Formal Foundations of Dynamic Types for Software Components [1]

Ralf H. Reussner
Department of Informatics
Universität Karlsruhe
Germany
reussner@ira.uka.de

March 30, 2000

[1]This document appeared as Interner Bericht (Technical Report) 08/2000 at the Department of Informatics, Universität Karlsruhe, Germany

Abstract

In this work we describe a the foundations of a type system for software components, which supports (1) error checking during composition, (2) automatic adaption of a component's services according to the resources of its surrounding environment, and (3) controlled extension of components by plug-ins. Since the type information is described at the interface of a component, this type system can also be regarded as an new notion for to enhance classical interfaces. This document gives a specification of a component type and describes algorithms performing the above actions (1) – (3).

Contents

Chapter 1

Introduction

Type systems allow static error checking of programs and support the compiler's task calculating the memory layout of the compiled program [Sco99]. Whereas type systems for object-oriented languages are a long lasting concern of research [PS94][Cas97], type systems for component-based programming are relatively new. The main questions tackled in this report are:

1. "What is the type of a software component?"
2. "What are the specific requirements to a type system for software components?"

As the concept of a type system is currently only applied to programming language constructs like variables, functions, or objects, it is clear, that the introduction of the concept type to software components extends the traditional meaning of a type.

1.1 Component-based Software Engineering

In the following we restrict our research in two ways. Firstly, we concentrate on components as defined by [Szy98]:

1. A component is a unit of independent deployment.
2. A component is a unit of third-party composition.
3. A component has no persistent state.

The last point emphasizes *persistent*. This does not mean, that a component has no internal state, like a collection of mathematical functions. No persistent state means, that different copies of a component will have the *same* state, when started in a particular environment. So they will behave in the same way, so usually it will make no sense, to have multiple copies of a component in a single environment. As a consequence, the alias-problem does not arise. The above definition does not conflict with the definition of D'Souza and

Wills, that "Reusable parts that can be adapted, but not modified, are called *components*"[DW99], in case this definition is restricted to (compiled) code.[1]

The second restriction of our research is motivated by the goals we want to support with our type system. According to Nierstrasz, a type should describe the applicability of the typed entity [Nie93], the type of a component is determined by its applicability information, i.e., the information necessary to apply a component. Certainly, a wide range of various issues influence the applicability of a component. We basically concentrate on information necessary to tackle some of the current problems of current component technology. This list is, of course, by no means comprehensive, but shows our focus on protocol-related problems.

1.2 Problems of Current Technology

As a consequence of Szyperski's above definition a software component has to be applicable in various contexts. More concrete, components are deployed through composition into other systems [Wec97]. So during the development and deployment of a component we can separate *compile-time*, *composition-time*, and *run-time*. The *link-time*, when libraries are linked to the component belongs to the compile-time (in case libraries are statically linked to the component), or to the composition time (when libraries belong to the run-time environment are are linked dynamically), or to both.

Error detection

Errors can occur at all three times. It is well known, that not all errors in a program can be found automatically (e.g., by a type system) statically, i.e., in advance, before run-time. This would be a contradiction to the halting-problem. But more important to the praxis is, that many errors can be found statically through a type system during compile time, that is without executing the program. Generally, *type errors* occur when a variable or function is used in an inappropriate context, e.g., a `float`-variable is used as an index in an array (assuming that the `float` will not be casted automatically in a `integer`). Unfortunately, todays component technology (such as the Objects Management Group's CORBA [OMG], Microsft's DCOM [DCO], or Sun Microsystem's EJB [EJB]) does not include a type system, which finds component specific errors during compile- or composition-time. Such component specific errors are, for example, errors due to coupling non-fitting components, errors due to missing but required resources, et cetera. These errors are not detected, when the user composes the components and expects the composition errors. Instead, the errors arise, when actually using the components. This may be long after a change in the system's configuration, and the composition step which caused the errors can be hard to identify.

[1]Whereas D'Souza and Wills explicitly mention parts of models and designs also as components.

We call the problem that errors are not detected before run-time the *'early error detection problem'*.

Component deployment

As mentioned above, components are applied by composition with specific environments (contexts). This is the reason, why manufacturers of components have a non-negligible interest in maximizing the number a component is applicable in. As mentioned elsewhere ([Reu99][RH99]), we identify two basic strategies to maximize this number of contexts a component is applicable in.

(a) *'Adding functionality strategy':* The more functionality a component offer, the more users will be satisfied with this component and will reuse it. (b) *'Dynamic enhancement strategy':* In this approach it is not tried to anticipate during design stage all functionality a component should have in its entire lifetime. Instead of that, certain connectors for further enhancements are defined. When during employment of a component new functionality is required, the component can be enhanced at run-time by so-called 'plug-ins'.

Generally, problems with the 'adding functionality strategy' are: firstly, it is difficult to formulate all requirements of a component in advance. Even when this can be done, a second problem arises: Adding new functionality does not in general improve a software component's reusability, since the added functions translate into new requirements to the environment where the component is to be embedded. Thus, the component becomes less reusable, contrary to the original intention. For example, imagine you are designing a printer management component. If you restrict its functionality to handle only local printers, it will not be very reusable, because it will not handle network printers. However, if you design the component for network printers, it will require a network even for managing the local printer, and that will not make it very popular with users. The above problem we call the *'functionality-reuse problem'*.

Problems with the 'dynamic enhancement strategy' are that today type systems handle this dynamic binding only rudimentarily. Usually a plug-in is a parameterized extra application. Using it as a plug-in just means that it can be launched automatically with correct parameters (e.g., viewers in browsers or file converters in printer controls). Today type systems cannot handle more sophisticated interfaces to plug-ins. Especially, it is not clear how the functionality of a plug-in-component enhances the functionality of the plug-in using component. The functionality of the plug-in does not really appear as new functionality of the plug-in using component at its interface. For example, this would be necessary when a user-interface has to adapt after inserting a plug-in in a component collaborating with the user interface. This is the *'type-extension problem'*.

1.3 Requirements to an Type System for Software Components

According to the problems specific for component based software engineering as mentioned above we pose certain requirements specific for types systems for software components.

Such a type system should support:

Adaption: Elimination of environmental dependencies through adaption of functionality according to the capabilities of the environment. In general, adaption can be achieved (a) by the component's developer, explicitly programming the (possibly many) cases, when the component has to deal with missing resources, or (b) by the run-time system, which automatically adapts the component. Here we demonstrate, how a type system can support this task.

Extension: Defined Interfaces to plug-ins.

Error Detection during composition-time: Detection of composition errors before run-time.

Pre-Compilation: Interface information as replacement for missing contextual information enables pre-compilation.

In this report we focus on the first three requirements. [Heu] and [HR99] shows the application of component types for the pre-compilation problem.

1.4 Overview: A Type System for Software Components and its Benefits

Our solution to the three above mentioned problems ('early error detection problem', 'functionality-reuse problem', and 'type-extension problem') is to add information to the interface of a component.

Concrete, we regard the type of a component consisting of two interfaces: (a) the call-interface, describing the protocol of allowed call sequences and (b) the use-interface, which specifies the required external resources and how these will be used. These interface and the used notation of enhanced finite automata is the content of chapter 2.

The use-interface combined with the call-interface, allows us to detect errors, when a components A wants to use another component B despite of non-fitting protocols. In this case the use-interface of component A will not fit to the call-interface of component B. When additionally combining these two interface of *one* component, component A can adapt its offered services (as described in its call-interface) according to the services its use-interface actually gets from component B. (This algorithm is explained in detail in section 3.2.) So we tackle the 'functionality-reuse problem', since the reuse components with a large functionality is not hindered anymore by missing external components. Component A simply restricts its functionality in a way it can work in an environment with less than the maximum of expected resources. This restriction of functionality may not always lead to a still usable component, but in general the attempt to adapt a component is more useful than just throwing an error.

The 'type-extension problem' is gone on with restrictions a component A imposes a plug-in component B. B can only be plugged in component A, when it fulfills these restrictions. This is described as *type extension* in section 3.3.

Generally this document focuses an a detailed description of our type concept, and the algorithms using this type concept. Pointers to related work are given in [RH99][HR99] and our project's web-site [Reu].

Chapter 2

Types for Software Components

This chapter describes in detail our notion of the type of a component. To do so, first we specify our model of a software component.

Services: All services of a component are function calls. These may be local or remote (e.g., like Javas Remote Method Invocation mechanism [JAV]).

Timing: The execution times of function calls may be different. Is a service called during the execution of another, the service call is queued until the execution of the other call finished.

Ordering: The above queuing of function calls should preserve the order of incoming calls. The order of starting the execution of a function is the order of incoming function calls.

Inheritance: We do not regard any kind of inheritance between components.

Identity: Each component should have a unique identity. (This is the consequence of Szyperskis definition of a component as quoted above in section 1.1. If there exist multiple copies of the same component, these are distinguishable.

So, without looking at the type of a component, a component K is described by S_K, where S_K is the set of services (=function calls) the component offers: $S_K := \{n | n \text{ Name of a public function of K}\}$.

2.1 Enhanced Finite Automata

As explained in section 1.4 we model the usage of a component by the protocol of allowed call sequences. Finite automata has proved useful to model protocols [Hol91]. Finite automata were often deployed to model call sequences for software units[1] [Nie93][YS97]. Note that the definitions of this document are tailored to our purposes and may differ from definitions of text books.

Definition 1 (Deterministic Finite Acceptor with error)
A Deterministic Finite Acceptor (DFA) D is a tuple $D = (I, S, F, E, s_0, \delta)$. I the input alphabet, S the set of states, $F \subseteq S$ is the set of accepting (final) states. $E \subseteq S \backslash F$ is the set of error states, $s_0 \in S$ is a designated *start-state*. $\delta : I \times S \rightarrow S$ is a *total* function. When an error-state is reached, it can never be left: $\forall e \in E.\forall i \in I.\delta(i, e) = e$. Note that the same condition is not true for accepting states. A state $a \in F$ may exist with $\exists i \in I.\delta(a, i) \notin A$.

Definition 2 (Transition Graph of a DFA)
Given a DFA $D = (I, S, F, E, s_0, \delta)$, we can form a directed graph $G := (E, N)$. The set of nodes N is the set of states S. $\langle e_1, e_2 \rangle \in E$ iff $\exists i \in I. \in S.\delta(i, e_1) = e_2$. That is, an edge $\langle e_1, e_2 \rangle$ exits iff an transition from state e_1 to state e_2 exists. According to the graph's offspring of an automata, we also denote edge $\langle e_1, e_2 \rangle$ as $\delta(i, e_1)$ with $i \in I$ appropriate.

