Automatic Component Adaptation By Concurrent State Machine Retrofitting

Heinz W. Schmidt[1] and Ralf H. Reussner[2]

[1] School of Computer Science and Software Engineering
Monash University, Melbourne, Australia
hws@csse.monash.edu.au
[2] Department of Informatics
Universität Karlsruhe (T.H.), Germany
reussner@ira.uka.de

Abstract. [1] It is a common wisdom of component technology that reuse is not obtained automatically: one has to design for reuse; and reusability has to be preserved as a key quality through design, implementation and maintenance. Besides other technologies aiming at reuse, the component based approach gains increasing attention. Although the idea of reusing prefabricated software components is not new, many obstacles hinder reuse and make it hard to achieve the benefits of reuse in practice.
In general few components are reused as they are. Often, available components are incompatible with what is required. This necessitates extensions or adaptations. In this paper we develop a method assisting the software engineer in identifying the detailed causes for incompatibility and systematically overcoming them. Our method also permits the synthesis of common adapters, coercing incompatible components into meeting requirements.

1 Introduction

It is a common wisdom of component technology that reuse is not obtained automatically: one has to design for reuse; and reusability has to be preserved as key quality through design, implementation and maintenance[BR88]. Successful reuse has been achieved in the area of algorithms and data structures, where common abstractions are agreed and widely understood, through components which provide basic infrastructure for many software projects [MS89,MS96]. A component's source code is not available in general [Szy98]. In practice much of component reuse therefore is black-box use or reuse. Such reuse may include genericity, where a range of pre-designed parameters allow customising the components for the using context in anticipated ways. This permits the management

[1] This report appeared simultaneously as Technical Report No. 2000/81 of the School of Computer Science and Software Engineering, Monash University, Melbourne, Australia and as 'Interner Bericht 25-2000' of the Department of Informatics, Universität Karlsruhe (T.H.), Germany

of product lines, with various individual component configurations and varying component selections and assemblies into an overall product [CE00,Bos00]. Considerable reuse has also been achieved in user interfaces with their various windows, controls, presentations and event mechanisms [Pre95]. On many platforms, such reuse has been achieved using a glass-box approach: The source code is accessible and piecemeal, features of a reused part are inherited, adapted, overridden or replaced. Reuse through implementation inheritance is harder to achieve than black-box reuse and actualisation of generic components. This is due to the opportunistic nature of inheritance: modifications under inheritance are often unplanned-for.

Semantic inheritance lifts this concept to the level of interface specifications, designs and software architectures definitions - usually referred to as *precode*, because such software artefacts usually precede the source code. Semantic inheritance works with black-box reuse because it does not rely on the availability of the source code itself. Sufficient information about the component is abstracted into the precode. *Black-box reuse with semantic inheritance* is therefore associated with higher hopes of being achievable because the semantics of specifications are less complex (abstracting from implementation detail) and more precisely defined (permitting automation and tools). Constraints can thus be put to the potential reuser clearly without the need to prearrange for all relevant customisations. An excellent example for this is the conformance notion arising from design-by-contract, where assertions capture the most relevant architectural and interface features of a reusable component and where conformance represents substitutability under code modification or under change in semantics: the substituted component must at least satisfy all semantic assertions, which hold for the replaced component. And this component in turn is checked for its proper use in all using contexts.

In todays enterprise systems, software is often distributed and multi-threaded. Here reuse has the added problem of inheritance anomalies which arise from the tight coupling of component routines and non-local synchronisation conditions in imperative concurrent code.

Due to these difficulties, required and provided functionality often do not conform or match, when a domain-specific distributed component, in precode, source code or binary, is retrieved for reuse.

Ideally, when the “best match” is still incompatible we would like to be able to identify the matching part clearly. Then we would like to adapt the partially matching component in order to maximise reuse. To date, incompatibility means the start of complex manual work. Only in the simplest cases, such incompatibilities are due to missing functions. More frequently a bunch of functions are tightly intertwined in their behaviour or they are not behaving as expected. For example, they may have undesired behavioural alternatives, exceptions, return values, or require extra synchronisation, or worse, imply some extraneous synchronisation unwanted in the context of reuse.

In this paper we propose an approach for modelling adaptable components. Our components are black-box but carry sufficient information to analyse compatibility at a detailed level and to reuse black-box components partially by hiding or extending their functionality by means of interface adapters. In part such adapters are generated automatically. First, the paper assumes that components are designed using the Unified Modeling Language (UML) [RJB99], specifically state charts. For these we assume a formal semantics is given in terms of finite state machines. Next we assume that software architectures are described formally including configurations of components with distinguished provided and required interface objects. Next, the paper reviews component compatibility for behavioural contracts, analyses incompatibilities in more detail, and, proposes automated and semi-automatic correction of such incompatibilities by adapter generation.

