B$) $A \subseteq B \Leftrightarrow A \cap B = A$.)

For practical purposes an interface model should provide an efficient algorithm for inclusion test. As a consequence, Turing-universal models (such as Turing-machines, $\lambda$-calculus, etc.) are ruled out, since equivalence and inclusion are not decidable for turing-universal models [47, 16]. A further discussion on suitable and unsuitable models is given in section 4.

#### 2.2.2 Expressive Power

Besides these formal requirements, an interface model should (because of obvious practical reasons) be able to model real-world component interfaces. For example, when modelling a stack it is important to specify that the `pop`-operation has never been called more often than the `push`-operation has been called before. Hence, the stack protocol has to model resource-dependencies between its services `push` and `pop`.

In general, these kinds of constraints exist between operations which allocate, de-allocate or use a shared resource piecewise. In the following, the term resources is used for items which have to allocated before, and de-allocated after use. A resource is *piecewise* allocable, deallocable and usable if it can be accessed in specific (resource-specific) units. A not piecewise accessable resource can only be allocated, deallocated or used as a whole.

Examples for piecewise accessable resources are: memory, network connections, files (however, it depends on the access control of the file if it is piecewise accessable or not: If access by records is allowed, the individual file is piecewise accessable, if not, the file is usable only as a whole). Not piecewise usable resources are: sound output or individual printers, but also memory content, like in container data-structures, such as stacks or queues. (In this case the allocation is done by `push` or `insert`, the usage and deallocation with `pop`.)

The following resource-dependencies between services occur in practice [19]:

1. All allocated resources must be de-allocated (but only the allocated resources).

2. Only allocated resources can be used.
3. Every allocation implies a de-allocation (in contrast to the first case it is valid to de-allocate a non-allocated resource).

In section 3.9 we show how to model these resource-dependencies with counter-constraints (as introduced below).

Note that the requirement of expressive power somewhat conflicts with the requirements of efficient algorithms for checking the inclusion: if we use an universal model, of course, we could easily model all kinds of resource-dependencies, but like mentioned above, we cannot perform interoperability or substitutability checks.

#### 2.2.3 Other Requirements

As a consequence of the separation of component development and component deployment, in practice components usually have to be adapted to fit in a new deployment context. This means that interoperability checks usually will fail and component adaptation mechanisms are actually of higher practical importance [37, 38, 43, 53] (and more recently [5]).

Most current approaches (except [23, 12]) are not easy to understand by non-mathematically trained people. This suggests that probably not one single model will have the desired mathematical properties *and* is easy to use. One solution is the use of different models: one for easy specification, and one as an input in analysis algorithms, primarily the inclusion check. Hence, algorithms for translating between different models are an important area of research. For the state machine model developed in this report an alternative way of specification by using a kind of temporal logic constraints given in source-code with a JavaDoc-like syntax is defined by Hunzelmann [19], together with tools for translating these specifications into CC-FSMs.

## 3 Specifying Protocols with a State Machine based Interface Model

### 3.1 Using Finite State Machines for Protocols Specification

Finite state machines (FSMs) [22] are a well-known notation for protocol specification [7, 15, 32, 53]. Most relevant communication protocols (e.g., TCP/IP [45]) are modelled by FSMs, which enable formal analysis and checks [15]. FSMs comprise a finite set of states. When modelling call sequences, we model for each state which methods are callable in this state. In many cases, a method call changes the state of the state machine, i.e., some other methods are callable after the call, while others, callable in the old state, are not callable in the new state. A change of states (from an "old" state to a "new" state) is called *(state) transition*. The Unified Modelling Language (UML) [41] includes the standard notion of FSMs. In the *state transition diagram* in figure 1 states are denoted by circles, transitions by arrows. There is one designated state in which the component is after its creation (i.e., the *start-state*). From this start-state all call sequences start.

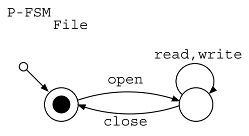


Figure 1: P-FSM of the `File`

A call sequence is only *valid* if it leads the FSM into a so-called *final-state*. These final-states are denoted by black cycles within the states. Hence, some of the call sequences described by the example FSM are `open-read-write-close`, while one cannot use a command sequence like `open-close-read`.

All state machine based models model a protocol by specifying valid call sequences as accepted words. Therefore, the input alphabet (as specified in the following) is the set of all service names of a component.

Usually the state-space $Z_{P_C}$ of the state machine modelling the protocol is different from the space of all internal states $Z_{I_C}$ of the component $C$ (as modelled, for example, with statecharts [13] in the statechart-diagrams of UML [41]). The states of $Z_{P_C}$ can be regarded as the equivalence classes of a partitioning of $Z_{I_C}$. Let denote $\Phi_C$ a total function which maps each state $z \in Z_{I_C}$ to the set of methods of $C$ callable in state $z$. Then $Z_{P_C} \cong Z_{I_C}/\Phi_C$.

The work presented in this report differs from work with communication protocols in two ways: (a) we are interested in properties of component configuration steps (i.e., testing for interoperability or substitutability), while communication protocols are tested for interoperability, liveness, and absence of deadlock [26, 15]. (b) FSMs proved to be sufficient to model relevant properties of communication protocols. Opposed to that, FSMs are not capable to model provides interfaces of common software components. For example, it is easy to prove that the relevant semamtic property of a stack – never more `push` operations than `pop` operations, see figure 7 – cannot be modelled by a FSM accurately. Note that the problem mainly arises when modelling the provides interface: it is important that the provides protocol must not model call sequences which are not supported by the component. Opposed to that, modelling sequences in the requires interface which are never emitted by the component does not affect the validity of the interoperability check (although it can be a nuisance for the user of the component).

## 3.2 Counter-Constrained Finite State Machines

In the following we define the model of counter-constrained finite state machines (CC-FSMs), and we present the constructions of the so-called cross-product-CC-FSM and the shuffle-CC-FSM.

## 3.3 Finite State Machines

Finite state machines (FSMs) were motivated by the description of nerve cells [22]. The control unit of the previously introduced turing machine is a FSM as well [47]. The theory of FSMs was later extended (e.g., [11, 30]). Besides many other areas of application in informatics, FSMs are used successfully in protocol specification and protocol verification [15], but also for software specification in general (see for example [2]). There are various definitions of FSMs. The following definition is suitable for our purposes.

**Definition 3.1 (Finite State Machine)**
A deterministic finite state machine $D$ is a tuple $D = (I, Z, \delta, F, z_0)$, with

- the finite set $I$ being the *input alphabet*,
- the finite set $Z$ being the *state set*,
- the total function $\delta : Z \times I \to Z$ being called *state transition function*,
- the subset $F \subseteq Z$ being the set of *accepting states* (final states), and
- the state $z_0 \in Z$ being the *start state*.

A FSM is in only one state at all times. At the beginning, this is the start state. In every step, one symbol of the input is read, and the state is changed in accordance with the transition function. The changing from one state to another is called *transition*. If the FSM is in one of the accepted states, the sequence of the so far read input symbols is called *accepted word*. The set of all accepted words is called the language recognised by the FSM $L_D \subseteq I^*$ .

**Remark 3.2**
Different to other definitions, in our model it is possible to leave accepting states. There is no such restriction of $\delta$.

**Remark 3.3**
Automata working in the way described above (state transition when reading an input symbol) are also called *real-time automata*, since the processing is effectively done in "real time"; the automaton can read an input symbol only once. (In contrast to other automata models regarding the input as being located on a tape that can be used by the read-head several times in both directions.) Furthermore, there are no state transitions without reading an input symbol (no so-called *$\epsilon$-transitions*).

**Definition 3.4 (Projection of a word)**
A *projection* of a word $w = c_1 \cdots c_n$ with regard to a set of symbols $\Sigma$ is the word $P_\Sigma(w)$ where all letters $c$ of the word $w$ were replaced by the empty word $\epsilon$, if $c \notin \Sigma$.

**Definition 3.5 (Projection of a set of words)**
The projection of a set of words $L$ with regard to a set of symbols $\Sigma$ is the set of words $L_\Sigma := \{P_\Sigma(w) | w \in L\}$

**Remark 3.6**
The function $\delta$ is a total function. Words not recognised by the FSM lead the FSM to a state that is not an accepting state. It is helpful to construct $\delta$ in such a way that all prefixes which can no longer result in an accepted word lead the FSM to an *error-state* $err$. This error-state $err$ is a non-accepting state which cannot be left anymore (thus, $\forall i \in I.\delta(err, i) = err$). It is sufficient to establish one single common error-state for all these prefixes.

**Remark 3.7**
The states together with $\delta$ can be considered a directed graph with $Z$ forming the set of nodes, $\langle z_1, z_2 \rangle$ being connected by an edge, if an input $e \in I$ exists, so that $\delta(z_1, e) = z_2$. Usually, the edge is annotated with the input $e$. It is now possible to apply commonly used terms from graph theory (such as reachability of nodes, cycles, etc.). The nodes corresponding to final states are marked by black circle. The node corresponding with the start state is marked by an arrow.