Definition 3 (Iterated Transitions)
As a shorthand we introduce $\hat{\delta} : I^* \times S \longrightarrow S$:

$$\hat{\delta}(w, s) := \delta(w_n, \delta(w_{n-1}, \cdots \delta(w_1, s) \cdots)), \tag{2.1}$$

with $w = w_1 \cdots w_n$.

Definition 4 (Accepted Language)
The language $L(D)$ of words accepted by D is defined (as usual), as the set of words over the input alphabet, which bring D (starting in the start-state s_0) in an accepting state: $L(D) := \left\{ w \in I^* | \hat{\delta}(w, s) \in A \right\}$. In case the starting state s_0 is an accepting state, the empty word ϵ is also in $L(D)$: $s \in A \Leftrightarrow \epsilon \in L(D)$.

Note that words of the complement of $L(D)$ (i.e., $L(D)^C = I^* \backslash L(D)$ bring D in an error-state: $\forall w \in I^* \backslash L(D).\hat{\delta}(w, s) \in E$

In the latter we often need a certain construction of sets, which we want to define with a shorthand.

Definition 5 (Cross-Union)
Let A and B be sets, than

$$C = A \cup (A \times B) \tag{2.2}$$

Then we call C the *cross-union* of A and B, in symbols: $C = A \otimes B$.

These definitions serve as a base for our following definitions of the call-interface (section 2.2), with its C-Automaton, and the use-interface (section 2.3), with its F-Automata and EC-Automata.

[1] We use the term *unit* as a placeholder for module, object, or component.

2.2 The Call-Interface

The call-interface models the services a component offers. The interface consists of:

a classical interface: Todays programming languages model interfaces (for *objects*) as a set of function-types. We consider a JAVA `interface` as a classical interface [AG98]. Functions are modeled by the types of their parameters, their return type, and the exceptions they may throw.

additional protocol information: This information specifies the allowed call sequences and is modeled in the *Call-Automaton* (C-Automaton, C-Aut).

In the following we give a basic definition of the C-Automaton, which is enhanced step-by-step, to refine our model of services offered by a component.

The C-Automaton

A straightforward way to model the allowed call sequences of a Component K is to use the K's service names as the input alphabet of its C-Automaton $I_{C-Aut_K} := S_K$. State-transitions correspond to service calls.

Definition 6 (allowed call sequence)
A word $w \in L(\text{C-Aut}_K)$ is an *allowed (valid) call sequence*.

Definition 7 (invalid call sequence)
A word $w' \in I^* \backslash L(\text{C-Aut}_K)$ is called an *invalid call sequence*.

Definition 8 (valid and invalid prefixes)
A prefix $p = w_1 \cdots w_i$ of a word $w = w_1 \cdots w_i w_{i+1} \cdots w_n, i \leq n$ is called a *valid prefix* when there exits a word v with $pv \in L(D)$. That is, when D is fed with p, D can be fed with a word v, so that D reaches an accepting state. If no such v exists, p is an *invalid prefix*.

Definition 9 (Component-Automaton (C-Automaton, C-Aut))
The C-Aut of a Component K is a DFA C-Aut$_K := (I, S, F, E, s_0, \delta)$, with

- the input alphabet I are the names of K's services: $I := S_K$,
- the set of states S chosen appropriate,
- a state a is in the set of accepting (final) states A, iff a sequences of service calls leaves K in a valid state, where no further services calls are necessary to bring K in a valid state,
- is a call f performed which is not a allowed in a given state t, than the result of $\delta(f, t)$ must be an error-state: $\delta(f, t) \in E$. For this purpose one error state is sufficient. The need for a set of error states is justified when dealing with exceptions.

- the start-state s_0 models a start of all allowed call sequences,
- and the transition function δ models calls to K's services. Each call corresponds to a transition. The first call f_i in a call sequence $f = f_1 \cdots f_{i-1} f_i$ which let f become an invalid call sequence and $f_1 \cdots f_{i-1}$ is a valid prefix, must bring $\delta(f_i, t)$ in an error-state. Whereas $\hat{\delta}(f_1 \cdots f_{i-1}, s) \in S \backslash E$. All valid call sequences must bring D in an accepting state.

Figure 2.1 shows an example. A `VideoMail`-component offers various services, such as: **`play`**, **`stop`**, **`pause`**, etc. Transitions leading to an error state are omitted.

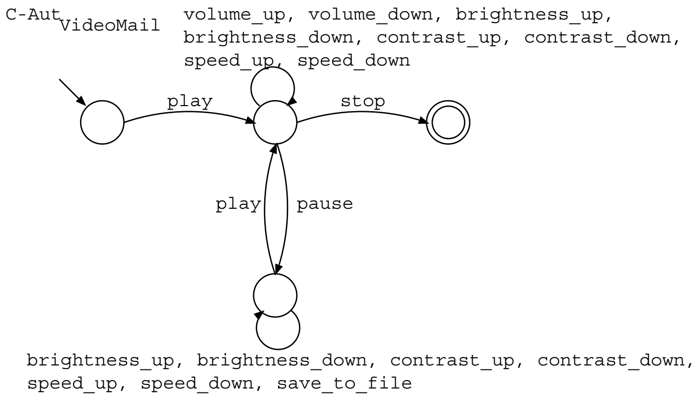
C-Aut VideoMail
volume_up, volume_down, brightness_up, brightness_down, contrast_up, contrast_down, speed_up, speed_down
play
stop
play
pause
brightness_up, brightness_down, contrast_up, contrast_down, speed_up, speed_down, save_to_file

Figure 2.1: Example: C-Aut of a `VideoMail` component.

The $C - Aut_{\text{VideoMail}}$ specifies for example **`play pause volume_up play stop`** as a valid call sequence, whereas **`pause stop`** is an invalid sequence. A valid prefix is **`play speed_down`**, since a call to **`stop`** could bring the C-Aut in an accepting state.

Opposed to common dynamic modeling of objects with state machines (e.g., Statecharts[Har87], SDL[T.84], UML[RJB99]), where the *behavior* of objects is *specified*, we use state machines to *describe* the *availability* of services. That means that states of the C-Aut do not necessarily correspond with internal states of the component. More precisely: several internal states can be subsumed in one state of the C-Aut.

Counters

One problem of the above model is, that there is no possibility to model any relations between calls. A simple but practically important relation between two calls is, that they are called equal times. Or imagine a stack. **`pop`** must not be called more often than **`push`** has been called.

In general, we want to model the allocation, the release, and the creation of resources. Each service of a component may allocate, release, or create arbitrary resources. Resources can be, for example, blocks of memory, file handles, sockets, etc. We would like to check the following conditions:

1. All allocated units of a resource must be released.
2. Not more units of a resource must be allocated than created.
3. Not more units of a resource must be released than allocated. (In this case, it is all right, when a component persistently keeps some units of a resource allocated.)
4. Not more units of a resource must be allocated than released, but the release operation even works for not allocated resources.

To model these conditions we enhance the C-Automaton with counters. We allow three kinds of counters.

Error non-negative counters are integer counters which never store negative integer value. The set $C_{E-\mathbf{N}_0}$ comprises all error non-negative counters. Decreasing a counter $c \in C_{E-\mathbf{N}_0}$ which holds the value zero leads to an error-state.

Idempotent non-negative counters are integer counters which never store a negative value. The set $C_{I-\mathbf{N}_0}$ comprises all error non-negative counters. Decreasing a counter $c \in C_{I-\mathbf{N}_0}$ which holds the value zero has simply no effect.

Integer-counters: These counters represent integer values and are elements of the set $C_{\mathbf{Z}}$.

The set of counters C is defined as: $C := C_{E-\mathbf{N}_0} \cup C_{I-\mathbf{N}_0} \cup C_{\mathbf{Z}}$.

Each function is associated with a set of counters. Those can be either increasing or decreasing. More formally we allow the association of counters to symbols of the input alphabet. We define two *total* functions. The first, Γ_M (Γ-Manipulate), maps an element of the input alphabet to a set of increasing or decreasing counters.

$$\Gamma_M : I \rightarrow \mathbf{P}\left(C \times \{++, --\}\right) \tag{2.3}$$

The symbols $++$ denote an increasing counter, that is we add one to this counter each time the automaton performs this transition. Is no counter manipulated for $a \in I$ then $\Gamma_M(a) := \emptyset$. Counters are initialized to zero before their first usage. The $--$ denotes a decreasing counter, i.e., one is subtracted each time the automaton perfumes this transition *and* the counter is greater than zero. Note that Γ_M need not necessarily be injective. That is several symbols of I may manipulate the same counter.

An symbol a associated with an increasing counter i we write as $a(i++)$, analogously for decreasing counter i we write $a(i--)$.

Γ_M is used to partition the input alphabet:

$$I_{\texttt{C-Aut}_K} := I_{\emptyset} := \{i \in I | \Gamma_M(i) = \emptyset\} \cup I_{\downarrow} := \{i \in I | \Gamma_M(i) \neq \emptyset\} \tag{2.4}$$

We define the set of increasing counters $C_+(i)$ associated with a symbol $i \in I_{\downarrow}$ and the set of decreasing counters $C_-(i)$ associated with a symbol $i \in I_{\downarrow}$.

$$C_+(i) := \{c \in C | c++ \in \Gamma_M(i)\} \tag{2.5}$$
$$C_-(i) := \{c \in C | c-- \in \Gamma_M(i)\} \tag{2.6}$$

As stated below, we require $C_+(i) \cap C_-(i) = \emptyset$. That is, we do not want to increase and decrease the same counter in one step.

Now we define our second total function Γ_T (Γ-Test).

$$\Gamma_T : C \to \{\geq 0, = 0\} \tag{2.7}$$

That is, we associate to each counter either the symbol ≥ 0 or the symbol $= 0$. So we can define the sets

$$\Gamma_{T,\geq 0} := \{c \in C | \Gamma_T(c) = '\geq 0'\} \tag{2.8}$$
$$\Gamma_{T,=0} := \{c \in C | \Gamma_T(c) = '= 0'\} \tag{2.9}$$

This means the automaton accepts iff it is in a final state s and

$$\forall c \in \Gamma_{T,=0}.c = 0 \wedge \forall c \in \Gamma_{T,\geq 0}.c \geq 0 \tag{2.10}$$

is true.