The approach hopes to bring "software *engineering*" to component technology in the sense that it uses scientific methods for repeatably achieved software quality and productivity improvements with a focus on building practical systems on time and within budget through increased automation.

More concretely our approach aims at deriving the following benefits from precode:

1. improved documentation of components by standardised architecture and behaviour definitions (using UML);
2. additional detailed checks of component suitability for a particular reuse context;
3. consequently accompanying facilities for selecting and matching library components to contexts of reuse;
4. detailed (in)compatibility diagnostics in component use and reuse;
5. consequently design and reuse decision support in the sense of "what-if" simulations for hiding, modifying or adding functionality;
6. automatic synthesis of adapters, hiding, modifying or adding functionality to coerce near-match yet incompatible components into compatibility.
7. evaluation and cost-benefits analysis of different alternative competing architectures and designs. This analysis includes measuring the missing components code and missing glue code.

2 Kens and Gates: Component Architectures and Interface Adapters

In distributed systems, besides the separation of interfaces and implementation, also the separation of architecture and interface definition is now widely accepted. Architecture definitions take a mix of black-box and glass-box approach in which successively some interior architectural and configuration aspects are revealed, together with a successive clarification of interfaces and connections. This approach is taken, for instance, in OLAN [BBB$^+$98], or in DARWIN [MDEK95,FS96,RE96a,RE96b], its predecessor [KMN89], and in our own DARWIN extension [Sch98,LSF00]. A general overview over ADLs is given in [JRvdLvdL00]. The separation of architecture and interface definitions goes back perhaps to the mid seventies with work on so-called Module Interconnection Languages (MILs), see e.g., [DK76]. In our methods and tools we termed a self-contained component a *ken*[2] [SC95,Sch98]. Such a composite ken may be hierarchically defined in terms of other more primitive kens. But most importantly it defines a *protection domain* with well defined connections from and to other kens. The ken encompasses a cluster of "internal" objects. It separates them from, and controls their interoperation with, the outside world. The connection control is exercised by so-called *gates*. Gates are interface objects – not just abstractions. They may serve as adapters and controllers not just reflection of component capabilities at runtime. Kens can only be entered via gates, whether (data) objects or control is transferred.

2.1 Gate Behaviour: Recognisers and Generators

Gates permit a black-box approach to kens. For understanding how to enter or interoperate with a ken, it should be sufficient to understand its gates.

Like DARWIN, we distinguish between *required* and *provided* gates. A provided gate describes possible connections to the external world for the purpose of providing a service. A required gate represents possible connections to other components required to perform the services provided.

In our architecture graphs, required gates are connected to provided gates (of other components) to show, as part of the architectural design, the kind of distributed components and their interoperation necessary to perform the overall function of the system.

In contrast to DARWIN ports, each gate lists the signatures of a number of methods and defines a finite state machine (FSM) as the protocol for method calls. For provided gates the FSM can be interpreted as the acceptable call sequences. An example of valid calls to a video-player component may be the sequence `play-pause-play-stop`, whereas the sequence `pause-stop` is commonly not supported. The provided gate FSM is abbreviated by P-FSM for short. For required gates it can be interpreted as an abstraction of the call sequences potentially generated during services provided. For short, the required gate FSM is abbreviated R-FSM.

Current industrial component models, such as Microsoft's (D)COM(+) [DCO], Sun Microsystems' and IBM's EJB [EJB], or OMG's Corba [OMG] model the interface of a component / object as a list of the offered services' signatures. This interface model has several drawbacks. Firstly, since only provided services are modelled one cannot check in advance, whether a component will work in a given environment. Secondly, and perhaps more importantly, the method names and then the existence of corresponding services are only a superficial aspect of a component's behaviour and its interoperability. Some services of a component may only be callable in certain situations. For example, first an initialisation service must be called, before other services are usable. Or one service excludes

[2] English: range of knowledge; Japanese: area (of local autonomy)

the usage of another, or requires the synchronisation with another component. Such constraints form a protocol of the provided services.

The drawbacks of commercial component models and their precursors in research labs gave rise to interface definition including behavioural specification. Automata and automata based calculi have been used widely for protocol verification and testing in telecommunication and real-time component systems[Mil80,KS87,BCG$^+$82,Har87]. With the wide-spread acceptance of UML and its associated use of state charts for interface modelling there is a revived interest in automata based approaches [Nie93,YS94,RH99].