**Convention 3.8 (Path)**
Since the transition function of a FSM can be represented as a graph, we also speak of *paths* of the FSM, meaning a path in the graph-theoretical sense in the transition function as a directed graph. A path $p$ is indicated by the

sequence of nodes received: $p = [k_1 \cdots k_n]$. If one would like to take into account the input symbols located on the path as well, the path can also be indicated as $p = [(k_1 = \delta(k_0, e_1)), (k_2 = \delta(k_1, e_2)) \cdots (k_n = \delta(k_{n-1}, e_n))]$ or simpler as $p = [(k_1, e_1) \cdots (k_n, e_n)]$, with $k_i \in Z, 0 \leq i \leq n$ and $e_i \in I, 1 \leq i \leq n$. The set $E_p := \{e_i, 1 \leq i \leq n\}$ is also called input symbols of the path $p$. The set of all paths of an FSM $A$ is called $P_A$.

**Definition 3.9 (Final path in a finite state machine)**
Final paths of the FSM are those paths in the graph that run from the start state to an accepting state. It is required from the final path not to have any cycles. The final paths of a FSM $D$ are abbreviated $FP_D$.

**Remark 3.10**
Since $Z$ is finite, every final path is of finite length. Due to the absence of cycles, there is only a finite number of final paths in a FSM.

**Definition 3.11 (Cycle)**
A *cycle* $z$ is a path in a FSM that has identical first and last nodes, therefore $z = [k_1 \cdots k_n = k_1]$. The set of all cycles of a FSM $A$ is called $Cycl_A$.

**Convention 3.12 (Connected paths)**
Two paths $p_1, p_2$ are called *directly connected* ($p_1 \to p_2$) if they have at least one common node. They are called *indirectly connected* if there is at least one path in the FSM that is directly connected with them. If $p_1, p_2$ are connected it also applies that $p_2$ can be *reached* by $p_1$.

**Convention 3.13 (Connected cycles)**
Two cycles $y_1, y_2$ are called *directly connected* ($y_1 \leftrightarrow y_2$) if they have at least one common node. They are called *indirectly* connected if there are cycles in the FSM that are directly connected with $y_1$, $y_2$ as well as mutually connected. Note that the connection is not defined through a path that is not a cycle.

**Convention 3.14 (Connected path and cycle)**
A path $p$ is *directly connected* ($p \leftrightarrow z$) with a cycle $y$ if they have at least one common node. They are called *indirectly* connected if there are cycles in the FSM that are directly connected with $p$ and $z$ as well as mutually. Note that the connection is not defined through a path that is not a cycle.

**Definition 3.15 (Reachable cycles)**
The set of all cycles reachable by a cycle $y$ is called $(y)^*$. It is useful to generalise to a set $Y$ of cycles: $(Y) := \cup_{y \in Y}(y)^*$.

**Definition 3.16 (Sequence of cycles)**
A cycle $y_1$ is located behind another cycle $y_2$ with respect to a final path $p$ ($y_1 \succ_p y_2$) if $y_1$ and $y_2$ are connected with $p$, and $y_2$ cannot be reached by $p$ without reaching $y_1$.

**Definition 3.17 (Sub-FSM)**
The *sub-FSM* $TA(D, s)$ of a FSM $D = (I, Z, \delta, F, z_0)$ is the FSM $(I, Z, \delta, F, s)$ whose start state is the state $s$. (Informally: The sub-FSM is the FSM in $D$ that follows the state $s$.)

**Definition 3.18**
**Equivalence and Isomorphism of Finite State Machines**
Two FSMs

$$A = (I_A, Z_A, \delta_A, F_A, z_{0_A})$$

and

$$B = (I_B, Z_B, \delta_B, F_B, z_{0_B})$$

are considered *equivalent* if $L_A = L_B$. $A$ and $B$ are called *isomorphic*$(A \cong B)$ if the states of $A$ can be mapped injectively onto the states of $B$, and if this mapping fulfils the following homomorphism constraint with respect to the transition function:

$$\begin{aligned} \exists \pi : Z_A \to Z_B . \forall z \in Z_A . \forall e \in I . \pi(\delta_A(z_a, e)) &= \delta_B(\pi(z), e) \wedge \\ \pi(z_{0_A}) &= z_{0_B} \wedge \\ \pi(F_A) &= F_B \end{aligned}$$

(In the last line, the extension of $\pi$ to sets is being used.) If $A \cong B$ applies, one also writes $\delta_A \cong \delta_B$ if the transition function is being examined.

## 3.4 Equivalence Check and Minimisation of Finite State Machines

For our application of state machines in an interface model for components, the question of equivalence of two components is important. This question is dealt with in the section on the decidability of the equivalence of CC-FSMs 3.5. Since CC-FSMs machines are based on FSMs (section 3.6), we also require a procedure to check the equivalence of two FSMs. Luckily, for every FSM with a total transition function there is an equivalent minimal FSM, which is unique modulo isomorphic state renamings. In the following, the term minimality will be defined, and an algorithm for minimising FSMs will be introduced.

**Definition 3.19 (Minimal Finite State Machine)**
A FSM $A$ is minimal if $L_A = L_B \Rightarrow |Z_B| \geq |Z_A|$ applies to all equivalent FSMs $B$.

The algorithm for minimising FSMs introduced here was developed by D. Huffman and E. Moore [18, 8] and makes use of the idea that equivalent states can be combined.

**Definition 3.20 (Equivalence of states)**
Two states $z_1, z_2 \in Z$ are equivalent if the transition function executes, for every input, a transition into the same state (which depends only on the input, but not on the state), thus

$$z_1 \cong z_2 :\Leftrightarrow \forall e \in I . \delta(z_1, e) = \delta(z_2, e) \tag{1}$$

This corresponds to $TA(D, z_1) \cong TA(D, z_2)$.

The search for equivalent states starts with a partition $P = (F, (Z \backslash F))$ of the stateset $Z$ which contains exactly two elements: the first element of the partition is the set of accepting states, the second element is the set of non-accepting states. The algorithm continues to refine this partitioning until a fixpoint has been reached. For reasons of completeness and to demonstrate its low complexity, this algorithm will be given here.

**Algorithm 3.21 (Minimisation of Finite State Machines)**

```
Input: a FSM $A$
Output: a minimal FSM $A'$ equivalent to $A$.
for each set $p \in P$ do
    new $\leftarrow \emptyset$;
    first $\leftarrow$first state in partition p;
    next $\leftarrow$next state in partition p or
                null if no next state in p exists;
    for each state $z \in p \backslash \{\text{first}\}$ do
        $\langle$separate non-equivalent states in p$\rangle$
        for each input symbol $e \in I$ do
            if $\delta(z, e)$ is in another partition than $\delta(\text{first}, e))$ then
                add $z$ to new;
            fi
        od
    od
    add new to $P$;
od
```

This algorithm combines all equivalent states in sets, which means it factorises $Z_A$ according to the equivalence relation 1. The resulting FSM $A'$ of the algorithm is defined as follows:

$$\begin{aligned}
I_{A'} &:= I_A \\
Z_{A'} &:= P \ \langle \textit{states of } A' \textit{ are thus sets os states of } A. \rangle \\
\delta_{A'}(z,e) &:= \delta_A(x,e), \text{ with } x \in z \text{ arbitrary} \\
z \in F_{A'} &:\Leftrightarrow \exists f \in z. f \in F_A \\
z_{0_{A'}} &:= z \in Z_{A'}. z_{0_A} \in z
\end{aligned}$$

To test the equivalence of two states, $\mathbf{O}(|I_A|)$ steps are required, so that we obtain the time complexity $\mathbf{O}(|Z_A| \cdot |I_A|)$ for the test of all states. This is called a pass. A pass needs to be repeated if the partition has been changed during the pass. At worst, there are no equivalent states (if $A$ is already minimal), and we obtain $\mathbf{O}(|Z|)$ sets in the partition, which means the partitioning is changed exactly $|Z|-1$ times, which means at worst $|Z|-1$ passes are required and one more to establish that the partitioning is not changing anymore. The following lemma results from this.

**Lemma 3.22 (Time complexity of the minimisation)**
*The time complexity of algorithm 3.21 to minimise a FSM lies in* $\mathbf{O}(|Z|^2 \cdot |E|)$.

In [17], an algorithm for the minimisation of FSMs with a total transition function is introduced, whose time complexity lies in $\mathbf{O}(|Z| \cdot \log(|Z|))$.

## 3.5 Counter-Constrained Finite State Machines

According to section 3.1, the power of FSMs is not sufficient for the intended interface description of software components. Therefore, the expressiveness of FSMs is extended by counters, which leads to the definition of the counter-constrained finite state machine (CC-FSM).

**Definition 3.23 (Counter)**
Every input symbol $e \in E$ has a counter $\#e$ counting the number of occurrences of the input symbol $e$ in a (partial) word. To indicate the word $w$, in which the counting was carried out, we write $\#_w e$.

As input symbols are mapped injectively onto counters, counters are identified with input symbols if the context is clear.

**Definition 3.24 (Counter constraints)**
A counter constraint is a triple $(A, B, k)$ with two disjunct sets $A$ and $B$ of input symbols and a natural number $k$. Four *types* of counter constraints are defined:

**Dist**$(A, B, k)$ := for a word $w$ and all prefixes of $w$ applies $\sum_{a\in A} \#a \geq \sum_{b\in B} \#b + k$.

**DistEq**$(A, B, k)$ := for all prefixes of a word $w$ applies $\sum_{a\in A} \#a \geq \sum_{b\in B} \#b + k$ and for the word $w$ applies $\sum_{a\in A} \#a = \sum_{b\in B} \#b + k$.

**ReversedDist**$(A, B, k)$ := for a word $w$ and all postfixes of $w$ applies $\sum_{a\in A} \#a \geq \sum_{b\in B} \#b + k$.