A counter automaton is a DFA with a set of counters C and the functions Γ_M and Γ_T. We now can state our above conditions $1-4$ in terms of our model. The condition 1, that all allocated units of a resource must be released can be modeled by a counter, which is increased for each allocation and decreased for each release. All the time this counter must be greater of equal zero. At the end this counter must be zero. The condition 2 states that never more units of a resource are allocated than are created before. That is for a creation we increase a counter, for an allocation we decrease this counter. This counter must never be negative (but may be positive at the end). The condition 3 says, that never more units of a resource may be released than allocated. Here we increase a counter for each allocation and decrease it for each release. Again, this counter must never be negative. In condition 4, (never more units of a resource allocated than released), we increase a counter for each allocation and decrease it for each release. This counter is an idempotent non negative, i.e., a release operation can also be called, when actually nothing is to release.

Figure 2.2 shows the application of counters when modeling a stack, what corresponds to condition 2.

We want to characterize the languages recognized by counter-automata by enhancing regular expressions. According to the analogy that regular expressions exactly describe the languages finite state machines recognize, we want

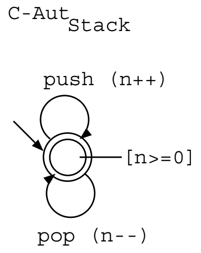

Figure 2.2: C-Automaton of a stack, using counters

to describe the languages recognized by counter automata by an enhancement of regular expression, we call *counter-regular expressions*. Our proceeding is analogous to the theorem of Kleene, that the languages described by regular expressions are exactly the languages recognized by finite state machines.

In the following we pose the following to restrictions to counter automata, to limit their power and ease the task to describe the languages recognized by them.

Same-State Condition (SSC):

$$\begin{aligned} \forall a \in I.\forall c \in C.c \in \mathrm{fst}(\Gamma_M(a)).\,(\exists s_s, s_2 \in S.\delta(a, s_1) = s_2) \rightarrow \\ \exists w \in a^*.\exists s \in S.\hat{\delta}(w, s_2) = s \wedge \delta(a, s) = s \end{aligned} \tag{2.11}$$

Figures (2.4 (a) and (b)) describe allowed state transitions according to the SSC, (c) describes not allowed state transitions according to SSC.

Counter-Manipulation Condition (CMC): if a counter i is decreased(e.g., $\delta(a(i--), s_2) = s_2$), then this counter *must* be increased (that is $\delta(a(i++), s_1 = s_1)$ before (that is on all paths from the start state to the state s). We do not allow increasing and decreasing the *same* counter in a transition, that is it exists no $a \in I.C_+(a) \cap C_-(a) \neq \emptyset$.

The same must hold for the reverse: when a counter is increased (i.e., $\delta(a(i++), s_1) = s_1$ then from all paths from s_s to an end state there must exist an state $s_2 \in S$ where i is decreased in a loop ($\delta(a(i--), s_2) = s_2$). Formally:

$$\begin{aligned} \forall a \in I_{\downarrow}.\,(\exists s_1 \in S.\delta(a, s_1) = s_1) \rightarrow \\ (1)\ \forall c \in C_+(a)\forall \text{ pathes } p \text{ from } s_1 \text{ to a final state } f. \\ \exists i \in I.\delta(i, s_2) = s_2 \wedge s_2 \in p \wedge c \in C_-(i) \\ (2)\ \forall c \in C_-(a)\forall \text{ pathes } p \text{ from the start state to } s_1. \\ \exists i \in I.\delta(i, s_2) = s_2 \wedge s_2 \in p \wedge c \in C_+(i) \end{aligned} \tag{2.12}$$

The CMC is necessary to exclude 'unnecessary' counter manipulations, which would complicate our test on equivalence (section 3.1). The SSC restricts the

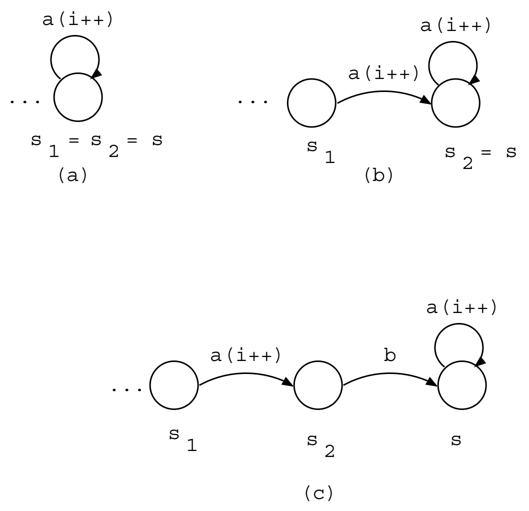

Figure 2.3: Counter application in compliance with (a,b)) and not in compliance with (c) the Same-state Condition.

counter to count only occurrences of *one* symbol. Without the SSC patterns of symbols could also be counted. (What makes the description of the language more complicated.)

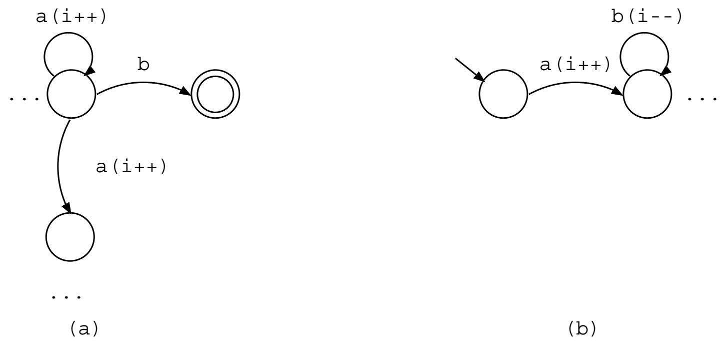

Figure 2.4: Counter application not in compliance with the Counter Manipulation Condition ((a) contrary to eq. 2.12(1), (b) contrary to eq. 2.12(2))

The language recognized by a counter-regular automaton can be characterized by a regular language L_r which satisfies one or more conditions (predicates) regarding the number of occurrences of symbols of the input alphabet. These conditions define a language $L := w \in L_r | P_1(w) \wedge \cdots P_n(w)$, where P_i are predicates as defined below. We associate to each symbol $x \in I$ a function

$x : I^* \to \mathbf{N}$. Function x counts the number of occurrences of symbol x in an arbitrary string over the input alphabet. For each predicate we have to define two index sets $J_l, J_r \subseteq \{1 \cdots |I|\}$. Then we define two functions: $lhs, rhs : I^* \to \mathbf{N}$, $lhs(w) := \sum_{j \in J_l} x_j(w)$ and $rhs(w) := \sum_{j \in J_r} x_j(w)$.

$$lhs(w) \overbrace{\left\{ \begin{array}{c} \geq \\ = \end{array} \right\}}^{A} rhs(w) \wedge \forall p \in \text{prefixes}(w).lhs(p) \overbrace{\left\{ \begin{array}{c} \geq \\ \text{is not related to} \\ \text{is idempotent to} \end{array} \right\}}^{B} rhs(p) \quad (2.13)$$

The letters A and B denote the possibilities for alternatives. The last alternative of B earns an explanation: idempotent occurrences of a symbol x or (a set of symbols X) in respect to occurrences of another symbol y (or set of symbols Y) means, that in a word $w \in I^*$, read from left to right, an occurrence of x (or any $x \in X$) is not counted by function $x(w)$, if $x(w) \geq y(w)$ (or $\sum_{x \in X} x(w) \geq \sum_{y \in X} y(w)$). For example, in $w = yxxy$ is $x(w) = 1$ if x is idempotent to y.

The following table shows how to map above predicates to counter automata, more concrete, how to map to definitions of Γ_M, Γ_T and select the right kind of counter ($\mathbf{Z}$, E-$\mathbf{N}$, I-$\mathbf{N}$). An entry is a number $z_1 z_2$, denoting the z_1th alternative taken in possibility A and the z_2th alternative chosen in possibility B.

	≥ 0	$= 0$
E-**N**	11	21
Z	12	22
I-**N**	13	23

Possibility A decides which end condition for a counter we pose (i.e., Γ_T), possibility B decides which kind of counter we use for this predicate. This allows us to construct the counter automaton out of a set of predicates and L_r. L_r defines a DFA D. For each predicate we introduce a new counter variable j. Possibility B determines whether $i \in C_{E-\mathbf{N}}$, or $i \in C_{\mathbf{Z}}$, or $i \in C_{I-\mathbf{N}}$. For all $x \in I_D$ we add $j++$ to $\Gamma_M(x)$, iff $x \in J_l$, and we add $j--$ to $\Gamma_M(x)$, iff $x \in J_r$. This construction shows, that we can associate to each set of predicates a counter automata. (Note that some combinations of possibilities and cyclic dependencies may lead to a not satisfiable conjunction of predicates, like the condition, e.g., $x_1 \geq x_2 \geq x_1$ in a word $w \in L_r$ and in all prefixes. This is unsatisfiable even when $L_r = I^*$. So $L = \emptyset$.)

The reverse construction (to associate to a counter automaton a set of predicates) requires the CMC and SSC. L_r is determined by δ. For each counter $c \in C$ we define a new predicate P. Possibility A in P is determined whether for c is tested to $= 0$ in a final state f, or to ≥ 0. The kind of c is determined by possibility B. J_l is defined as $J_l := \{i \in I | c \in C_+(i)\}$. Analogously

$J_r := \{i \in I | c \in C_-(i)\}$. Because of the CMC we have for each increasing counter c at least one symbol x, which decreases c. (And for each decreased counter c at least one symbol x, which increases c.) So we can be sure that neither J_l or J_r is empty. Due to the SSC it suffices to look at single symbols and not to strings, since the SSC ensures that only symbols can be counted, not strings.

2.3 The Use-Interface

The use-interface models the external services a components requires and the all possible call sequences to these external services. For each function a component has, a so-called *Function-Automaton* (F-Aut, section 2.3.1) describes this function's calls to other services. Inserting all F-Automata in the C-Aut of a component results in the so-called *enhanced component automata* (EC-Aut, section 2.3.2) of this component, which describes the *component's* calls to other services.