Nierstrasz [Nie93] proposes the modelling of the provided services with a nondeterministic finite state machine. Yellin and Strom describe the protocol of offered and required services in one finite state machine, and use this protocol information to generate adapters [YS97].

In our FSM based approach we wish to take advantage of the rich theory about FSMs on the one hand, but also hide the technical details of the formalism entirely inside our method and tools. Firstly, in order to use our methods, the software engineer does not have to understand the details of the algorithms presented in this paper. More over, in parallel projects at Karlsruhe University we a studying automatic generation of FSM based component interfaces from source code and Message-Sequences-Charts.

2.2 Compatibility

In design-by-contract we distinguish between correctness and conformance. A component implementation is *correct* in relation to its interface contract when it is both *consistent* and *complete*. Roughly, consistency means that two behaviours distinct according to the specification, are distinct in the implementation's behaviour. A trivial example is the distinction between true and false, or that between returning from a call and raising a defined exception. Completeness means roughly, that any behaviour observable according to the specification, is indeed implemented. A simple form of completeness implies that all features listed in the interface are actually implemented; more complex forms of specification require all possible orders of calls permitted according to the specification to be served by the implementation.

Correctness is thus a relation between implementation and interfaces. Quite distinct from correctness, we define conformance as a relation between interfaces of two different components such that either these components can interoperate adequately or one can replace the other. Regarding substitutability, conformance is defined between two instances of the same kind. The conformance between two kens can be reduced to the conformance between their provided gates and that of their required gates. Conformance regarding interoperability is defined for bindings. *Compatibility* finally, extends the above relationships. A component is compatible to its environment if its contracts (more generally its precode) are conformant to a given architectural context, its implementation must be correct, and possibly compliance may entail a number of other aspects including

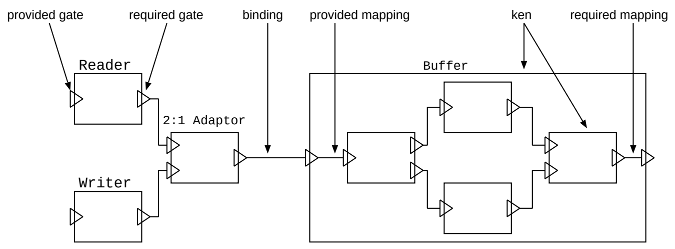


Fig. 1. Example: Kens, Gates, Bindings and Mappings

compliance with standard notations for its precode, standard protocols, possibly domain-specific, etc.

2.3 Ken Architecture and Reconfiguration

In our gray-box approach to kens we show the hierarchical decomposition of kens into lower-level kens and gates. A configuration of sub-kens with their interoperation connections is shown inside the box for the encompassing ken.

This leads to the distinction of gate *mapping* from gate *binding*. When the required gate of a ken is connected to the provided gate of a neighbour ken, this is called a *binding*.

A binding is considered legitimate if the provided gate conforms to the

required gate. Intuitively this means that every call sequence generated by the FSM of the required gate is accepted by the provided gate's FSM. This includes a form of subtyping and thus permits significant variation - again in contrast to DARWIN which always requires identity: the required FSM defines a sublanguage of the provided one - in the sense of formal automata theory.

In contrast to a binding, a *mapping* relates a provided gate of the composite ken to the provided gate of one of its interior kens, or one of its required gates to the required gate of an interior ken.

Because there are many provided and required gates to one ken, conformance under substitution has two forms (in accordance with [FZZ96]):

1. *conformance* demands that
   (a) in each provided mapping, the interior gate conforms to the exterior gate (contravariant conformance);
   (b) in each required mapping, the exterior gate conforms to the interior gate (covariant conformance);
   (c) there may be unmapped interior provided gates;
   (d) there may be unmapped exterior required gates;
2. *partial conformance* is like conformance except that
   (a) there may be unmapped exterior provided gates, if these are not used in the encompassing ken's context;
   (b) there may be unmapped interior required gates, if these cannot be reached from required gates;

Partial conformance is thus context-dependent and requires a more global analysis of at least the immediately adjacent components.