**ReversedDistEq**$(A, B, k)$ := for all postfixes of a word $w$ applies $\sum_{a\in A} \#a \geq \sum_{b\in B} \#b + k$ and for the word $w$ applies $\sum_{a\in A} \#a = \sum_{b\in B} \#b + k$.

The sets $A$ and $B$ of a counter constraint $c$ are also called $c_A$ and $c_B$. It is now possible to define the CC-FSM.

**Definition 3.25 (Counter-constrained State Machines)**
A *counter-constrained finite state machine* (CC-FSM) is a tuple $(E, Z, \delta, F, z_0, C)$, with $D = (E, Z, \delta, F, z_0)$ describing a deterministic FSM and $C$ being a set of counter constraints. The language recognised by the CC-FSM exactly contains those words from $L_D$ for which all constraints in $C$ are true.

Since every CC-FSM contains a FSM, the conventions and definitions regarding *path*, *final path*, *cycle*, direct and indirect connectedness are transferable to counter-constrained finite state machines, as counters are not of importance for these path-related terms.

**Convention 3.26 (Usability of a path)**
A path $p$ of a counter-constrained finite state machine $Z$ is *usable* if there is a word $w \in L_Z$ that is accepted through the path $p$. Formally: If $p = [(z_1, e_1), \cdots, (z_n, e_n)]$, then there is at least one prefix $u$ and at least one postfix $v$, to which applies $w = ue_1 \cdots e_n v \in L_Z$. $u$ and $v$ may be empty.

**Convention 3.27 (Usability of a cycle)**
A cycle $z$ of a counter-constrained finite state machine $Z$ is called *usable* if there is an unlimited number of words $w \in L_Z$ that are accepted through the cycle $z$. Formally: If $z = [(z_1, e_1), \cdots, (z_n, e_n)]$, then there is at least one prefix $u$ and at least one postfix $v$, to which applies $|\{n|w = u(e_1 \cdots e_n)^n v \in L_Z\}| = \infty$. $u$ and $v$ may be empty.

## 3.6 Decidability of Equivalence and Inclusion

The efficient decidability of the equivalence and the inclusion of interfaces is important for the subsequent application of counter-constrained finite state machines in an interface model. For a practically relevant subset of counter-constrained finite state machines, efficient algorithms for checking equivalence and inclusion will be given here.

**Definition 3.28 (Leftcycles, Rightcycles)**
The *leftcycles* of a counter constraint $z$ are the subset $LC$ of the cycles of a counter-constrained finite state machine $K$, which is defined as follows:

$$LC_z := \{y \text{ is cycle in } K | E_y \cap z_A \neq \emptyset\}$$

The *rightcycles* $RC$ are defined analogously:

$$RC_z := \{y \text{ is cycle in } K | E_y \cap z_B \neq \emptyset\}$$

($E_y$ designates the set of input symbols of the cycle $y$, see convention 3.8.) Informally, a cycle is a leftcycle of a counter constraint $z$ if at least one of its input symbols occurs on the left side of the counter constraint (thus, $z_A$). This applies analogously to the rightcycles and the right side (thus, $z_B$) of the counter constraint.

**Remark 3.29**
Left- and rightcycles of a counter constraint $z$ do not necessarily have to be disjunct. (Since cycles may contain different input symbols, input symbols from $z_A$ and $z_B$ may occur in a cycle.)

At first, it is required from counter constraints not to destroy the usability of paths.

**Definition 3.30 (Valid counter constraint)**
A counter constraint $z$ is *valid* for a FSM $Z$ if there does not exist any cycle or path in $Z$ whose usability is destroyed by $z$.

**Remark 3.31**
If all non-usable cycles of a counter-constrained finite state machine $Z$ are removed by changing the transition function, one obtains a new counter-constrained finite state machine $Z'$ which recognises the same language ($L_Z = L_{Z'}$). Thus, the precondition to use only valid counter constraints is not really a restriction. In the following, only valid counter constraints will be examined.

The following requirement, in contrast, represents a restriction from the theoretical point of view, but has no effects on the practical use (compare remark 3.33).

**Definition 3.32 (Uniform counter constraint)**
A counter constraint $z \in C_Z$ is *uniform* with respect to a counter-constrained finite state machine $Z$, if

$$\forall y_1, y_2 \in Cycl_Z . \forall p \in P(y_1, y_2) . \Sigma_{a \in z.A} \#_p a = \Sigma_{b \in z.B} \#_p b \qquad (2)$$

applies, with $P(y_1, y_2)$ being defined as the set of all paths leading from the start state to a final state, thereby touching the cycles $y_1$ and $y_2$ (in that order), and not containing any cycles themselves (not containing $y_1$ and $y_2$ either).

**Remark 3.33 (Uniform counter constraints are sufficient)**
For the provides interface, counter-constrained finite state machines with uniform counter constraints can always be used. A FSM with a non-uniform counter constraint would prescribe an use of a resource for at least $n$-times (with a statically determined, fixed $n$) in a path; the thus required $n$-times release of the resource, however, would be modelled only by a counter constraint. Since the number of calls of the release-operation is statically determined as well, the same behaviour can always be modelled so that the $n$-times use *and* release of the resource via the path are prescribed and only other (for example, more frequent) uses and releases by counter constraints are checked.

For the equivalence check, the relations between cycles created by counter constraints are important. A uniform counter constraint $z$ establishes a set $Exp_z \subseteq \mathbf{N}^2$ for two cycles $y_1, y_2$:

$$Exp_z(y_1, y_2) := \{(n_1, n_2) | n_1 \cdot (\Sigma_{a \in z.A} \#_{y_1} a) - n_2 \cdot (\Sigma_{b \in z.B} \#_{y_2} b) - z.k \ \mathrm{rel}_{z.t} 0\} \quad (3)$$

$\mathrm{rel}_{z.t}$ "=" means, if the type of $z$ is `DistEq` or `ReversedDistEq`. If the type of $z$ is `Dist` or `ReversedDist`, $\mathrm{rel}_{z.t}$ means "$\geq$". The relation between cycles of a counter-constrained finite state machine and the language recognised by it is examined in the following considerations.

**Convention 3.34**
We agree on the set $T_Z(y_1, y_2) \subset \mathbf{N}^2$ for two cycles $y_1, y_2$ of the counter-constrained finite state machine $Z$ as

$$T_Z(y_1, y_2) := \{(n_1, n_2) | \exists \alpha, \beta, \gamma \in (I_Z)^* . \alpha {y_1}^{n_1} \beta {y_2}^{n_2} \gamma \in L_Z\} \qquad (4)$$

**Convention 3.35**
Based on the above convention, for a counter-constrained finite state machine $Z$ and two of its cycles $y_1, y_2$, we agree on the set $W_Z(y_1, y_2) \subset L_Z$ as

$$W_Z(y_1, y_2) := \{w | w = \alpha {y_1}^{n_1} \beta {y_2}^{n_2} \gamma \in L_Z, (n_1, n_2) \in T_Z(y_1, y_2)\} \qquad (5)$$

$W_Z(y_1, y_2) \subseteq L_Z$ clearly depends on the relation between the cycles $y_1$ and $y_2$. It is also possible that more than one counter constraint affects the cycles $y_1$ and $y_2$. Thus, we define

$$R_Z(y_1, y_2) := \cap_{z \in C_Z} Exp_z(y_1, y_2) \tag{6}$$

When checking the equivalence of two counter-constrained finite state machines $A, B$ one needs to test whether all sets $T_A(y_1, y_2)$ and $T_B(y_1, y_2)$ of two counter-constrained finite state machines are equal for all pairs of cycles $y_1, y_2$ (given that $\delta_A = \delta_B$).

Since these sets are not finite, this procedure cannot be directly executed. In the following, it will be described how it is possible to test equality or inclusion of the sets $Exp_z(y_1, y_2)$ without explicitly calculating them.

**Definition 3.36 (Relations between cycles)**
A counter constraint $z$ is provided. Furthermore, $y_1 \in LC(z)$ and $y_2 \in RC(z)$ shall apply. The tuple $r_z := (t, k, l, r)$ is designated as the *relation* between $y_1, y_2$ induced by $z$. The type $t$ of the relation equals the type of the counter constraint $z$ (meaning, `Dist`, `DistEq`, `ReversedDist` or `ReversedDistEq`. Also, $k := z.k$, and $l, r$ are counter variables: $l$ shall indicate the number of occurrences of the symbols from $z.A$ in $y_1$, $l := \Sigma_{a \in z.A} \#_{y_1} a$, $r$ the number of occurrences of the symbols from $z.B$ in $y_2$, thus $r := \Sigma_{b \in z.B} \#_{y_2} b$.[1]

Since $Z$ can have several counter constraints, several relations may exist between $y_1$ and $y_2$. The possible connections between these relations will be defined now.

**Definition 3.37 (Compliant and excluding relations)**
Two relations $r_{z_1} = (t_1, k_1, l_1, r_1)$ and $r_{z_2} = (t_2, k_2, l_2, r_2)$ between two cycles $y_1, y_2$ can be related in the following ways.

$r_{z_2}$ **strengthens** $r_{z_1}$ if

$$Exp_{z_2} \subseteq Exp_{z_1} \tag{7}$$

$r_{z_1}$ **is compliant with** $r_{z_2}$ if

$$Exp_{z_1} = Exp_{z_2} \tag{8}$$

$r_{z_1}$ **and** $r_{z_2}$ **exclude each other** in all other cases. (This is actually merely a necessary constraint. When explicitly constructing the sets $Exp_{z_1}, Exp_{z_2}$, one would be able to establish whether $Exp_{z_1} \cap Exp_{z_2} = \emptyset$, which is the exact exclusion criterion.)