We will see, that the EC-Automaton is constructed in a way, that, given only an EC-Automaton, we can subtract all F-Automata from it, yielding the C-Automaton and all F-Automata. In pseudo-formal notion:

$$\texttt{C-Aut}_K + \{\texttt{F-Aut}_f | f \in S_K\} \leftrightarrows \texttt{EC-Aut}_K \tag{2.14}$$

2.3.1 The F-Automata

As mentioned, a F-Automaton describes a function's calls to external services. Therefore a closure must be created: if function `a()` calls the *internal* function `b()`, then F-Aut$_a$ has to include the external calls of function `b()`, that is the F-Aut$_b$. Of course, the same is valid when constructing F-Aut$_b$.

Definition 10 (Function-Automaton(F-Automaton, F-Aut))
A F-Automaton of a function a is a DFA F-Aut$_a := (I, S, F, E, s_0, \delta)$, with

- the input alphabet I is the transitive closure of a's calls to external services:

$$\begin{aligned} I := \text{externalservices}(a) \quad &:= \quad \{g | f \text{ calls } g \wedge g \text{ extern}\} \\ &\cup \left(\bigcup_{\substack{f \text{ calls } g \wedge \\ g \text{ intern}}} \text{externalservices}(g) \right) \end{aligned} \tag{2.15}$$

- the set of states S chosen appropriate,
- a state a is in the set of accepting states A, iff a the function may return in this state,
- the set of error-states is empty. We define it when we are talking about exceptions.

- the start-state s_0 models the entry point of the function,
- and the transition function δ models calls to external services. Each call corresponds to a transition. All valid call sequences must bring D in an accepting state.

Figure 2.5 shows as an example of the F-Automaton of a the above VideoMail's function `play`.

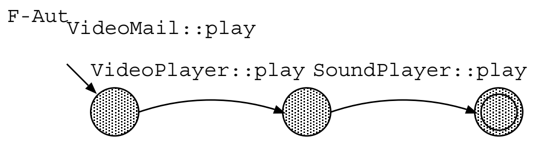

Figure 2.5: F-Automaton of a Function, showing two calls to external function. First calling `VideoPlayer::play` and then `SoundPlayer::play`.

The Or-Semantics Problem

When using a finite automata based notion to model traces, one problem occurs [All97]: what are the semantics of constructions like the one shown in figure 2.6(a). Function `f` may call external function `a` or external function `b`. It is clear, that one of these external functions is necessary, to bring `f` in a final state. But it is not clear, what happens to `f` in the case one of the external functions is missing. Two interpretations are possible: (1) both external function are necessary. If one is missing, function `f` cannot work. (2) if one external function is missing, `f` will not bother, since there is still one path to a final state. To decide, which interpretation is valid for this specification, more information must be supplied. We solve this, in assuming for the construction shown in 2.6(a) the interpretation (1) – both external functions are necessary. The other interpretation is chosen, when a condition is associated to a transition, like shown in figure 2.6(b). In this example, `b` is necessary, but unavailability of `a` does not hinder function `f`.

Actually, this solution is an enhancement of the input alphabet defined in definition 10, formula 2.15. Here we add conditions like the one shown in figure 2.6(b) to the input alphabet.

$$I := \text{externalservices}(a) \otimes \{[\text{ifavail}]\} \qquad (2.16)$$

Modeling Exceptions

Exceptions can be thrown (or raised) in Function-Automata and can be handled in (possibly other) Function-Automata or in the C-Aut. We are only interested in modeling exceptions, which

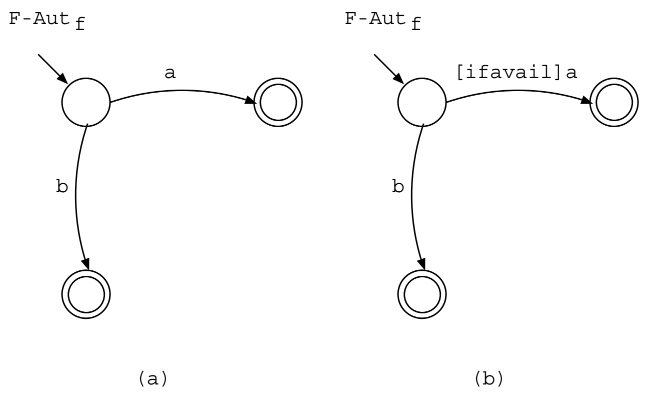

Figure 2.6: Unclear semantic of construction (a). Clarified semantic of construction (b).

- change the availability of services, and so have to be modeled statically in the C-Aut. Besides, we look at exceptions, which
- change the sequences of called external services. An exceptions which only locally affects the control-flow in a F-Aut, and does not effect the availability of services can be modeled in the F-Automaton, where it may be thrown.

So, in the latter, we are only interested in exceptions which effect the availability of services. Since they are raised in a F-Automaton and effect the availability of services, we need a mechanism to transfer information from a F-Aut to the C-Aut. This can be done in the following way:

In the C-Aut a function call is modeled as one transition, but as several transitions: one transition to an 'intermediate state' modeling the call of a function, and several transitions from this intermediate state to other states, modeling the different possible outcomes of this called function: (1) normal outcome, no error, (2) ... (n) exceptions which require different handling in the C-Automaton. An example is shown in figure 2.7.

The F-Automaton has to indicate whether it just returns or an exception was thrown. We use different error-states for exceptions in the F-Aut, which have to be handled appropriate in the C-Automaton (as shown in figures 2.8 and 2.7). Note that several different exceptions of the F-Automaton may be handled identically in the C-Automaton, and so can be all lead to the same error-state in the F-Automaton. Figure 2.8 shows an example of an F-Automaton modeling several exceptions.

The formal notation of an C-Automaton's input alphabet changes. Let $SE_K \subseteq S_K$ denote services of K which may throw exceptions. For each element $f \in SE$ we define a set EX_f as the set of exceptions f may throw.

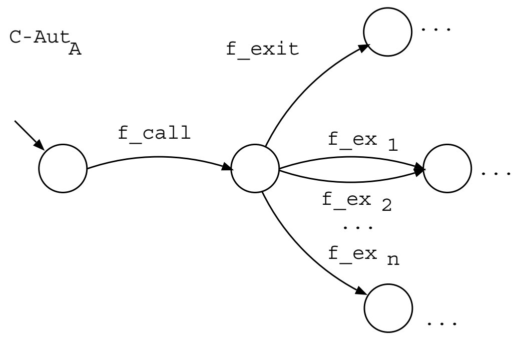

Figure 2.7: Modeling exceptions in the C-Aut.

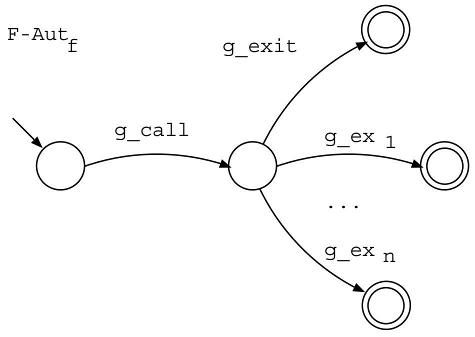

Figure 2.8: F-Aut with several exceptions.

$$I_{\text{C-Aut}_K} := (S_K \backslash SE_K) \cup \left(\bigcup_{f \in SE_K} (\{\text{call}\} \times \{f\} \times (EX_f \cup \{\text{exit}\})) \right) \tag{2.17}$$

First we add the symbol `exit` to each set SE, modeling the exit of a function without an exception thrown.

We need not change the definition of a F-Automaton, except that we now allow several error-states, as described above.

The connection between the F-Automata and the C-Automaton is generalized in the next section.

2.3.2 The EC-Automaton

As described in equation 2.14 the Enhanced Component Automata is build out of the C-Automaton and the F-Automata of a component. Note that the following construction serves as a *model*, that means, an implementation need not necessarily use this construction explicitly.

Definition 11 (Enhanced Component-Automaton)
(EC-Automaton, EC-Aut)
Given a component C C-Aut$_K$ and its F-Automata the EC-Aut$_K$ is given by a DFA EC-Aut$_K$:= $(I, S, F, E, s_0, \delta)$, with

- the input alphabet I:

$$I_{\text{EC-Aut}_K} := \bigcup_{f \in I_{\text{C-Aut}_K}} (\{\text{call}\} \times \{f\} \times (EX_f \cup \{\text{exit}\})) \tag{2.18}$$

- the set of states S and the transition function δ chosen according to the construction algorithm shown below.
- a state a is in the set of accepting (final) states A, iff a sequences of service calls leaves K in a valid state, where no further services calls are necessary to bring K in a valid state,
- is a call f performed which is not a allowed in a given state t, than the result of $\delta(f, t)$ must be an error-state: $\delta(f, t) \in E$. For this purpose one error state is sufficient.
- the start-state $s_{0_{\text{EC-Aut}_K}}$ corresponds to $s_{0_{\text{C-Aut}_K}}$.

Construction of the EC-Automaton

Input: C-Aut$_K$, $\{\text{F-Aut}_f | f \in S_K\}$
Output: EC-Aut$_K$

```
beginning from the start-state
traverse each edge of the C-Aut
for each edge $\delta_{\text{C-Aut}}(f,s)$ do
    add to $S_E C$: s, $S_{\text{F-Aut}_f}$, $\delta_C(f,s)$;
    $\langle$inserting the F-Aut$\rangle$
    $\delta_{\text{EC-Aut}}(f_{\text{call}},s) \leftarrow s_{0_{\text{F-Aut}_f}}$;
    for each s $\in S_{\text{F-Aut}_f}$ do
        for each a $\in I_{\text{F-Aut}_f} \backslash EX_f$ do
            $\delta_{\text{EC-Aut}}(a,s) \leftarrow \delta_{\text{F-Aut}_f}(a,s)$;
        od
        $\langle$connecting exceptions$\rangle$
        for each e $\in EX_f \cup$ f-exit do
            $\delta_{\text{EC-Aut}}(e,s) \leftarrow \delta_{\text{C-Aut}}(e,s)$;
        od
    od
od
```

Figure 2.9 shows a part of the EC-Aut$_{\text{VideoMail}}$ constructed according the above algorithm using the C-Aut$_{\text{VideoMail}}$ (figure 2.1) and the F-Automaton shown is figure 2.5.