3 Modelling Component Behaviour with Finite State Machines

In the Unified Modeling Language (UML) extended finite state machines (FSMs) [Har87] are used to model the behaviour of objects [RJB99]. An FSM model abstracts from many facets of implementation (source code) behaviour and hence reduces the complexity of reasoning about objects. While we follow the UML notation in our presentation of FSMs, for the purpose of this paper, technically, an FSM consists of the following elements:

a finite set S of states. The system is in exactly one state at any time. The system spends an significant amount of time in each state. A single special state is distinguished as the *initial state* ($s_0 \in S$). There is a non-empty set of distinguished final states ($F \subseteq S$). There also exists a special distinguished *error state* ($e \in S - F$) which, once reached, the system cannot leave. Each state has an associated behaviour, which is described by the following elements.

an alphabet (finite set) I_e of input events. Each element (event) of that set is accepted in at least one state.

an alphabet I_a of actions. An action is triggered by incoming events or before or after transitions from one state to the next.

a transition function t. A transition from a source state s to a target state s' is performed, when an event e occurs. During this transition the action a is fired. An FSM is *deterministic*, when there is at most one transition for each source state and input event. Non-deterministic FSMs occur only as intermediate constructions in our algorithms. We do not support their use in modelling interfaces. This is not a restriction, because every non-deterministic FSM can be converted into a deterministic one. We can model transitions in deterministic FSMs with a transition function. This function t takes as argument an input and the source state and maps that to the target state. Usually actions are regarded as results of a transition. In our approach actions are regarded as inputs for transitions, like events. In none of our FSMs we have transitions associated with an event *and* an action. Hence we can regard events and actions as inputs (I): $I := I_e \cup I_a$. Now, we can define the transition function $t : S \times I \to S$.
Ongoing activities in one state can be modelled by transitions remaining in the same state ($s = s'$). Actions are thus performed when leaving or entering a state. Transitions are (approximately) instantaneous, that is, they take zero time.

The left FSM in Figure 2 may illustrate the graphical notation we use (UML). The states are denoted as circles. State 1 is depicted as initial state (entered by a special circle arrow). State 3 is the (only) final state - indicated by a solid centred circle. A transition from $t(s, e/a)$ is depicted by an arc from state s to state $s' = t(s, e/a)$ inscribed with a label e/a. We only show non-error transitions. All events not shown shown lead to the error state.

For different purposes we use different specialisations of such FSMs. Firstly, so-called *recognisers* omit actions. The left FSM of Fig. 2 is a recogniser. All actions are missing and transitions and traces are just event sequences. Provided gates are modelled this way. For example the above mentioned recogniser describes the supported sequences of calls to a `VideoMail` component. An example for such a supported sequence is `play pause play stop`, whereas `pause stop` is not supported. While this seems simple and clear, one has to provide this information explicitely. E.g., some home video-players support the sequence `play pause pause stop` whilst others do not. Generally the problem is, that while the component changes its state due to method calls, the set of actually supported callable methods changes. We use a *provides-FSM* to describe this protocol. The component is in the initial state when leaving the constructor. Final states leave the component in a state, where the usage of the component may end. When a sequence of method calls drives the FSM in a final state, we call this sequence a *valid sequence.* Is the sequence not supported by the component, the sequences is called *invalid* and leads the FSM in a so called *error state*, i.e., a state, what cannot be left by the FSM. So formally, the provides-FSM is by the tuple:
P-FSM$_K := (I_K, S_K, F_K, e_K, s_{0K}, t_K)$

A FSM without events is a *generator* for action calls driving another component. The behaviour of required gates is modelled in this way. This is described in section 3.2

Hybrid forms where recognition and generation transitions are mixed in one FSM are simply called *translators.* They describe the mapping of inputs to outputs.

3.1 Normalisation

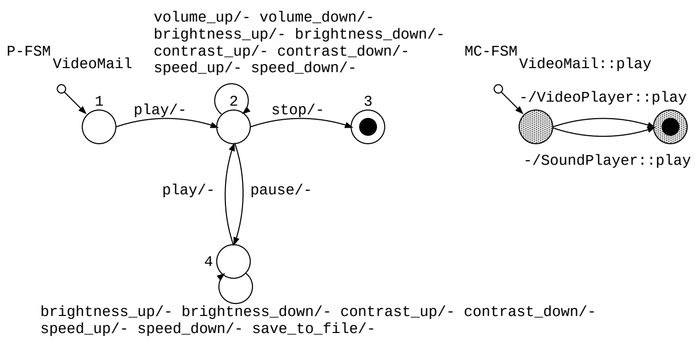
We normalise architecture and interface definitions into a canonical form reducing the complexity of our analysis and synthesis algorithms. These simplifications are purely in terms of the underlying semantics and mechanisms, not at the level of user-defined behaviour models.

For our purposes pure recogniser transitions and generator transitions are sufficient. This leads to a further simplification. We assume the event and action alphabet are disjoint ($I_e \cap I_a = \emptyset$) and hence transitions can be modelled as triples (s, x, t), where the transfer x is either an event or an action symbol.