[1]For sake of simplicity and brevity a different definition of a relation is given in [40].

In the following, it will be demonstrated how one can determine with the help of the relations $r_{z_1}, r_{z_2}$ if strengthening, compliance or exclusion exist, without explicitly calculating the sets $Exp_{z_1}, Exp_{z_2}$.

If both $t_1$ and $t_2$ are `DistEq` or `ReversedDistEq`, it applies to $(n_1, n_2) \in Exp_{z_1}$

$$n_1 \cdot (\Sigma_{a \in z_1.A} \#_{y_1} a) - n_2 \cdot (\Sigma_{b \in z_1.B} \#_{y_2} b) - z_1.k = 0 \tag{9}$$

which may be expressed by the use of $l_i := z_i.l = \sum_{a \in z_i.A} \#_{y_1} a$, $r_i := z_i.r = \sum_{b \in z_i.B} \#_{y_2} b$ and $k_i := -z_i.k / r_i$ as $n_1 \cdot l_1 - n_2 \cdot r_1 + k_1 = 0$. (These definitions merely serve to prepare the geometrical interpretation which will be given subsequently.) If $(n_1, n_2) \in Exp_{z_2}(y_1, y_2)$, too, $n_1 \cdot l_2 - n_2 \cdot r_2 + k_2 = 0$ applies. This can be written as

$$\begin{aligned} n_2(n_1) &= \frac{l_1}{r_1} \cdot n_1 + k_1 \wedge && (10) \\ n_2'(n_1') &= \frac{l_2}{r_2} \cdot n_1' + k_2 && (11) \end{aligned}$$

If this is understood as functions $n_2, n_2' : \mathbf{R} \to \mathbf{R}$, two straight lines are described by the equations (10) and (11). Thus, compliance exists always when the straight lines are identical; $r_{z_2}$ strengthens $r_{z_1}$, if $n_2'(n_1') \geq n_2(n_1)$ always applies.

**Example 3.38**
In figure 2, two relations are visualised as straight lines. The relation $rel_2 = (\texttt{Dist}, 3, 1, 1)$ strengthens the relation $rel_1 = (\texttt{Dist}, 2, 1, 2)$. The points entered correspond to the tuples $(n_1, n_2)$, which are sufficient for both relations.

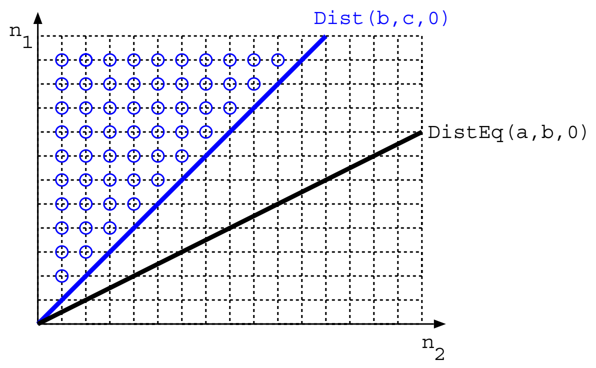


Figure 2: Visualisation of relations as lines in a plane

Conclusion:

**Lemma 3.39**
*If $t_1$ and $t_2$ both are either* `DistEq` *or* `ReversedDistEq`*, strengthening already implies compliance, since only identical straight lines may be subsets of each other. Equation 7 is fulfilled if applies:*

$$l_1 = l_2 \wedge r_1 = r_2 \wedge k_1 = k_2 \tag{12}$$

*If $t_1$ and $t_2$ both are either* `Dist` *or* `ReversedDist`*, equation (12) has to apply to the compliance as well, for the strengthening ($r_2$ strengthens $r_1$)*

$$\frac{l_2}{r_2} \geq \frac{l_1}{r_1} \wedge k_2 \geq k_1 \tag{13}$$

*is sufficient.*

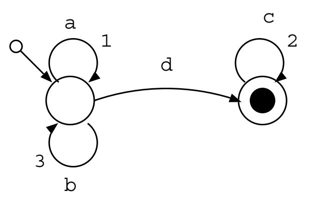
**Remark 3.40**
To fulfil (7), the precondition (12) is necessary. The weaker condition $\frac{l_2}{r_2} = \frac{l_1}{r_1} \wedge k_2 \geq k_1$ is not sufficient for (12). If $l_1 \neq l_2$ and $r_1 \neq r_2$, with $\frac{l_2}{r_2} = \frac{l_1}{r_1}$, $n'_2(n'_1) = n_2(n_1)$ only applies if $n_2, n'_2$ are considered real or rational functions. However, since $n_2, n'_2$ are functions of $\mathbf{N} \to \mathbf{N}$, only the stronger constraint (12) ensures that $n'_2(n'_1) = n_2(n_1)$.

**Remark 3.41**
Note that $k_i = k_i(z_i, r_i)$. According to the definition of $k_i$, in order to test $k_2 \geq k_1$, it needs to be examined whether $k_2 \cdot r_2 \geq k_1 \cdot r_1$ applies.

**Definition 3.42 (Cycle relation table)**
A *cycle relation table* (CRT) of a counter-constrained finite state machine $Z$ is a two-dimensional table into which the relations between all pairs of cycles $(y_1, y_2) \in LC(z) \times RC(z)$ for all $z \in C_Z$ have been entered. $y_1$ is agreed upon as the *row index*, $y_2$ be the *column index*. Each entry $CRT(y_1, y_2)$ represents a set of tuples.

**Example 3.43 (Cycle relation table)**
In figure 3(a), a counter-constrained finite state machine with the counter constraints `DistEq(a,b)`, `Dist(b,c)` is shown. (b) depicts its CRT.

**Remark 3.44 (Entering data into the CRT)**
If one finds an already occupied field when entering a relation into the CRT, this may lead (a) to no change of the entry (in the case of compliant relations) or (b) to a strengthening of the entry or (c) to an error in the case of mutually excluding relations. For finding out, which counter constraints are contradictory for which cycles, one has to store for each entry in the CRT the counter constraints which led to this entry.



(a)

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | - | - | (DistEq,0,1,1) |
| 2 | - | - | - |
| 3 | - | (Dist,0,1,1) | - |

(b)

Figure 3: Counter-constrained finite state machine with its CRT

**Convention 3.45 (Non-conflicting counter constraints)**
If no mutually excluding relations occur when filling in the CRT of a counter-constrained finite state machine $Z$, $C_Z$ is called *non-conflicting.*

**Remark 3.46**
The relations "strengthening" and "compliance" between relations serve to calculate the set $R_Z(y_1, y_2)$, thus, the intersection of the sets $Exp_z(y_1, y_2)$. The procedure introduced here to test these relations leads to a very simple computation of the intersections: identity in the case of compliance or the smaller (contained) set is the intersection in the case of strengthening. Instead of using this simple calculation of the intersection, other procedures to describe the strengthening can be applied as well. (Generally, it is possible to indicate the strengthening by a set of segments of straight lines. As a result, fewer conflicts occur when filling in the CRT, and thus also those counter-constrained finite state machines could be dealt with that have conflicting sets of counter constraints when applying the procedure applied here. But the efficient procedure introduced here is sufficient for the use of counter-constrained finite state machines to model the provides interface of software components.)

**Lemma 3.47**
*If no conflict occurs when filling in the CRT of a counter-constrained finite state machine $Z$, and if $C_Z$ has only uniform counter constraints, to all cycles $y_1, y_2 \in Cycl_Z$ applies:*

$$R_Z(y_1, y_2) = CRT_Z(y_1, y_2) \tag{14}$$

**Proof 3.48**
In the case of two non-conflicting counter constraints $z_1, z_2 \in C_Z$, regarding two cycles $y_1, y_2 \in Cycl_Z$ it applies either

- they strengthen each other: thus for example $Exp_{z_1}(y_1, y_2) \subset Exp_{z_2}(y_1, y_2)$ (the proof is symmetric for the case of $Exp_{z_1}(y_1, y_2) \supset Exp_{z_2}(y_1, y_2)$), or
- they are compliant: thus, $Exp_{z_1}(y_1, y_2) = Exp_{z_2}(y_1, y_2)$

Since merely those two cases occur for non-conflicting and uniform counter constraints when constructing the CRT, it applies that the set of those tuples described by the entry into $CRT(y_1, y_2)$ is equal to $Exp_{z_i}(y_1, y_2)$, with the following applying to $z_i \in C$: $\forall z \in C.Exp_{z_i}(y_1, y_2) \subset Exp_z(y_1, y_2)$. Thus, $Exp_{z_i}(y_1, y_2) = \cap_{z \in C} Exp_z(y_1, y_2) = R_Z(y_1, y_2)$. ■

*In this case, the sets $R_Z(y_1, y_2)$ definitely depend on the $CRT_Z$.*

Due to the above-mentioned, it is possible to prove the following lemma.