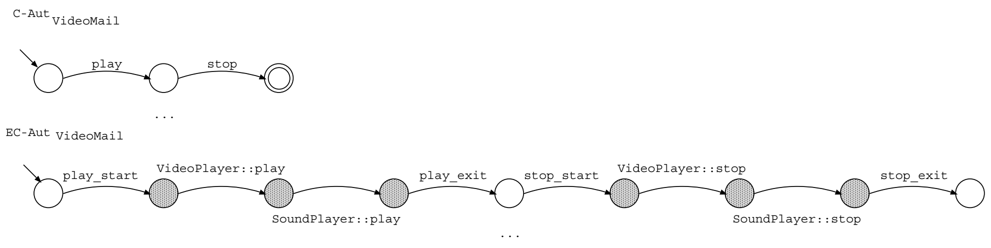

Figure 2.9: EC-Aut of `VideoMail`.

Reconstruction of the C-Automaton

Input: EC-Aut$_K$
Output: C-Aut$_K$

```
beginning from the start-state
traverse the EC-Aut
for each edge $\delta_{\text{EC-Aut}}(f_c all, s)$ do
    create set $EX_f$ of all transitions $\delta_{\text{EC-Aut}}(f_{\text{exception}_i}, s_j)$;
    find transition $\delta_{\text{EC-Aut}}(f_{\text{exit}}, s_e)$;
    $\langle$all states and transitions between belong to F-Aut$_f\rangle$
    $\langle$s is $s_{0_{F\text{-}Aut_f}}$ and $s_e$ its final-state
```

```
        ⟨*states in* $EX_f$ *its error-states.*⟩
        **if** $EX_F$ != 0 **then**
            ⟨*introduce intermediate state, call to it and exceptions*
            *from it and exit from it.*⟩
            add $f_{\text{intermediate}-<f_{\text{counter}}>}$ to $S_C$;
            $\delta_C(f_{\text{call}}, s) \leftarrow \delta_{\text{EC-Aut}}(f_{\text{call}}, f_{\text{intermediate}-<f_{\text{counter}}>})$;
            **for each** transition $\delta(f_{\text{exception}_i}, s_j)$ **do**
                add $s_j$ to $S_C$; $\delta_C(f_{\text{exception}_i}, f_{\text{intermediate}-<f_{\text{counter}}>}) \leftarrow \delta_{\text{EC-Aut}}(f_{\text{exception}_i}, s_j)$;
                add $s_e$ to $S_C$;
                $\delta_C(f_{\text{exit}}, f_{\text{intermediate}<f_{\text{counter}}>}) \leftarrow \delta_{\text{EC-Aut}}(f_{\text{exit}}, s_e)$;
            **od**
            $f_{\text{counter}}$++; ⟨*this counter is used to make*
            *intermediate states unique for each call of function f*
            *(note f is a variable for a function).*⟩
        **else**
            $\delta_C(f, s) \leftarrow \delta_{\text{EC-Aut}}(f_{\text{exit}}, s_e)$;
        **fi**
    **od**
```

Chapter 3

Algorithms and Typing Rules

This section presents several algorithms utilizing the type information we defined in chapter 2 to tackle the problems we mentioned in section 1.2. To each algorithm we show typing rules, describing formally the algorithms actions.

3.1 Type Equality

A basic algorithm for each type system is the check of type equivalence. In our case we use the unique minimal for for finite automata, to test for equivalence of two EC-Automata. We have equal types off the minimized automata are isomorph. The time complexity of this minimization is $\mathbf{O}(|S|^2 \cdot |I|)$. Thus we additionally need a test for isomorphism. Fortunately, in our case, this test can be done efficiently, linear in the number of transitions ($\mathbf{O}(|E_{\text{EC-Aut}}|)$). Since we can limit the number of transitions by $\mathbf{O}(|S|^2 \cdot |I|)$, altogether we have this complexity.

Minimization

The minimization algorithm for DFA's can be found in textbooks on theoretical computer science or compiler construction, e.g., [ASU86]. The notion presented here is adopted from [Hol90]. The idea is that states are equivalent, when they behave equally, i.e., for each symbol of the input alphabet they perform a transition to the same state.

$$s_1 \sim s_2 :\Leftrightarrow \forall i \in I.\delta(i, s_1) = \delta(i, s_2) \tag{3.1}$$

The original algorithm starts with a partition of $S = (F, (S \backslash F))$, that is the first set in the partition are the accepting states, the other set the non-accepting states. The algorithm refines the initial partition further until a fix-point is reached.

Input: an Enhanced DFA: EC-Aut
Output: an equivalent minimized unique Enhanced DFA: EC-Aut'

```
for each partition $p \in P$ do
    new $\leftarrow \emptyset$;
    first $\leftarrow$first state in partition p;
    next $\leftarrow$next state in partition p or false iff there is none;
    for each state $s \in p-$ first do
        $\langle$separate non equivalent states$\rangle$
        for each symbol $i \in I_{\text{EC-Aut}}$ do
            if $\delta(i,s)$ is in another partition as delta(i,first) then
                add s to new;
            fi
        od
    od
    add new to $P$;
od
```

This algorithm separates all non equivalent states. Afterwards each partition contains only equivalent states. So the minimized automaton has the partitions as states (and transitions between states of different partitions as transitions).

To test the equivalence of two states we need $\mathbf{O}(|I|)$ steps, that we have to do for all states, so the complexity for checking all states is $\mathbf{O}(|S| \cdot |I|)$. This we call one *pass* through all states. Unfortunately we have to repeat this test for all states again, when our partition changes. In the worst case, we have no equivalent states, that is we need $\mathbf{O}(|S|)$ new partitions, and we find only one a new partition in a pass. As a result the time-complexity of this algorithm is $\mathbf{O}(|S|^2 \cdot |I|)$.

Informally, this algorithm works, because we do not change either the regular language nor the predicates describing this language. A formal prove follows.

Note that in the above algorithm we do not make use of Γ_M. That is we do not look at counter manipulations at all. Since the set of counters is not changed during minimization, we associate the Γ_T of the original automaton also to the minimized version.

Let $h_{\Gamma_M,\Gamma_T}(\delta) := L(A)$ with $A = (I, S, F, E, s_0, \delta, \Gamma_M, \Gamma_T)$. That is, h_{Γ_M,Γ_T} denotes the language described by a counter-automata, with the input alphabet I, the set of states S, final states F, and error states E are fixed.

Let $h_f(\delta) := L(A)$ with $A = (I, S, F, E, s_0, \delta)$. So h_f denotes the languages accepted by a finite automata $A = (I, S, F, E, s_0, \delta)$.

Let δ' denotes the output of the minimalization algorithm for input δ. Likewise, S' denotes the set of states of the minimized version of $A = (I, S, F, E, s_0, \delta)$. Note, that due to the construction of S' from the last partitioning of S, we can regard S' as a subset of S, since each state $s' \in S'$ can be seen as the representative of a set of the last partitioning, so $s' \in S$. Thus, $S' \subseteq S$.

The correctness of the minimization algorithm for 'ordinary' deterministic finite state machines can be stated now as follows:

$$h_f(\delta) = h_f(\delta') \tag{3.2}$$

For the following we need the definition of a function γ:

Definition 12

$$\begin{aligned} \gamma_T : F &\rightarrow \mathbf{P}\left(C \times \{\geq 0, = 0\}\right) \\ \gamma_T(s) &:= \begin{cases} \bigcup_{c \in C} (c \times \Gamma_T(c)) & \text{if } s \in F \\ \emptyset & \text{otherwise} \end{cases} \end{aligned} \tag{3.3}$$

Let $h_{\Gamma_M,\gamma_T}(\delta)$ denote the set of words $w \in L(A)$, where the counters, manipulated according Γ_M, satisfy all conditions in $\gamma_T(s)$, where $s = \hat{\delta}(w, s_0)$. Due to the construction of γ_T it is

$$h_{\Gamma_M,\Gamma_T}(\delta) = h_{\Gamma_M,\gamma_T}(\delta)$$

To prove the correctness and the uniqueness of the minimization we need the following

Definition 13 (γ_T-Condition)

$$\forall w \in h_{\gamma_M,\gamma_{T_1}}(\delta_1).\gamma_{T_1}(\hat{\delta_1}(w, s_0)) = \gamma_{T_2}(\hat{\delta_2}(w, s_0)) \tag{3.4}$$

Fortunately, this holds when $\delta_2 = \delta_1'$. That is when δ_2 is the minimized transition function of the automaton $A = (I, S, F, E, s_0, \delta)$. Analogously, we need $S_2 = S'$ and

$$\gamma_2 = \gamma_{1|S'} \tag{3.5}$$

To prove this, lets assume

$$\exists w \in h_{\gamma_M,\gamma_{T_1}}(\delta_1).\gamma_{T_1}(\hat{\delta_1}(w, s_0)) \neq \gamma_{T_2}(\hat{\delta_2}(w, s_0)) \tag{3.6}$$

Since $S_2 \subseteq S_1$ (because each element of S_2 can be identified with one element of S_1), the function γ_{T_1} is defined on S_1 *and* S_2, and according to its definition (3.5) $\gamma_{T_1 | S_2} = \gamma_{T_2}$. From that and (3.6) we conclude, $\gamma_{T_1}(\hat{\delta_1}(w, s_0)) \neq \gamma_{T_1}(\hat{\delta_2}(w, s_0))$. This means that the states $\hat{\delta_1}(w, s_0)$ and $[\hat{\delta_2}(w, s_0)]$ are not Γ_T-equivalent. (Otherwise $\gamma_{T_2}(\hat{\delta_2}(w, s_0)) = \gamma_{T_1}(\hat{\delta_2}(w, s_0))$ should equal $\gamma_{T_1}(\hat{\delta_1}(w, s_0))$.) That is a contradiction to the construction of δ_2 out of δ_1, since the (1) last partition of S_1 (namely S_2) is a refinement of $S_1 / \sim_{\gamma_{T_1}}$ and (2) $\forall w \in I^*.\hat{\delta_1}(w, s_0) \in [\hat{\delta_2}(w, s_0)]$.

The basis for the above algorithm was the definition of equivalent states (equation 3.1). Equivalence for counter automata means: two states s_1 and s_2 are equivalent, when (a) for all symbols of the input alphabet all transition lead

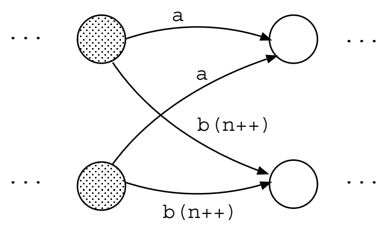

Figure 3.1: From equivalent white states, we can conclude that the shaded states are also equivalent.

to an equivalent state ($\forall i \in I.\delta(i, s_1) \in [\delta(i, s_2)]$), and (b) the same counters are manipulated in the same way. Figure 3.1 shows two equivalent states.