UML concurrent state machines permit the synchronisation of two FSMs by means of actions emitted by one and recognised by the other. The corresponding pair e/a can always be modelled by two transitions: e followed by a, such that e produces an intermediate state from which a arises as the sole action.

A further simplication normalises connections such that each gate has a unique binding or mapping. In other words a canonical architecture graph does not have multiple connections ending in the same gate. For this purpose a *split-operator* and a *join-operator* is introduced. The former provides a single gate and requires two gates to which incoming calls are dispatched appropriately. The latter, inversely, joins the incoming call streams of two provided gates and channels them to its sole required gate.

With these two operators, all other kens can be normalised in the canonical representation by merging their gates into a single provided and a single required gate by using the *shuffle-FSM* construction defined in a subsequent section. The shuffle-FSM represents all possible interleavings of the original component behaviours.



P-FSM
VideoMail
volume_up/- volume_down/-
brightness_up/- brightness_down/-
contrast_up/- contrast_down/-
speed_up/- speed_down/-
1
play/-
2
stop/-
3
play/-
pause/-
4
brightness_up/- brightness_down/- contrast_up/- contrast_down/-
speed_up/- speed_down/- save_to_file/-
MC-FSM
VideoMail::play
-/VideoPlayer::play
-/SoundPlayer::play

Fig. 2. Examples: Provided FSM of `VideoMail` component (left) and Method FSM of a method (right, shaded states only to ease traceability to Figure 3).

3.2 Component Behaviour: Translators

Each provided method of a component gives rise to a sequence of calls via the required gates. Since we wish to track the causes and effects of binding incompatibilities through a chain of components we need to model the abstract behaviour of such invocations. To this end, additionally to the gate FSMs, we require the user to specify for each provided method of a component a generator FSM, the so-called M-FSM.

In UML based software engineering processes, such method FSMs may occur at the level of detailed design before the actual implementation.

Figure 2 (right) shows as an fictive example of the method FSM of a the above `VideoMail`'s method `play`.

Figure 3 shows as an example of this construction a part of the C-FSM$_{\mathrm{VideoMail}}$ constructed by "inserting" using the P-FSM$_{\mathrm{VideoMail}}$ and the method FSM (both shown in Figure 2).

The following subsection describes the method FSM and the construction of the component FSM in more detail.

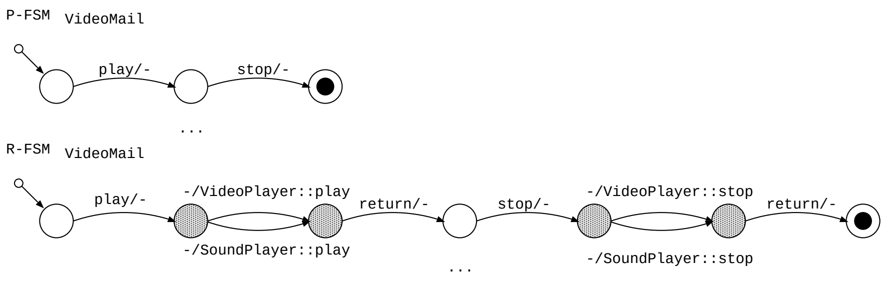


Fig. 3. Required FSM of `VideoMail`.

Therefore a transitive closure must be computed: if method `a()` calls the *internal* method `b()`, M-FSM$_a$ has to include the external calls of method `b()`, that is the M-FSM$_b$. Of course, the same is valid when constructing M-FSM$_b$. The method FSMs are now defined as follows

Definition 1 (Method FSM). *A method FSM (M-FSM) of a method f is a FSM $(I_f, S_f, F_f, e_f, s_{0f}, t_f)$, such that*

- *the input alphabet I is the transitive closure of a's calls to required methods (i.e., methods of the required gate).*
- *a state a is in the set of final states F, iff a the method may return in this state,*
- *and the transition function t models calls to external services. Each call corresponds to a transition. All valid call sequences must bring the M-FSM in an accepting state. Only valid call sequences are modelled.*

Now, given the provided gate (more precisely, the P-FSM of a component) and all the M-FSMs, we are capable of constructing the actual translator for a component. Intuitively, this translator replaces every transition (method invocation) of the P-FSM by inserting a copy of the M-FSM corresponding to the respective method. The resulting translator is called the *component FSM*, short C-FSM.

Algorithm 1 (Construction of the C-FSM)
The easiest way to explain the construction of the C-FSM out of the P-FSM and the M-FSMs is to look at the FSMs as graphs. Then each transition $t(s, \text{method})$ in the P-FSM graph is graphically substituted by the corresponding M-FSM$_{\text{method}}$. A transition labeled with the designated input symbol "return" is drawn from the final state(s) of the inserted M-FSM$_{\text{method}}$ to the state s in P-FSM, which is the result of $t(s, \text{method})$. In terms of this graphical explanation the final states of the C-FSM are only the final states of the P-FSM, not the final states of the inserted M-FSMs.