**Lemma 3.49**
*If two counter-constrained finite state machines A and B have only valid, uniform and non-conflicting counter constraints, the following applies:*

$$\delta_A \cong \delta_B \wedge CRT_A \cong CRT_B \Rightarrow L_A = L_B \tag{15}$$

**Proof 3.50**
For a proof by contradiction it is assumed that $L_A \neq L_B$. Therefore, $\exists w \in L_A.w \notin L_B$. (The proof is symmetric for the case $w \in L_B \wedge w \notin L_A$.) The word $w$ has at least two cycles $\bar{y}_1, \bar{y}_2$ that are related. (Otherwise, this would immediately result in a contradiction to $\delta_A \cong \delta_B$.) Thus, $w$ may also be written as:

$$w = \alpha \bar{y}_1^{\bar{n}_1} \beta \bar{y}_2^{\bar{n}_2} \gamma$$

(with $\alpha, \beta, \gamma$ suitable). Conclusion: $(\bar{n}_1, \bar{n}_2) \in R_A(\bar{y}_1, \bar{y}_2)$. On the other hand, however, $w \notin L_B$ is $\delta_A \cong \delta_B$, thus $\not\exists y_1, y_2 \in Cycl_B.\exists(n_1, n_2).w = \alpha {y_1}^{n_1} \beta {y_2}^{n_2} \gamma$, and thus

$$\exists y_1, y_2 \in Cycl_B.R_B(y_1, y_2) \neq R_A(\bar{y}_1, \bar{y}_2) \tag{16}$$

Lemma 3.47 states that the sets $R_A(y_1, y_2)$ clearly depend on $CRT_A$, and the sets $R_B(y_1, y_2)$ definitely on $CRT_B$. Since it is provided that $CRT_A \cong CRT_B$, it applies that

$$\forall y_1, y_2 \in Cycl_A = Cycl_B.R_A(y_1, y_2) = R_B(y_1, y_2)$$

This represents a contradiction to equation (16). ■

**Lemma 3.51 (Usability of cycles)**
*If a counter-constrained finite state machine has only valid and non-conflicting counter constraints, all its cycles are usable.*

**Proof 3.52**
A valid counter constraint receives the usability of all cycles according to definition. Thus, for each counter constraint $z_i$ and for each pair of cycles $(y_1, y_2))$ there is at least one word $w = \alpha(y_1)^{n_1}\beta(y_2)^{n_2}\gamma$, with $\alpha, \beta, \gamma$ suitable and $(n_1, n_2) \in Exp_{z_i}(y_1, y_2)$. If a cycle should not be usable, this may occur only with more than one (valid) counter constraint. These counter constraints $z_a, \cdots, z_b$ thus exclude the word $w$ $(a, b \in \mathbf{N}, b > a)$. Therefore, $Exp_{z_a}(y_1, y_2) \cap \cdots \cap Exp_{z_b}(y_1, y_2) = \emptyset$. This represents a contradiction to the absence of conflicts of the counter constraints. ■

To demonstrate that two counter-constrained finite state machines are equivalent (accept the same language) exactly when their transition functions and their cycle relation tables are equivalent, specific requirements have to be made to the counter constraints.

**Definition 3.53 (Simple counter constraint)**
A *simple counter constraint* is a counter constraint $(A, B, k)$ where $|A| = |B| = 1$. Instead of $(A, B, k)$, we also write $(a, b, k)$, with $a$ being the only element of $A$, $b$ the only element of $B$.

**Definition 3.54 (Chain of counter constraints)**
A sequence of simple counter constraints $(a_i, b_i, k_i), 0 \leq i \leq n, n >$ forms a *chain* if $b_i = a_{i+1}, 0 \leq i < n$ applies. A set of counter constraints is called *chained* if a chain exists which its entire counter constraints are part of. A chain is called *cyclic* if $a_0 = b_n$ applies.

**Remark 3.55**
A cyclic chain needs to contain at least one `ReversedDist(Eq)`- constraint and at least one `Dist(Eq)`-constraint. If the chain consisted merely of `Dist(Eq)`-constraints (or merely of `ReversedDist(Eq)`-constraints), the conditions with respect to the prefixes (or postfixes, as the case may be) could not be fulfilled.

**Remark 3.56**
The cyclic dependencies of the counter constraints can be fulfilled only by counters having the same value at the end of the input. This applies no matter if all constraints test the equality at the end of the input or not (meaning, if `Dist`-constraints or `DistEq`-constraints, or, as the case may be, `ReversedDist`-constraints or `ReversedDistEq`-constraints exist). That

is why many different cyclic chains describe the same facts - that all counters need to have the same value.

The following definitions are meant to exclude the case that different sets of counter constraints describe the same restriction.

**Definition 3.57 (Normalised set of counter constraints)**
A cyclic chain is *normalised* if all counter constraints contained within the chain test the equality at the end of the word. (Which means, contain only `DistEq`-constraints or `ReversedDistEq`- constraints.) A set of counter constraints is normalised if all chains within the set are normalised.

With this knowledge, it is now possible to transfer the term of minimality from FSMs (definition 3.19) to counter-constrained finite state machines.

**Definition 3.58 (Minimal counter-constrained finite state machine)**
A counter-constrained finite state machine $(E, Z, \delta, F, z_0, C)$ is *minimal* if

- $(E, Z, \delta, F, z_0)$ is minimal and
- $C$ is normalised.

With these definitions, it is possible to prove the following theorem:

**Theorem 3.59**
*The following applies to two minimal counter-constrained finite state machines $A = (I_A, Z_A, \delta_A, F_A, z_{0_A}, C_A)$ and $B = (I_B, Z_B, \delta_B, F_B, z_{0_B}, C_B)$, both having only valid, uniform, non-conflicting and simple counter constraints:*

$$L_A = L_B \Leftrightarrow \delta_A \cong \delta_B \wedge CRT_A \cong CRT_B \tag{17}$$

*Note that $\delta_A \cong \delta_B$ requires $Z_A \cong Z_B$, $F_A \cong F_B$ and $z_{0_A} \cong z_{0_B}$ to apply.*

**Proof 3.60**

- "$\Leftarrow$" is lemma 3.49.
- "$\Rightarrow$" will be proven in two steps.

  1. First, we will demonstrate that $L_A = L_B \Rightarrow \delta_A \cong \delta_B$. We will use an arbitrarily chosen $f \in FP_A$. The word $w_f \in L_A$, which has been accepted through the path $f$, is being examined. Since, according to the precondition, $L_A = L_B$, $w_f \in L_B$ applies. The path through which $B$ accepts the word $w_f$ be $f'$. To show that $f'$ is a final path in $B$, it needs to be excluded that $f'$ contains cycles. We assume that $f'$ contains at least one cycle. Thus, $w$ can be

written as $\alpha\beta^n\gamma$, with $\alpha, \beta, \gamma \in {I_B}^*$ suitable. Since $B$ contains only valid and non-conflicting counter constraints, all cycles are usable (according to lemma 3.51). We conclude that there is an unlimited number of words $w_i$ in $L_B$ with $w_i = \alpha\beta^{n_i}\gamma$. In particular, there is the word $w_0 := \alpha\gamma \in L_B$. Since $L_A = L_B$, $w_i \in L_A$ applies to all $i \geq 0$. Thus, there is a cycle in $f$ as well, since all $w_i = \alpha\beta^{n_i}\gamma$ have at least the prefix $\alpha$ in common with $f$. As $A$ is deterministic, $A$ has to use the prefix $\alpha$ of $f$ when accepting the $w_i$. The fact that $f$ contains a cycle is contradictory to the choice of $f$ as final path. Thus, $f' \cong f$ and $f'$ is a final path in $B$, too. The same argumentation applies to all paths in $FP_B$. Thus, $FP_A \cong FP_B$.

Furthermore, we need to demonstrate that $Cycl_A \cong Cycl_B$ applies. We therefore assume for a proof by contradiction that $Cycl_A \ncong Cycl_B$ applies. Thus, there is a cycle $y \in Cycl_A$ to which applies $\nexists y' \in Cycl_B . y \cong y'$. (Otherwise, we flip $A$ and $B$.) There is no word in $L_B$ which is accepted through the $y$ (since $y' \notin Cycl_B$). Since $L_A = L_B$ applies, there is no word in $L_A$ which is accepted through the cycle $y$. We conclude that $y$ is not usable in $A$. This represents a contradiction to the precondition that $A$, according to lemma 3.51, has only valid and non-conflicting counter constraints.

2. In the second step, we will show that $L_A = L_B \wedge \delta_A \cong \delta_B \Rightarrow CRT_A = CRT_B$. We assume for a proof by contradiction that

$$\exists y_1, y_2 \in Cycl_A . CRT_A(y_1, y_2) \neq CRT_B(y_1, y_2)$$

Two cases may cause this inequality.

**First case:** Inequality due to a reversed-relation and a non-reversed-relation. (For example, `ReversedDist` vs. `Dist`.) Since all counter constraints are non-conflicting in accordance with the precondition, the entry with the reversed-relation has been created only by reversed-counter-constraints. Analogously, the non-reversed-relation has been created by non-reversed counter constraints only. Thus, if $R_A(y_1, y_2) \neq \emptyset, R_B(y_1, y_2) \neq \emptyset$, then $R_A(y_1, y_2) \neq R_B(y_1, y_2)$. This represents a contradiction to $L_A = L_B$. Thus, it has to apply that $R_A(y_1, y_2) = R_B(y_1, y_2) = \emptyset$. This is a contradiction to the precondition of $y_1$ and $y_2$ being usable.