The adoption of this algorithm for our model of extended finite state machines concerns the conditions for a final state to accept, as defined by γ_T. Two states $s_1, s_2 \in S$ are considered as γ_T-equivalent, iff their image under γ_T is the same.

$$s_1 \sim_{\gamma_T} s_2 :\Longleftrightarrow \gamma_T(s_1) = \gamma_T(s_2) \tag{3.7}$$

So γ_T induces the an equivalence relation, which we use to build the factor set $S/\sim_{\gamma_T}$. The initial partition of the minimization algorithm is exactly this factor set.

The following equivalence is a consequence of the γ_T-Condition.

$$h_f(\delta_1) = h_f(\delta_2) \Leftrightarrow h_{\Gamma_M,\Gamma_T}(\delta_1) = h_{\Gamma_M,\Gamma_T}(\delta_2) \tag{3.8}$$

Proof $\Rightarrow$:
We assume that $L(h_f(\delta_1)) = L(h_f(\delta_2))$. Let w be an arbitrary word of $h_{\Gamma_M,\Gamma_T}(\delta_1)$. Since always

$$L(h_f(\delta)) \supseteq h_{\Gamma_M,\Gamma_T}(\delta) \tag{3.9}$$

and due to our premise we have

$$w \in L(h_f(\delta_1)) = L(h_f(\delta_2)) \tag{3.10}$$

From (3.10) we know

$$\hat{\delta_1}(w, s_0) \in F \wedge \forall c \in \Gamma_{T,\geq 0}.c \geq 0 \wedge \forall c \in \Gamma_{T,=0}.c = 0 \tag{3.11}$$

We conclude from (3.10) that $\hat{\delta_2}(w, s_0) \in F$. So from (3.11) and the fact, that the word w generates the same sequence of counter manipulations under δ_1 and δ_2 we also know that

$$\forall c \in \Gamma_{T,\geq 0}.c \geq 0 \wedge \forall c \in \Gamma_{T,=0}.c = 0 \tag{3.12}$$

But this means that $w \in h_{\Gamma_M,\Gamma_T}(\delta_2)$.
Proof $\Leftarrow$:
Here we know that $h_{\Gamma_M,\Gamma_T}(\delta_1) = h_{\Gamma_M,\Gamma_T}(\delta_2)$ and we assume that

$$L(h_f(\delta_1)) \neq L(h_f(\delta_2))$$

The latter implies that (possibly with switching the indices)

$$\underbrace{\exists w \in L(h_f(\delta_1))}_{a}.\underbrace{\neg w \in L(h_f(\delta_2))}_{b} \tag{3.13}$$

Because of (3.9) we know from (3.13b) and our premise that

$$\neg w \in h_{\Gamma_M,\gamma_{T_2}}(\delta_2) = h_{\Gamma_M,\gamma_{T_1}}(\delta_1) \tag{3.14}$$

From (3.13a) we get

$$\hat{\delta_1}(w, s_0) \in F \tag{3.15}$$

But since (3.14)

$$\text{A condition posed in } \gamma_{T_1}(\hat{\delta_1}(w, s_0)) \text{ is false} \tag{3.16}$$

Especially

$$\gamma_{T_1}(\hat{\delta_1}(w, s_0)) \neq \emptyset \tag{3.17}$$

From (3.13b) we get

$$\hat{\delta_2}(w, s_0) \in E \tag{3.18}$$

and, as a consequence, $\gamma_{T_2}(\hat{\delta_2}(w, s_0)) = \emptyset$, which is with (3.18) a contradiction to γ_T-C.

To prove the uniqueness of the minimalization algorithm for counter automata, we assume that two counter automata A_1, A_2 exist, with $\Gamma_{M_{A_1}} = \Gamma_{M_{A_2}} := \Gamma_M$ and $\Gamma_{T_{A_1}} = \Gamma_{T_{A_2}} := \Gamma_T$, and

$$h_{\Gamma_M,\Gamma_T}(\delta_1) = h_{\Gamma_M,\Gamma_T}(\delta_2) \tag{3.19}$$

but $\delta_1' \neq \delta_2'$. From (3.19) we follow, using (3.8), that $h_f(\delta_1) = h_f(\delta_2)$. Using that and the assumption $\delta_1' \neq \delta_2'$ we have a contradiction to the uniqueness of the result of the minimization algorithm.

The correctness of the minimization for counter automata can also be proven by contradiction. Let us assume, that two counter automata A_1, A_2 exist, with $\Gamma_{M_{A_1}} = \Gamma_{M_{A_2}} := \Gamma_M$ and $\Gamma_{T_{A_1}} = \Gamma_{T_{A_2}} := \Gamma_T$, $\delta_1' = \delta_2'$ but $h_{\Gamma_M,\Gamma_T}(\delta_1) \neq h_{\Gamma_M,\Gamma_T}(\delta_2)$. Due to (3.8) it follows $h_f(\delta_1) \neq h_f(\delta_2)$. Using (3.2) we get $h_f(\delta_1') \neq h_f(\delta_2')$. This is a contradiction to our assumption $\delta_1' = \delta_2'$.

Test for Isomorphism

For the test of isomorphism we have to construct a mapping between the states. (We can exploit the fact, that the automata's input alphabets are *identical.*) If and only if we are able to construct a bijective mapping, the automata are isomorph. Fortunately, this can be done by traversing the transitions of one automata, and constructing the mapping step-by-step when visiting an unvisited state. When visiting a visited state, we check check the mapping for bijectiveness in constant time.
Input: two minimized Enhanced DFA's: EC-Aut$_A$, EC-Aut$_B$
Output: yes iff both automata are isomorph, and in case of yes, a permutation $\pi : S_{\text{EC-Aut}_A} \to S_{\text{EC-Aut}_B}$

```
$\pi(s_{0_A}) \leftarrow s_{0_B}$;
beginning from the start-state of EC-Aut$_A$
traverse each state $s \in S_{textEC-Aut_A}$
**for each** transitions $\delta_{\text{EC-Aut}_A}(f, s)$ originating from s **do**
    **if** $\delta_{\text{EC-Aut}_B}(f, \pi(s))$ is undefined **then**
        **return** false;
    **fi**
    **if** $\delta_{\text{EC-Aut}_B}(f, \pi(s))$ is already visited **then**
        **if** $\delta_{\text{EC-Aut}_B}(f, \pi(s))! = \pi(\delta_{\text{EC-Aut}_A}(f, s))$ **then**
            **return** false;
        **fi**
    **else**
        $\pi(delta_{\text{EC-Aut}_A}(f, s)) \leftarrow \delta_{\text{EC-Aut}_B}(f, \pi(s))$;
        mark $\delta_{\text{EC-Aut}_B}(f, \pi(s))$ as visited;
    **fi**
**od**
**if** not all transitions from $\pi(s)$ used **then**
    **return** false;
**fi**
```

3.2 Type Adaption

As motivated by the 'functionality-reuse' problem in section 1.2, type adaption is a mechanism to enhance the reusability of a component through reducing its dependencies to the environment. It is based on the observation that users actually only use a subset of a component's functionality. So a restricted functionality is often sufficient for the user. More important than a full functionality is that the user has not to provide many other infra-structural resources (e.g., libraries, other components, but also system updates, etc.) which a only necessary to support the part of a component's functionality, the user actually does not need.

In terms of our types, when component A adapts to component B (the latter representing the infrastructure), then the C-Automaton of a component

A restricts its functionality to its services which are supported by B. In case B supports all functionality no restriction happens. The like, when A does not need B. In praxis the most interesting and most common case is, when A needs B and C-Aut$_B$ does not offer all functionality A (in form of its EC-Aut$_A$) requires.

Our algorithm for computing the new C-Automaton of A (adapted to B), that is C-Aut$_{A\times B}$, is computed as follows:

1. Compute EC-Aut$_A$ out of C-Aut$_A$ and $\{$F-Aut$_f | f \in S_A\}$. For this we use the algorithm for the construction of an `EC-Aut` given in section 2.3.2.

2. Compute the cross product EC-Aut$_{A\times B}$ out of EC-Aut$_A$ and C-Aut$_B$. That is the intersection $L($EC-Aut$_A) \cap L($C-Aut$_A)$. The algorithm for doing that deviates a little from the common algorithm of cross product construction (e.g., the one given in section 3.4), since we have different kind of input alphabets for EC-Automata and C-Automata. A suitable algorithm is given below.

3. Compute C-Aut$_{A\times B}$ out of EC-Aut$_{A\times B}$ using the $\{$F-Aut$_f | f \in S_A\}$. This algorithm is also shown below.

Algorithm for computing the EC-Aut$_{A\times B}$

Input: EC-Aut$_A$ and C-Aut$_B$.
Output: EC-Aut$_{A\times B}$.

```
$s_{0_{A\times B}} \leftarrow (s_{0_A}, s_{0_B})$;
add $s_{0_{A\times B}} to S_{A\times B}$;


**for each** unvisited state $(s_a, s_b) \in S_{A\times B}$ **do**
    mark $(s_a, s_b)$ as visited;
    **for each** transition $\delta_A(f_{\mathrm{call}_n}, s_a)$ **do**
        **if** not state $(\delta_A(f, s_a), \delta_B(f, s_b)) \in S_{A\times B}$ **then**
            **for each** $f_{e_{n_i}} \in$ (follow($\delta_A(f_{\mathrm{call}}, s_a)$)) **do**
                add unvisited state $(f_{e_{n_i}}, \delta_B(f, s_b))$ to $S_A x B$;
                $\delta_{A\times B}(f, (s_a, s_b)) \leftarrow (f_{e_{n_i}}, \delta_B(f, s_b))$;
            **od**
        **fi**
    **od**
**od**
⟨*identify error-states if possible*⟩
**for each** state $(s_a, s_b) \in S_{A\times B}$ **do**
    **if** $s_a \in E_A$ or $s_b \in E_B$ **then**
        add e to $E_{A\times B}$ ⟨*possibly already in*⟩
        **for each** transition $\delta_{A\times B}(f, s) = (s_a, s_b))$ **do**
            $\delta_{A\times B}(f, s) \leftarrow e$;
        **od**
        fi
    od
```

The function `follow: state s` $\rightarrow$ `set of states` searches for a transition `f_call_n` all states $\delta(\text{f\_ex}_n, s_j)$ and $\delta(\text{f\_exit}, s_j)$ (where s_j is the state from which this transition originates). We use this function `follow` also in the next algorithm.