In [Reu00] detailed algorithms are given for the construction of the required interface out of the provided interface and the method FSM of a component, and for the reconstruction of the provided interface out of the required interface.

This FSM provides a mapping from called methods (events) to emitted sequences of calls to external components (sequences of actions). Each transition is annotated with either an event or an action.

Theorem 1. *The C-FSM, constructed according the above algorithm, models at least all possible sequences of calls to external methods emitted by component K.*

Proof. *Assume there exists a sequence $s := s_1 ... s_n$ of calls to external methods, which can be emitted by K, but is* not *modelled by C-FSM$_K$. Lets denote s_i the last method call in the sequence s, which is also modelled in C-FSM$_K$ and s_{i+1} the first method call in s which is not modelled by C-FSM$_K$. We use proof by contradiction. We have to look at three cases:*

1. *s_i and s_{i+1} are called by the same method m of K:*
   Hence, the partial sequence $s_i s_{i+1}$ should be modelled in M-FSM$_m$. This is in contradiction to the construction of C-FSM$_K$ out of K's M-FSMs.
2. *s_i is called by K's method m and s_{i+1} is called by a method n and n was called (possibly indirectly) by method m of K:*
   Thus, the partial sequence $s_i s_{i+1}$ should be modelled in M-FSM$_m$ because we look at the transitive closure of methods calls performed by m. This is in contradiction to the construction of C-FSM$_K$ out of K's M-FSMs.
3. *s_i is called by K's method m and s_{i+1} is called by a method n (and n is not called by m):*
   Then the methods m and n are called consecutively. Therefore the partial sequence mn is modelled in C-FSM$_K$. According to the consctruction of C-FSM$_K$ the partial sequence $s_i s_{i+1}$ must also be modelled in C-FSM$_K$, what is a contradiction.

All three cases ended in a contradiction, whence C-FSM$_K$ models a superset of all possible call sequences.

Note that the C-FSM$_K$ may model more sequences than possibly emitted in reality by component K. This is because the M-FSM$_K$'s of K also may model a superset of the methods possible external call sequences.

3.3 Component Consistency

Once the component FSM C-FSM is generated, the question arises whether it is is consistent with the required gate FSM R-FSM specified by the software architect. This consistency check boils down to an inclusion test between finite state machines. This inclusion check can be performed by using a more general formula:

$$G - \text{C-FSM} \subseteq \text{R-FSM} \Leftrightarrow G - \text{C-FSM} \cap \overline{\text{R-FSM}} = \emptyset.$$

The negation $\overline{\text{R-FSM}}$ of the required gate FSM denotes the complement state machine, which maps final states into non-final states and vice versa. The intersection of two state machines is a well defined operation (e.g., [Nel68]) and testing equality to the empty set translates into searching for a reachable final state. The complexity of the consistency check lies mainly in the construction of the intersection, which is

$$|S_{\text{R-FSM}}| \cdot |S_{G-\text{C-FSM}}| \cdot min(|I_{\text{R-FSM}}|, |I_{G-\text{C-FSM}}|).$$

4 Adapters

In the following adapter synthesis algorithms, we introduce an algebraic notation to describe different ken configurations. If A and B denote FSMs (for gates or kens), then $A+B$ denotes the shuffle-FSM of A and B. Similarly, $A \rhd B$ denotes the adaptation of A to the required FSM of B. The semantics of these operators will now be defined below.

4.1 1:n-Adapter

In this section we cover the case $A \rhd B + C$. Hence, $\rhd$ is a *split*-operator. It dispatches the calls of A to the right component (B or C). One way to check whether A's required gate fits to the provided gates of B and C (and to possibly adapt A) is to model the services provided by B and C in one single interface.

Problem 1 (1:n Adapter)
Given two provided gates P_1 and P_2, how can one merge their behaviours into a single combined behaviour P.

To solve this problem we construct the *shuffle-FSM* $P_1 + P_2$. The basic idea is, that both provided FSMs can switch states independently. In each state of P_1 all P_2, events acceptable in that state are acceptable in the combined FSM. The converse also holds. The resulting interleaving is modelled exactly by the shuffle language of the provided gates. The formal construction of the shuffle of two FSMs is a well known operation (motivated by shuffle languages [Sha78]) and works as follows.