**Second case:** Inequality due to an Eq-relation and a non-Eq-relation. (For example, `DistEq` vs. `Dist`.) We assume, without restricting the universality, that $CRT_A(y_1, y_2)$=`DistEq` and

$CRT_B(y_1, y_2) = \texttt{Dist}$. (Analogous argumentation for the reversed-variant.) Since the counter constraints are normalised, the `Dist`-relation is not based upon a `Dist`-counter constraint that is located in a cyclic chain. Thus, the `Dist`- relation is strictly weaker than the `DistEq`-relation. Therefore,

$$(\bar{n}_1, \bar{n}_2) \in T_B(y_1, y_2).(\bar{n}_1, \bar{n}_2) \notin T_A(y_1, y_2)$$

exists. Conclusion:

$$\exists w \in W_B(y_1, y_2).w \notin W_A(y_1, y_2) \tag{18}$$

with $w = \alpha {y_1}^{\bar{n}_1} \beta {y_2}^{\bar{n}_2} \gamma$ suitable for $\alpha, \beta, \gamma$ from $I_B{}^*$. Since $(\bar{n}_1, \bar{n}_2) \notin T_A(y_1, y_2)$, $w \notin L_A$ applies. On the other hand, due to $W_B \subseteq L_B$, $w \in L_B$. This represents a contradiction to $L_A = L_B$.

■

**Corollary 3.61**
**(Complexity of the equivalence check of counter-constrained finite state machines)**
*The construction of cycle relation tables and their equality check requires many steps in $|C_A| \cdot |Cycl_A|^2$. (Note that $C_A = C_B$ as well as $Cycl_A \cong Cycl_B$ can be considered as preconditions, since they are necessary prerequisites for equivalence anyway.) $|Cycl_A|$ can be estimated by the number of transitions (which, in the case of a deterministic FSM, cannot be larger than $|Z|^2 \cdot |I|$, thus $|Cycl_A| < 2^{|Z|^2 \cdot |I|}$). In the subsequent application of the interface modelling, however, the number of cycles will be noticeably smaller. Since the counter-constrained finite state machines must be minimised, one can conclude from lemma 3.22 that the complexity of the equivalence check for counter-constrained finite state machines is $\mathbf{O}(max(|Z|^2 \cdot |E|, |C_A| \cdot |Cycl_A|^2))$.*

**Lemma 3.62 (Inclusion of counter-constrained finite state machines)**
*As described in section 3.7, the intersection of two languages recognised by counter-constrained finite state machines can be recognised by counter-constrained finite state machines. That is why the inclusion check for counter-constrained finite state machines meeting the requirements given in theorem 3.59 is successful. To arbitrarily chosen $A, B$ applies: $A \subseteq B \Leftrightarrow A \cap B = A$.*

**Remark 3.63**
If the counter constraints fail to meet the requirements of theorem 3.59 (in particular, non-simple counter constraints), one may, instead of carrying out the equivalence check through the cycle relation table, execute a conservative equivalence check between two counter-constrained finite state machines $A$ and $B$, which tests whether

1. $\delta_A \cong \delta_B$ and
2. $C_A = C_B$

applies. This check recognises two non-equivalent counter-constrained finite state machines as such, since the above-mentioned preconditions are sufficient for the equivalence. However, the above-mentioned preconditions are not necessary, as the following counter-example will prove. Picture two counter-

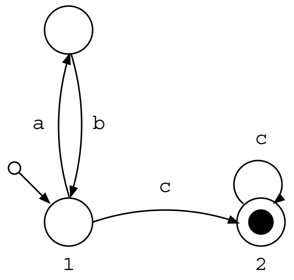


Figure 4: Example of the conservatism of the above-mentioned simple equivalence check (counter constraints in the text). This example would be treated correctly by an equivalence check over the CRT.

constrained finite state machines $A$ and $B$, which have the same transition function shown in figure 4. The counter-constrained finite state machine $A$ has the set of counter constraints $C_A := \{\texttt{DistEq}(a, c)\}$, the counter-constrained finite state machine $B$ has the set of counter constraints $C_B := \{\texttt{DistEq}(b, c)\}$. Both state machines recognise the same language $L_A = L_B = \{(a|b)^n c^n\}$, which means they are equivalent although they have different sets of counter constraints.

## 3.7 Cross-Product-Automaton and Shuffle-Automaton

The languages recognised by FSMs are closed under the intersection (for example, [16]). There is also a procedure to construct one automaton from two FSMs $A$ and $B$, which describes the intersection of the languages $L_A$ and $L_B$. This FSM is called cross-product-FSM.

**Definition 3.64**
**(Cross-product of Counter-Constrained Finite State Machines)**
The cross-product-CC-FSM $K = (I_{A\times B}, Z_{A\times B}, \delta_{A\times B}, F_{A\times B}, z_{0_{A\times B}}, C_{A\times B})$ of two counter-constrained finite state machines $A = (I_A, Z_A, \delta_A, F_A, z_{0_A}, C_A)$ and $B = (I_B, Z_B, \delta_B, F_B,$
$z_{0_B}, C_B)$ is defined as follows:

- $I_{A\times B} := I_A \cap I_B$
- $Z_{A\times B} := Z_A \times Z_B$
- $\delta_{A\times B}((z_a, z_b), e) := (\delta_A(z_a, e), \delta_B(z_b, e))$, with $z_a \in Z_A$, $z_b \in Z_B$ und $e \in I_A \cap I_B$
- $(z_a, z_b) \in F_{A\times B} :\Leftrightarrow z_a \in F_A \wedge z_b \in F_B$
- $z_{0_{A\times B}} := (z_{0_A}, z_{0_B})$
- $C_{A\times B} := C_A \cup C_B$

**Remark 3.65**
The above definition of $\delta$ generally leads to a set of non-accepting states that cannot be left anymore. These states are thus equivalent and may therefore always be combined to one single error-state of $A \times B$. (If the states of this set are considered as tuples, at least one component contains the error-state of $A$ or $B$.)

**Remark 3.66**
The sets of counter constraints are joined $C_{A\times B} := C_A \cup C_B$, since words from the intersection of the languages $L_A, L_B$ have to fulfil all counter constraints. It needs to be considered here that the properties of non-conflicting sets of counter constraints or merely of valid counter constraints of the single automata do not necessarily have to be transferred to the cross-product-automaton. If necessary, those properties can be obtained by adapting $\delta_{A\times B}$.

The definition of the cross-product-automaton represents the parallel execution of the two single automata. The word is accepted if both automata reach an accepting state. Therefore, the input alphabet may be restricted to the intersection of the input alphabets of both single automata (symbols not located within the intersection lead to an error-state in one of the automata). A similar construction for automata with disjunct input alphabets is the so-called shuffle-FSM, which recognises the so-called shuffle-languages (or the so-called shuffle-product) of $L_A$ and $L_B$ [44]. The shuffle-product of two languages is a "mixture" of the languages. A word $w$, which may contain symbols of both input alphabets $I_A$ and $I_B$, belongs to the shuffle-language if the word being created when all symbols in $w$ that belong to $I_A$ are deleted belongs to $L_B$ and when all symbols belonging to $I_B$ are deleted, the thus created word belongs to $L_A$. The definition of the shuffle-automaton is identical with that of the cross-product-automaton except for the definition of the input alphabet and of the transition function.

**Definition 3.67 (Shuffle Counter-Constrained Finite State Machine)**
The shuffle-CC-FSM $S = (I_{A*B}, Z_{A*B}, \delta_{A*B}, F_{A*B}, z_{0_{A*B}}, C_{A*B})$ of two CC-FSMs $A = (I_A, Z_A, \delta_A, F_A, z_{0_A}, C_A)$ and $B = (I_B, Z_B, \delta_B, F_B, z_{0_B}, C_B)$ with $I_A \cap I_B = \emptyset$ is defined as follows.

- $I_{A*B} := I_A \cup I_B$
- $Z_{A*B} := Z_A \times Z_B$
- For the transition function, it is required that $I_A$ and $I_B$ are disjunct:
$$\delta_{A*B}((z_a, z_b), e) := \begin{cases} (\delta_A(z_a, e), z_b) & \text{if } e \in I_A \\ (z_a, \delta_B(z_b, e)) & \text{if } e \in I_B \end{cases} \tag{19}$$
- $(z_a, z_b) \in F_{A*B} :\Leftrightarrow z_a \in F_A \wedge z_b \in F_B$
- $z_{0_{A*B}} := (z_{0_A}, z_{0_B})$
- $C_{A*B} := C_A \cup C_B$

The remark regarding the final state that is applicable to the cross-product-CC-FSMs (3.65) applies here as well.

**Remark 3.68**
The complexity of the cross-product- and shuffle-automata construction is primarily founded in the placing of the transitions (which lies in $\mathbf{O}(|Z_A| \cdot |Z_B| \cdot (|I_A| + |I_B|)))$. The complexity can thus be estimated by $\mathbf{O}(S^2 \cdot I)$, with $S := \max(|Z_A|, |Z_B|)$, $I := \max(|I_A|, |I_B|)$.

Since the definitions for the construction of the cross-product-automaton and the shuffle-automaton are based on counter-constrained finite state machines, the following lemma can be formulated.

**Lemma 3.69**
**(CC-FSMs closed under the intersection and shuffle-operation)**
*The languages recognised by counter-constrained finite state machines are closed under the intersection and the shuffle-operation.*

## 3.8 Characterisation of the Languages Recognised by CC-FSMs

The counter-constrained finite state machines discussed above are motivated by the application, i.e., to describe components which use piecewise allocable resources. Thus, the classification of languages recognised by counter-constrained finite state machines is not the focus of this report. Some statements, however, are of interest.