Algorithm for computing the C-Aut$_{A\times B}$

Input: EC-Aut$_{A\times B}$ and $\{\text{F-Aut}_f | f \in S_A\}$.
Output: C-Aut$_{A\times B}$.

```
    beginning from the start-state
    traverse the EC-Aut
    for each edge $\delta_{\text{EC-Aut}}(f_{\text{call}}, s)$ do
        create set $EX_f$ of all transitions $\delta_{\text{EC-Aut}}(f_{\text{exception}_i}, s_j)$;
        find transition $\delta_{\text{EC-Aut}}(f_{\text{exit}}, s_e)$;
        if (check(F-Aut$_f$, $f_{\text{call}}$)) then
            ⟨all states and transitions between belong to F-Aut$_f$⟩
            ⟨s is $s_{0_{\text{F-Aut}_f}}$ and $s_e$ its final-state,
            states in $EX_f$ its error-states.⟩
            if $EX_F != 0$ then
                ⟨introduce intermediate state,
                call to it and exceptions from it
                and exit from it.⟩
                add $f_{\text{intermediate}-<f_{\text{counter}}>}$ to $S_C$;
                $\delta_C(f_{\text{call}}, s) \leftarrow \delta_{\text{EC-Aut}}(f_{\text{call}}, f_{\text{intermediate}<f_{\text{counter}}>})$;
                for each transition $\delta(f_{\text{exception}_i}, s_j)$ do
                    add $s_j$ to $S_C$;
                    $\delta_C(f_{\text{exception}_i}, f_{\text{intermediate}-<f_{\text{counter}}>}) \leftarrow \delta_{\text{EC-Aut}}(f_{\text{exception}_i}, s_j)$;
                    add $s_e$ to $S_C$;
                    $\delta_{\text{C-Aut}}(f_{\text{exit}}, f_{\text{intermediate}-<f_{\text{counter}}>}) \leftarrow \delta_{\text{EC-Aut}}(f_{\text{exit}}, s_e)$;
                od
                $f_{\text{counter}}$++; ⟨this counter is used to make
                intermediate states unique for each call of function f
                (note f is a variable for a function).⟩
            else
                $\delta_{\text{C-Aut}}(f, s) \leftarrow \delta_{\text{EC-Aut}}(f_{\text{exit}}, s_e)$;
            fi
        fi
    od
```

Function `check F:F-Aut` $\times$ `s:S` $\rightarrow$ `boolean` checks, whether the F-Aut `F` is isomorph to the part of the EC-Aut 'between' `s` and `follow(s)`. In this test the transitions denoted with `ifavail` in `F` are not regarded.

3.3 Type Extension

As mentioned in section 1.2 often the functionality of a component cannot be anticipated completely in advance. So, while in the type adaption mechanism a component exactly describes which functionality it requires from other components to deliver its functionality, in the type extension mechanism a component can be enhanced by the not completely specified functionality of a *plug-in component*.

Although we cannot (and are not willing to) specify the functionality of the plug-in component in advance, we usually do not want to use any plug-in component. Two conditions, when to accept (or reject) a plug-in component are:

Minimal functionality is required. For example in the mail-system example we cannot deploy mails without a function `play`.

Unwanted functionality should be excluded. E.g., we do not want any mail having access to the local file system due to security reasons. (Of course, to employ this types to guard systems against harmful applets, etc. we have to make sure that a component's type *really* describes the component's functionality and used services. To ensure this relation well-known techniques of cryptography can be deployed, like signing the type of a component, hash-functions, etc.[Sch96])

Of course, we want to check these conditions when the user inserts a plug-in component, not later, when the user deploys the functionality of the plug-in. Therefore, both conditions must be expressed and checked in our type-system.

In the following let A be a component, which makes use of a plug-in component P. The *minimal functionality condition* can be described by set inclusion. We add a set of minimal functionality mf_A to the type of A. Actually this set is given by an enhanced finite automaton MF-Aut$_A$, describing a language (L(MF-Aut$_A = \mathrm{mf}_A$). Then we require that

$$\mathrm{mf}_A \subseteq \text{C-Aut}_P \tag{3.20}$$

As described in section 3.4, equation 3.24, the set inclusion can be checked efficiently.

Similarly, the unwanted functionality condition can be stated, by defining a set uf_A (and a relating enhanced automaton UF-Aut$_A$ with L(UF-Aut$_A = \mathrm{uf}_A$). Then we reject every component P which not has

$$\mathrm{uf}_A \nsubseteq \text{EC-Aut}_P \tag{3.21}$$

Note that when A makes use of several plug-in components $P_1 \cdots P_n$ the sets mf_A and uf_A must be indexed additionally by the plug-in component they are relating to: mf_{A,P_i} and uf_{A,P_i}. The same is valid for the corresponding automata MF-Aut$_A$ and UF-Aut$_A$.

The basic idea of type extension is, that the C-Aut$_P$ is inserted into the C-Aut$_A$, which relates to the extension of A's functionality by P's functionality. To do so, we have do define three items:

start$_A \in S_A$ is the state of C-Aut$_A$, to which s_{0_P} is connected to.

Final$_A : F_P \rightarrow S_A$ maps each final states of C-Aut$_P$ to a state of C-Aut$_A$.

Exception$_A : (\cup_{f \in SE_P} EX_f) \rightarrow S_A$ maps each state of C-Aut$_P$ which is reached only by an exception[1] to a state in C-Aut$_A$, where this specific exception is handled by.

The state start$_A$ and the states in dom (Final$_A$) $\cup$ dom (Exception$_A$) are also called *connecting states*, since they are used to connect the automata.

The algorithm to create the extended type C-Aut$_{A+P}$ can be stated with this definitions.

Algorithm for computing the C-Aut$_{A+P}$

Input: C-Aut$_A$ and C-Aut$_P$.
Output: C-Aut$_{A+P}$.

```
$S_{\text{C-Aut}_{A+P}} \leftarrow S_{\text{C-Aut}_A} \cup S_{\text{C-Aut}_P}$;
identify $s_{0_P}$ with start$_A$;
for each state $s \in$ dom(Final$_A$) do
    identify $s$ with Final$_A(s)$;
od
for each state $s \in$ dom(Exception$_A$) do
    identify $s$ with Exception$_A(s)$;
od
for each state $s \in S_{A+B}$ do
    if $s \in S_A$ then
        for each transition $\delta_A(f, s)$ do
            $\delta_{A+P}(f, s) \leftarrow \delta_A(f, s)$;
        od
    fi
    if $s \in S_P$ then
        for each transition $\delta_P(f, s)$ do
            $\delta_{A+P}(f, s) \leftarrow \delta_P(f, s)$;
        od
    fi
od
```

Here we give an example, where the `MailUserAgent` (as shown in figure 3.2) is extended by `VideoMail` $\times$ `VideoPlayer`. The extended C-Aut is shown in figure 3.3. In our example start$_{\text{MailUserAgent}}$ is the gray shaded state. Since dom (Exception$_{\text{MailUserAgent}}$) $= \emptyset$, Exception$_{\text{MailUserAgent}}$ is not defined. Let s_f

[1]Note that no other state can be reached by exception than dom (Exception$_A$).

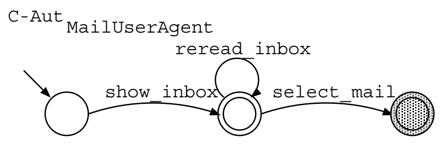

Figure 3.2: The **MailUserAgent** before extension

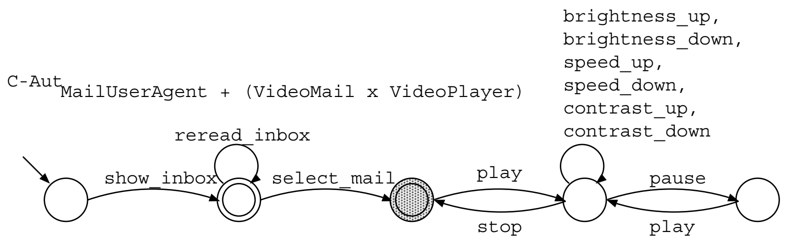
C-Aut MailUserAgent + (VideoMail x VideoPlayer)
reread_inbox
show_inbox
select_mail
play
stop
brightness_up, brightness_down, speed_up, speed_down, contrast_up, contrast_down
pause
play

Figure 3.3: Type extended **MailUserAgent**

denote the only state in $F_{\text{VideoMail} \times \text{VideoPlayer}}$. We define $\text{Final}_{\text{MailUserAgent}}(s_f) := \text{start}_{\text{MailUseragent}} = s_{0_{\text{VideoMail} \times \text{VideoPlayer}}}$

3.4 Subtyping

Like in most type systems, we relate subtyping with substitutability [CW85]. If Component B is a subtype of Component A, $B \trianglelefteq A$, then all occurrences of A can be replaced type safely with B. In the context of our type system that is, (i) B offers at least the services A offers, and (ii) B does not require more external services than A does. We can express conditions (i) and (ii) in terms of languages accepted by our enhanced DFA's, that is, our component type:

$$L(\text{C-Aut}_B) \supseteq L(\text{C-Aut}_A) \wedge \quad (3.22)$$
$$L(\text{EC-Aut}_B) \subseteq L(\text{EC-Aut}_A) \quad (3.23)$$

Fortunately these subset relations are checkable efficiently ($\mathbf{O}(|S|^2 \cdot |I|)$). Note that for arbitrary sets A, B:

$$A \subseteq B \Leftrightarrow A \cap B = A \quad (3.24)$$

This equation gives us an algorithm to test inclusion using the cross-product automata construction (shown below) and isomorphism test algorithm (section 3.1).

As mentioned in section 3.2 the cross-product automaton describes the intersection of two regular languages. The algorithm, given in 3.2 intersects an `EC-Aut` and a `C-Aut`, and has to cover special cases due to the different kinds of input alphabets of these automata (see definition 9, resp. section 2.3.2). In our case here, we have to build the cross-product of two automata, with the kind of input alphabet (namely two EC-Automata).