Algorithm 2 (Construction of Shuffle-FSM)
Given two FSMs A and B the resulting *shuffle-FSM* $A + B = (I, S, F, e, s_0, t)$ is constructed as follows

- the input alphabet I is the union of I_A and I_B. Note that the input alphabets I_A and I_B must be disjoint. This can always be achieved by prefixing the method names with the name of their ken).
- the set of states S is the Cartesian product of the state sets S_A and S_B: $S := \{(s_a, s_b) | s_a \in S_A, s_b \in S_B\}$.
- a state (s_a, s_b) is in the set of accepting states $F \subset S$, iff $s_a \in F_A$ or $s_b \in F_B$.
- in principle, all states (s_a, s_b) are an error state if $s_a = e_A$ or $s_b = e_B$. All these error states can be combined to one error state e.
- the initial state is (s_{0A}, s_{0B}),

– and the transition function $t : S \times I \rightarrow S$ is defined

$$t((s_a, s_b), i) := \begin{cases} (t_A(s_a, i), s_b) \text{ iff } i \in I_A \\ (s_a, t_B(s_b, i)) \text{ iff } i \in I_B \end{cases} \quad (1)$$

Note that the resulting FSM is deterministic, since both FSMs are deterministic and have a disjoint input alphabet.

Lemma 1. *The shuffle-FSM (constructed from A and B) contains all allowed call sequences to a combined interface of A and B.*

Proof. follows from the construction of the transition function of the shuffle-FSM as defined in equation 1.

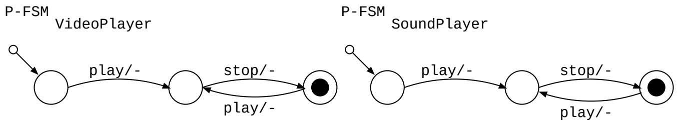


Fig. 4. P-FSM$_{\text{VideoPlayer}}$ (left) and P-FSM$_{\text{SoundPlayer}}$ (right)

Figure 5 shows an example, where the shuffle-FSM of the provided gate of the `VideoPlayer` component (Figure 4, left) and the provided gate of a `SoundPlayer` (Figure 4,right) is shown. Now, for example, we can adapt the functionality of the `VideoMail` (using `VideoPlayer` and `SoundPlayer`) according the functionality of that shuffle FSM. The complexity of this algorithm lies mainly in the definition

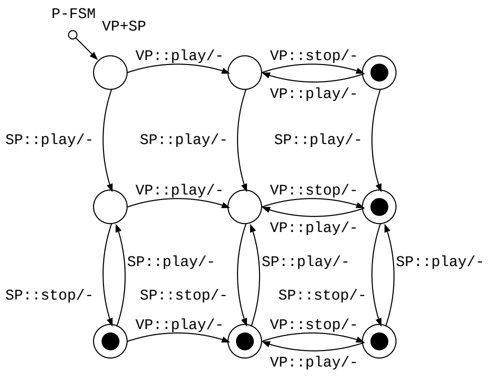


Fig. 5. P-FSM$_{\text{VideoPlayer+SoundPlayer}}$

of transitions. The number of resulting transitions is bounded by

$$|S_A| \cdot |S_B| \cdot (|I_A| + |I_B|).$$

4.2 n:1-Adapter

Fig. 6 shows a producer-consumer system 6. A producer writes to a buffer, then a consumer reads and clears the buffers. The producer can continue writing the next symbol to the buffer. (For sake of brevity, lets assume buffer size 1. This means, producer and consumer communicate using a simple handshake protocol.) It is clear that synchronization between producer and consumer is necessary. The consumer has to wait for the producer to fill the buffer. Likewise, the producer has to wait for the consumer to read and clear the buffer. The task of the join-operator is to automatically find these points of synchronisation.

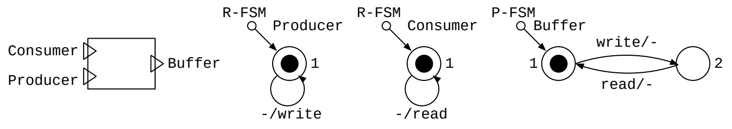


Fig. 6. `Producer` as producer, `Consumer` as Consumer and `Buffer` as input for the join-operator generation.

Problem 2 (Synchronisation)
Given two required gates R_1 and R_2, how can one merge their behaviours into a single combined behaviour R such that

1. conflicting calls exclude each other (calls are conflicting, when they both call the same method of a provided gate.
2. calls from R_1 and R_2 are synchronised relative to a shared provided gate P.