### 3.8.1 Relationship to the Chomsky Hierarchy

The first two examples given below demonstrate that the set of languages recognised by counter-constrained finite state machines is located "somewhat orthogonal" to the Chomsky-hierarchy.

It is possible to prove that the counter-constrained finite state machines defined in 3.25 are unable to recognise the proper context-free language $L_{\text{pal-marked}} := \{w\$w^R\}$ (i.e., the language of palindrome words with a marked centre). This is emphasised by the following plausibility consideration. The reason for this is that the counter- constrained finite state machines have no stack where $w$ could be stored; they would merely be able to check with their counters whether different input symbols occurred with the same frequency, which, however, is neither sufficient nor necessary. This language, however, is even recognised by deterministic push-down-automata (thus, in a sense, is located "close" to the regular languages), and moreover, even in real-time (which also corresponds to the processing in the case of counter-constrained finite state machines).

The counter-constrained finite state machines defined in 3.25 recognise the proper context-sensitive language $L_{\text{tripel-n}} := \{a^n b^n c^n\}$. This is achieved by the CC-FSM having the transition function depicted in figure 5 and the (normalised) set $C := \{\text{DistEq}(a, b, 0), \text{DistEq}(b, c, 0)\}$.

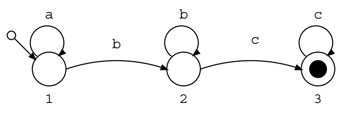


Figure 5: Transition function of a counter-constrained finite automaton able to recognise $L_{\text{tripel-n}}$.

### 3.8.2 Recognition of Dyck-languages

The language family of the Dyck-languages (for example, see [42]) which will now be briefly introduced, is interesting because it is suitable to model allocation and release of resources. A Dyck-language $D_n$ over an alphabet of $n$ different pairs of brackets is the language of all words correctly bracketed. For an arbitrary but fixed $n \in \mathbf{N}$, a counter-constrained finite state machine in accordance with definition 3.25 recognising $D_n$ can be constructed. (Note, however, that for every $n$ a different transition function needs to be indicated.) The transition function of a counter-constrained finite state machine recognising $D_2$ is pictured in figure 6. The set of counter constraints is

$$C := \{\text{DistEq}((,),0), \text{DistEq}([,],0)\}$$

This transition function has a "sub-automaton" for every sequence of open

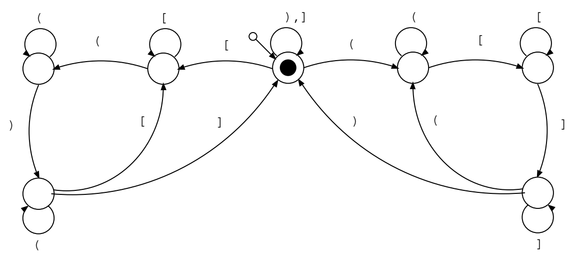


Figure 6: Transition function of a counter-constrained finite state machine recognising $D_2$.

brackets; to the left of the start state in the diagram is the sub-automaton for the sequence "[(", to the right the one for "([". These sub-automata ensure the exclusion of sequences starting with the closing of a pair of brackets although another pair of brackets has not been closed yet. The counter constraints make sure that, for each type of brackets, there are never more close brackets than open brackets and that there is a close bracket for every open bracket at the end of the input. This construction scheme can be transferred to $D_n, n > 2$.

## 3.9 Modelling with Counter-Constraints

According to the list of resource dependencies given in section 2.2 counter-constraints can be used in the following way. In the sequel, $a$ denotes a component's service allocating a resource, $d$ a service de-allocating a resource, and $u$ a service using this resource.

1. All allocated resources must be de-allocated: `DistEq(`$a$`,`$d$`,0)`
2. Only allocated resources can be used: `Dist(`$a$`,`$u$`,0)`
3. Every allocation implies a de-allocation (in contrast to the first case it is valid to de-allocate a non-allocated resource): `ReversedDist(`$d_r$`,`$a_r$`,0)`

The classical example (as given in the motivation) is, of course, the abstract datatype of a stack. A stack [3] can be seen as one of the most fundamental and widely used abstract datatypes. The relevant property of a stack's provides interface is that one can push elements on it (`push`-operation) and can retrieve the most recently pushed element (`top`-operation). The `pop`-operation retrieves and removes the most recent pushed element. As a consequence, one never can call `pop` more often than one has called `push` before.

The transition function of the stack's provides interface automaton is extremely simple. The above mentioned restriction is expressed in the counter-constraint. $\mathit{Dist}(\texttt{push}, \texttt{pop}, 0)$ expresses, that `pop` can be called more often than `push`. Likewise, the annotational specification only consists of this one

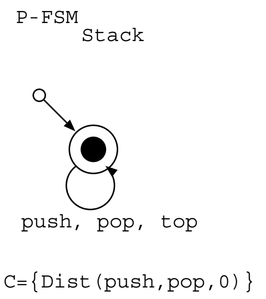


Figure 7: The provides interface of a stack

constraint: $\mathit{Dist}(\texttt{push}, \texttt{pop}, 0)$.

## 3.10 Limitations

As argued in section 2.2, a universal model is of limited use when modelling component interfaces. Choosing a non-universal model means that one only can model some (but not all) aspects of the component behaviour. Of practical relevance for component based software engineering is the aspect of the component's provides and requires protocols. Although our model is capable of modelling most relevant protocols, our model fails to specify the complete behavioural semantics of components. As an example, the different semantics of the abstract datatypes stack and queue are not reflected in our interface model. Both abstract datatypes have the same protocol (neglecting different naming conventions), especially the same counter-constraint that one never can remove more items than available. But, of course, the semantic difference of a stack and a queue (FIFO versus LIFO) is not reflected in our model.

# 4 Related Work

## 4.1 Other Models of Component Protocol Specification

Component models of wide use defined within current industrial middleware platforms (i.e., the Common Object Request Broker Architecture (CORBA) from the Object Management Group [33], Enterprise Java Beans (EJB) from Sun Microsystems [9] and the Common Object Model (COM) with many variants or different namings (e.g., DCOM, COM+,.NET) from Microsoft Corp. [31]). The interfaces of components from these platforms include only signature lists. Except for CORBA, requires interfaces are not specified. None of these models gives rules for specifying valid call sequences. As argued in this paper missing requires interfaces hinder interoperability checks. Missing rules decrease the benefit of interoperability and substitutability checks substantially, because many common errors remain undetected. The drawbacks of commercial component models and their precursors in research labs as mentioned in the introduction gave rise to interface definitions including rules for behavioural specifications. These rules have been expressed in different notations, each having specific advantages and drawbacks.

Predicate based approaches for specifying protocols [25, 54] can describe protocols of arbitrary complexity. Unfortunately, this universality makes checking protocol compatibility uncomputable. Restrictions in the universality enable at least complex checks. For example, restricted logic based calculi, especially variants of temporal logics have been investigated to model protocols [12, 26]. Similarly, approaches using process calculi [28] to provide more interface information are able to specify real-world protocols, but do not provide efficient algorithms for checking protocol compatibility [50].

Petri-nets [34] are a powerful modelling technique for concurrent processes. Many variants and algorithms for checking various properties exist (e.g., [36]). In [52] Petri-nets are used for modelling components within software architectures. While efficient algorithms exist for some Petri-net models to check global properties (such as liveness, absence of deadlocks, etc.) other properties which are important for architectural system configuration (like interoperability and substitutability checks) cannot be checked in general. So-called bounded Petri-nets can be translated into FSMs, and, hence, can make use of the FSM's benefits. Of course, the beneficial properties of Petri-nets, such as a neat representation of even large state-spaces are lost when translating them into FSMs.

The use of finite state machines to model protocols and to check their compatibility is well known from the telecommunication and distributed systems communities, e.g., [15, 24]. Finite state machines are also deployed successfully for modelling object behaviour and specifying and automating test sequences [4]. Nierstrasz proposes their use to model the type of an object [32]. In his approach finite state machines only describe the provides interface.

## 4.2 Other Extensions of State Machines with a Decidable Inclusion or Equivalence Problem

A useful overview of the context-free language classes relevant in our context is given in [10]. Unfortunately, little is known about the decidability of counter-constrained finite state machines working in real-time. On the one hand, the works of Higuchi et al. [14] show that the equivalence problem for DROCAS (deterministic restricted one-counter automata) working in real-time is decidable. These automata have a finite automaton serving as control unit and a stack serving as counter. The term *restricted* means that no symbol exists to designate the end of the stack. Thus, it is not possible to determine during the processing of the input that the stack is empty (the empty stack is used merely as a precondition for acceptance at the end of the input). In the case of DROCAS, the requirement of real-time processing does not influence the decidability of the equivalence and inclusion. If one allows a stack-end-symbol and does not use real-time processing, the result is a DOCA (deterministic one-counter automaton), of which it is known that only the equivalence remains decidable [48, 49]. If more than one counter is permitted, already with two counters one obtains a universal machine, which means neither equivalence nor inclusion are decidable [29].

On the other hand, the requirement of real-time processing really restricts the power of a turing machine [16].

Unfortunately, it is not determined whether equivalence and inclusion are decidable in the case of a model that has more than one counter but works in real-time. Also, counter-constrained finite state machines (such as the one described here) with many counters, which, however, merely have to fulfil counter constraints and cannot be taken into account in the transition function, have yet not been theoretically examined.