General Cross Product Construction

This algorithm for cross product construction of DFA's can also be applied to enhanced DFA's, since we regard counters and other extensions as extra symbols of the input alphabet, so that en enhanced DFA can be described by an DFA, not interpreting the counter, etc..

Input: two enhanced DFA A and B.
Output: one enhanced DFA C with $L(C) = L(A) \cap L(B)$.

```
$s_{0_C} \leftarrow (s_{0_A}, s_{0_B})$;
add $s_{0_C}$ to $S_C$;

for each unvisited state $(s_a, s_b) \in S_C$ do
    mark $(s_a, s_b)$ as visited;
    for each transition $\delta_A(f, s_a)$ do
        if not state $(\delta_A(f, s_a), \delta_B(f, s_b)) \in S_C$ then
            add unvisited state $(\delta_A(f, s_a), delta_B(f, s_b))$ to $S_C$;
        fi
        $\delta_C(f, (s_a, s_b)) \leftarrow (delta_A(f, s_a), delta_B(f, s_b))$;
    od
od
⟨identify error-states if possible⟩
for each state $(s_a, s_b) \in S_C$ do
    if $s_a \in E_A$ or $s_b \in E_B$ then
        add $e$ to $E_C$ ⟨possibly already in⟩
        for each transition $\delta_C(f, s) == (s_a, s_b))$ do
            $\delta_C(f, s) \leftarrow e$;
        od
    fi
od
```

The last part eliminates redundant error states (which could in an implementation done during the construction phase and need not require an extra pass). Since we use total transition functions, transitions not representing an valid input lead to an error-state. Sine it is possible that one of the input automata (say A) does fall in an error state (which cannot be left by definition), but B does not fall in an error state, the cross-product contains several states like (e, s_i), where $e \in E_A$ but all s_i are not in E_B. All these states (e, s_i) are error-states of C, and thus can be identified to a single error state $e \in E_C$. This is done by the last part of the algorithm.

The time-complexity of this algorithm mainly depends on the number of transitions to traverse ($\in \mathbf{O}(|S|^2 \cdot |I|)$) and the size of S_C, which is the product

$|S_A| \cdot |S_B|$. In case $|S_A| \approx |S_B|$, both can be bounded by $\mathbf{O}(|S|^2 \cdot |I|)$.

Typing Rules for Subtyping

The following rules describe subtyping for adapted and extended types. Please note, that component A is subtype of component B ($A \trianglelefteq B$) means that A offers at least the services B offers, (as described in equation 3.22). This is equivalent to $B \trianglerighteq A$ what means that B has *less* or equal services compared to A, although B is the supertype of A.

Type Adaption If $a : A$ is already adapted to $b : B$, and be is replaced by a component $c : C$ and C offers possibly less services than b (that is C is a supertype of B), than the newly adapted component `a adapts_to c` may also offer less services than `a adapts_to b`.

$$\frac{a \text{ adapts\_to } b : A \times B \qquad c : C \trianglerighteq B}{a \text{ adapts\_to } c : A \times C \trianglerighteq A \times B} \tag{3.25}$$

Note that in the above equation we used $\trianglerighteq$. Using $\triangleright$ would make this equation invalid, because it is **not** true that from a adapts_to $b : A \times B \wedge c : C \triangleright B$ follows a adapts_to $c : A \times C \triangleright A \times B$. This is because, when a makes no use of the functionality b offers and c does not offer, than a adapts_to $c : A \times C = A \times B$.

As a consequence, we cannot predict what happens in the case, when a adapts_to $b : A \times B \wedge c : C \triangleleft B$ (that is $c : C$ has *more* functionality than b). Two cases are possible:

1. a adapts_to $c : A \times C \triangleleft A \times B$: This happens when b offers not all the functionality a expects, and c offers some (or all) of this functionality expected by a and not realized by b.
2. a adapts_to $c : A \times C = A \times B$: Here, component c offers only functionality not expected (and so not used) by a. In the case b offers all functionality a expects, than no $c : C \triangleleft b : B$ exists with a adapts_to $c : A \times C \triangleleft A \times B$.

An extreme case of equation 3.25 is, that no $b : B$ or $c : C$ exists, where a can adapt to. In the special case, when an unadapted $a : A$ which has to adapt to an other component $b : B$, still offers a meaningful functionality, we know that

$$a : A \triangleright a \text{ adapts\_to } b : A \times B \tag{3.26}$$

In general, we cannot assume that unadapted components offer useful functionality. This clearly depends on the context of application. So generally it makes no sense to talk about $a : A \triangleright A \times B$, since it is not clear whether type A denotes something meaningful.

Type Extension The following rule for type extension is similar to the above rule 3.25.

$$\frac{a \text{ extends\_with } b : A + B \qquad c : C \trianglelefteq B}{a \text{ extends\_with } c : A + C \trianglelefteq A + B} \tag{3.27}$$

As a difference also the stronger version with $\triangleleft$ instead of $\trianglelefteq$ is also valid:

$$\frac{a \text{ extends\_with } b : A + B \qquad c : C \triangleleft B}{a \text{ extends\_with } c : A + C \triangleleft A + B} \tag{3.28}$$

This is because, when extending component a extends_with $b : A$ with a component $c : C$ which is more powerful than component $b : B$, than also a extends_with $c : A + C$ offers more functionality than a extends_with $b : A + B$ (that is $A + C \triangleleft A + B$).

A difference of between the adaption-rule 3.25 and the extension-rules (3.27 and 3.28) is that we adapted a component of type $A \times B$ with a component $c : C$ having less functionality than $b : B$ has. This was because adaption usually means restriction of functionality, and, as we have seen, the other case is more complicated. Opposed to this, in rules 3.27 and 3.28 we handle the case of extending a component of type $A + B$ with a component $c : C$ having *more* functionality than $b : B$. This models the case that type extension exactly means *extension* of functionality.

But we also can give a rule for restricting functionality, when replacing $b : B$ with $c : C$ in a extends_with $b : A + B$ and c having less functionality than b.

$$\frac{a \text{ extends\_with } b : A + B \qquad c : C \trianglerighteq B}{a \text{ extends\_with } c : A + C \trianglerighteq A + B} \tag{3.29}$$

Analogously, the strong version also holds:

$$\frac{a \text{ extends\_with } b : A + B \qquad c : C \triangleright B}{a \text{ extends\_with } c : A + C \triangleright A + B} \tag{3.30}$$

Since unextended types must always define a certain basic functionality, we can state always:

$$a : A \triangleright a \text{ extends\_with } b : A + B \tag{3.31}$$

Covariant Subtypes In the above equations we handled changes in the second argument. The changes in the first argument can be stated in the following rule:

$$\frac{A_1 \trianglelefteq A_2}{A_1 * B \trianglelefteq A_2 * B} \tag{3.32}$$

where $* := \times | +$. That is that when we replace a component of type A_2 with a component by one of its subtypes A_1, than also a component of the type A_1 adapted to or extended with a component of any other type B at least the functionality of A_2 adapted to or extended with a component of type B.

The following rule includes the above cases. So we have a subtype-relation covariant in both arguments:

$$\frac{A_1 \trianglelefteq A_2 \qquad B_1 \trianglelefteq B_2}{A_1 * B_1 \trianglelefteq A_2 * B_2} \tag{3.33}$$

According to the discussion of rule 3.25, the strong version with $\triangleleft$ is not valid, when $* = \times$.

Chapter 4

Future Work

The approach to a type system for components is certainly only a starting point to a new and enhanced understanding of types and extended interface description languages (IDL's). Many points are still unclear or untackled, like:

- The hierarchical (de)composition of components: a natural property of components is, that an assembly of components can be seen itself also as a component. Conversely we should support the decomposition of a component into several other components. Possibly Statecharts[Har87] which allow a hierarchical composition of finite state machines are a good point to start research.

- Synchronization of components: Several components are often using another component simultaneously. Which type information is necessary to derive required points of synchronization?

- Do we want a inheritance of component? Or mere generally, are there more associations between components, than delegation which is typed here?

Bibliography

[AG98] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, Reading, MA, USA, second edition, 1998.

[All97] Robert J. Allen. *A Formal Approach to Software Architecture.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers & Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[Cas97] Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation.* Progress in Theoretical Computer Science. Birkhauser, Boston, 1997.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–521, December 1985.

[DCO] Microsoft Corp., The DCOM homepage. http://www.microsoft.com/com/tech/DCOM.asp.

[DW99] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, Reading, MA., 1999.

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

[Har87] D. Harel. Statecharts: a visuel approach to complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Heu] Dirk Heuzeroth. Eine Software-Architektur für flexible Übersetzer für Sprachen mit modernen Programmierparadigmen. PhD thesis in preparation.

[Hol90] Allen I. Holub. *Compiler Design in C.* Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990. Prentice-Hall Software Series, Editor: Brian W. Kernighan.

[Hol91] Gerald J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[HR99] Dirk Heuzeroth and Ralf Reussner. Dynamic Coupling of Binary Components and its Technical Support. In *First Workshop on Generative and Component based Software Engineering (GCSE) – Young Researchers Workshop*, September 27–30 1999.

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

[Nie93] Oscar Nierstrasz. Regular types for active objects. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices 28(10)*, pages 1–15, October 1993.

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

[PS94] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems.* John Wiley & Sons, Chichester, 1994.

[Reu] Ralf H. Reussner. The DCTS project homepage. http://www.ira.uka.de/˜reussner/dcts/dcts.html.

[Reu99] 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.

[RH99] Ralf H. Reussner and Dirk Heuzeroth. A Meta-Protocol and Type system for the Dynamic Coupling of Binary Components. In *Proceedings of the OOPSLA'99 Workshop on Object Oriented Reflection and Software Engineering*, November 5 1999.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, MA, 1 edition, 1999.

[Sch96] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, Inc, 1996.

[Sco99] Michael Scott. *Programming Language Pragmatics.* Morgan Kaufmann Publishers, San Mateo, CA, 1999.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, Reading, MA, 1998.

[T.84] C. C. I. T. T. Functional Specifications and Description Language (SDL). Rec. z.100-z.104, Geneva, 1984.

[Wec97] Wolfgang Weck. Inheritance using contracts and object composition. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP'97)*, pages 105–12. Turku Centre for Computer Science, September 1997.

[YS97] D. Yellin and R. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.