Krämer's work on *synchronising interfaces* [23, 20] heavily influenced the here presented extension of FSMs. No statements on the decidability of the equivalence are given for his model with (simple) counter constraints. The above definition of a subset of finite automata with simple counter constraints can be applied to his model, too.

## 5 Conclusions

We discussed general requirements for calculi modelling software component protocols. Especially, we concentrated on the requirements of modelling protocols of services sharing resources (like stacks, queues, etc.) and checking interoperability and substitutability statically at configuration-time. We presented the model of counter-constrained finite state machines, an extension of finite state machines. Our new model has the amendable properties of being able to model those protocols with resource-dependencies and having a decidable inclusion and equivalence problem.

Future work is suggested in the area of automata theory as well as practical computer science, namely component meta models. As discussed in the related work section, it is an open problem whether equivalence and inclusion are decidable in the case of a model that has more than one counter but works in real-time. Also, it is not clear whether the decidability results remain if one extends the here proposed model by constraints dealing with sums of counters instead of single counters.

Although the area of component interoperability checking is of practical relevance, future work should concentrate on (a) the systematic treatise of interoperability problems (i.e., component adaptation) and (b) the analyses of more powerful interface models specifying extra-functional properties, such as performance and reliability.

## Acknowledgements

The author would like to acknowledge the insights and help given in many fruitful discussions with Gerhard Goos, Jun Han, Gunnar Hunzelmann, Bernd Krämer, Wolfram Menzel, Kathrin Paschen, Heinz Schmidt, Roland Vollmar, and Thomas Worsch.

## References

[1] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[2] Helmut Balzert. *Lehrbuch der Software-Technik: Teil 1: Software-Entwicklung.* Spektrum Akademischer Verlag, Heidelberg, Germany, 1996.

[3] F. L. Bauer. The cellar principle of state transition and storage allocation. *Annals of the History of Computing*, 12(1):41–49, 1990.

[4] G. V. Bochmann, E. Cerny, M. Gagné, C. Jarda, A. Léveillé, C. Lacaille, M. Maksud, K. S. Raghunathan, and B. Sarikaya. Experience with formal specifications using and extended state transition model. *IEEE Trans. Communications*, 30(12):2506–2511, December 1982.

[5] Andrea Bracciali, Antonio Brogi, and Carlos Canal. Dynamically Adapting the Behaviour of Software Components. In *Coordination*, volume 2315 of *Lecture Notes in Computer Science*, pages 88–95. Springer-Verlag, Berlin, Germany, 2002.

[6] Přemysl Brada. Towards automated component compatibility assessment. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP'01)*, June 2001.

[7] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In Volker Gruhn, editor, *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01)*, volume 26, 5 of *SOFTWARE ENGINEERING NOTES*, pages 109–120, New York, September 10–14 2001. ACM Press.

[8] E.F. Moore. Gedanken-experiments on sequential machines. In C.E. Shannon and J. MacCarthy, editors, *Automata Studies*, pages 129–153, Princeton, New Jersey, 1956. Princeton University Press.

[9] Sun Microsystems Corp., The Enterprise Java Beans homepage. http://java.sun.com/products/ejb/.

[10] Jürgen Freudig. Konformitätsprüfung jenseits von Typanalyse. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe (TH), Germany, September 1998.

[11] Victor M. Gluschkov. *Theorie der abstrakten Automaten*. Number XIX in Mathematische Forschungsberichte. VEB Deutscher Verlag der Wissenschaften, Berlin (Ost), Germany, 1963.

[12] Jun Han. Temporal logic based specification of component interaction protocols. In *Proccedings of the 2nd Workshop of Object Interoperability at ECOOP 2000*, Cannes, France, June 12.–16. 2000.

[13] D. Harel. Statecharts: a visuel approach to complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[14] Ken Higuchi, Mitsuo Wakatsuki, and Etsuji Tomita. A polynomial-time algorithm for checking the inclusion for real-time deterministic restricted one-counter automata which accept by accept mode. *TIEICE: IEICE Transactions on Communications/Electronics/Information and Systems*, E81(1), 1998.

[15] Gerald J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, Englewood Cliffs, NJ, USA, 1991.

[16] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, USA, 1979.

[17] J. E. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite-automaton. In Z. Kohavi, editor, *Theory of Machines and Computations*, pages 189–196. Academic Press, New York, NY , USA, 1971.

[18] D. A. Huffman. The synthesis of sequential switching circuits. In *Edward F. Moore (Ed.), Sequential Machines: Selected Papers.* Addison-Wesley, Reading, MA, USA, 1964.

[19] Gunnar Hunzelmann. Generierung von Protokollinformation für Softwarekomponentenschnittstellen aus annotiertem Java-Code. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe (TH), Germany, April 2001.

[20] H.-A. Jacobsen and Bernd J. Krämer. Modeling interface definition language extensions. In *IEEE Proceedings of TOOLS Pacific '00, Sydney*, pages 242–252. IEEE Computer Society Press, 2000.

[21] Sun Microsystems Corp., The JAVA homepage. http://java.sun.com/.

[22] S. C. Kleene. Representation of events in nerve nets and finite automata. In C. Shannon and J. McCarthy, editors, *Automata Studies, Annals of Math. Studies 34*, pages 3–40. Princeton, New Jersey, 1956.

[23] Bernd Krämer. Synchronization constraints in object interfaces. In Bernd Krämer, Michael P. Papazoglou, and Heinz W. Schnmidt, editors, *Information Systems Interoperability*, pages 111–141. Research Studies Press, Taunton, England, 1998.

[24] Bernd Krämer and Heinz W. Schnmidt. Types and modules for net specifications. In K. Voss, H. J. Genrich, and G. Rozenberg, editors, *Concurrency and Nets*, pages 269–286. Springer-Verlag, Berlin, Germany, 1987.

[25] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[26] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, New York, USA, 1992.

[27] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.

[28] R. Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[29] Marvin Minsky. *Computation: Finite and infinite machines.* Prentice Hall, Englewood Cliffs, NJ, USA, 1972.

[30] R. J. Nelson. *Introduction to Automata.* Wiley & Sons, New York, NY, USA, 1968.

[31] Microsoft Corp., The .NET homepage. http://www.microsoft.com/net/default.asp.

[32] Oscar Nierstrasz. Regular types for active objects. In *Proceedings of the 8th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-93)*, volume 28, 10 of *ACM SIGPLAN Notices*, pages 1–15, October 1993.

[33] Object Management Group (OMG). The CORBA homepage. http://www.corba.org.

[34] C. A. Petri. Fundamentals of a theory of asynchronous information flow. In *Information Processing 62*, pages 386–391. IFIP, North-Holland, 1962.

[35] F. Plasil, S. Visnovsky, and M. Besta. Bounding component behavior via protocols. In *Proceedings of TOOLS USA '99*, pages 387–398. IEEE, August 1999.

[36] W. Reisig. *Petri Nets: An Introduction.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1985.

[37] Ralf H. Reussner. Dynamic types for software components. In *Companion of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99)*, November 5–10 1999. extended abstract.

[38] Ralf H. Reussner. Enhanced component interfaces to support dynamic adaption and extension. In *34th Hawaiin International Conference on System Sciences*. IEEE, January 3–5 2001.

[39] Ralf H. Reussner. *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Logos Verlag, Berlin, 2001.

[40] Ralf H. Reussner. Counter-contraint finite state machines: A new model for resource-bounded component protocols. In Bill Grosky, Frantisek Plasil, and Ales Krenek, editors, *Proceedings of the 29th Annual Conference in Current Trends in Theory and Practice of Informatics (SOFSEM 2002), Milovy, Tschechische Republik*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, November 2002.

[41] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, MA, USA, 1999.

[42] Arto Salomaa. *Formal Languages*. Academic Press, New York, NY , USA, 1973.

[43] Heinz W. Schmidt and Ralf H. Reussner. Generating Adapters for Concurrent Component Protocol Synchronisation. In *Proceedings of the Fifth IFIP International conference on Formal Methods for Open Object-based Distributed Systems*, March 2002.

[44] Alan C. Shaw. Software descriptions with flow expressions. *IEEE Transactions on Software Engineering*, 4(3):242–254, May 1978.

[45] W. R. Stevens. *TCP/IP Illustrated, Volume 1; The Protocols*. Addison-Wesley, Reading, MA, USA, 1994.

[46] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, Addison-Wesley, Reading, MA, USA, 1998.

[47] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[48] Leslie G. Valiant. Decision procedures for families of deterministic pushdown automata. Research Report CS-RR-001, Department of Computer Science, University of Warwick, Coventry, UK, August 1973.

[49] Leslie G. Valiant. Regularity and related problems for deterministic pushdown automata. *Journal of the ACM*, 22(1):1–10, January 1975.

[50] A. Vallecillo, J. Hernández, and J.M. Troya. Object interoperability. In A. Moreira and S. Demeyer, editors, *ECOOP '99 Reader*, number 1743 in LNCS, pages 1–21. Springer-Verlag, Berlin, Germany, 1999.

[51] A. Vallecillo, J. Hernández, and J.M. Troya. Object interoperability. In *ECOOP '00 Reader*, number 1964 in LNCS, pages 256–269. Springer-Verlag, Berlin, Germany, 2000.

[52] W.M.P. van der Aalst, K.M. van Hee, and R.A. van der Toorn. Component-based software architectures: A framework based on inheritance of behavior. BETA Working Paper Series WP 45, Eindhoven University of Technology, 2000.

[53] D. Yellin and R. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

[54] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.