In the following, for the sake of simplicity, we show the C-FSMs without any "return" transitions. In an actual implementation these transitions are regarded as invisible transition (producing the empty word λ) when constructing the shuffle-FSM.

The algorithm to find these synchronization points works as follows:

Algorithm 3

1. Compute the shuffle-FSM $A + B$ as defined in Algorithm 2 from A and B. Note that the input alphabets I_A and I_B are not necessarily disjoint. So, the resulting shuffle-FSM may be non-deterministic. But for later use, we annotate each transition t with the name of the required gate it came from (either A or B) and we refer to that annotation as the *owner* of e. A method of A or B called from an edge e is denoted by method(e). When constructing the shuffle-FSM, we define a mapping $\Pi : S_{A+B} \times I \rightarrow \{S_A \times I_A\} \cup \{S_B \times I_B\}$, which maps each transition of $A + B$ to its originating transition in A or B.
2. Build the intersection FSM of the shuffle-FSM $A + B$ and the provided gate FSM C. The resulting $((A + B) \times C)$ is non-deterministic, iff $A + B$ is non-deterministic.

3. Derive synchronization information from $((A + B) \times C)$ and A:

```
for each path $p$ in $((A + B) \times C)$ from $s_{((A+B)\times C)}$ to an accepting
state do
    ⟨in paths with circles, circle only twice⟩
    $e_{\mathsf{old}}$ ←null;
    for each edge $e \in p$ do
        annotate $\Pi(e)$ with excludes $E(e)$;
        if $e_{\mathsf{old}} \neq$ null then
            if owner($e_{\mathsf{old}}$) ≠ owner($e$) then
                annotate $\Pi(e_{\mathsf{old}})$ with "-,enables $\Pi(e)$";
                ⟨enabling the other transition⟩
                annotate $\Pi(e)$ with "enables$\Pi(e)$,-";
                ⟨waiting on the other transition⟩
            fi
        fi
        $e_{\mathsf{old}}$ ←$e$;
    od
od
```

The set $E(e)$ denotes all edges i from the state where e originates from, having the owner(i) ≠ owner(e) and method(i) = method(e).

The intermediate Producer + Consumer and the (Producer + Consumer) × Buffer are shown in Figure 7. The annotations are given in statechart event syntax

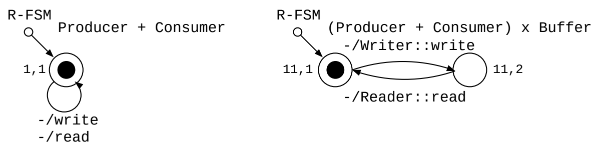


Fig. 7. Intermediate FSM constructions: Producer + Consumer and (Producer + Consumer) × Buffer.

[Har87], as also used for the dynamic models in UML. An annotation `a/b` means that this transition has to wait on event `a` and fires event `b` (when the transition is used, i.e., event `a` arrives). The result of the algorithm are the annotations "`-/enables` C-FSM$_{\text{Producer}}$`::write_1`" and "`enables` C-FSM$_{\text{Consumer}}$`::read_1/-`" for the `read` operation. Both annotations can be combined to "`enables` C-FSM$_{\text{Consumer}}$`::read_1/ enables` C-FSM$_{\text{Producer}}$`::write_1`".

Similarly, the result for the `C-FSM`$_{\text{Producer}}$`:write` operation is "`enables` C-FSM$_{\text{Producer}}$`:write_1/enables` C-FSM$_{\text{Consumer}}$`:read_1`".

To find the dependency between the write- and the read-operation, we have to visit the states in the order 1,2,1,2,1. That is, we have to take the loop twice. To see, why this algorithm solves problem 2, we state

method call annotations	
A:a_1	`excludes B:a_1, -,enables B:b_1`
B:a_1	`excludes A:a_1`
B:b_1	`enable B:b_1,-`

Table 1. resulting annotations.

Lemma 2. *The FSM $((A+B)\times C)$ describes all possible sequences of calls from A and B to the component C.*

Proof. *Analogous to Lemma 1 the FSM $(A+B)$ describes all possible sequences of calls to external methods, which A and B can emit simultaneously. The intersection with the provided gate FSM C restricts $(A+B)$ to the call sequences supported by C.*

Theorem 2. *Algorithm 3 solves the synchronisation problem 2.*

Proof. *From Lemma 2 we know that $((A+B)\times C)$ describes all possible sequences of calls from A and B to the component C. If a state $s \in S_{((A+B)\times C)}$ has several edges i, which are all calling the same method from C then only one call can be performed, that is the other calls are excluded.*

Synchronisation is required between consecutive calls, when the first call is emitted by another component than the second call. These dependencies are detected by traversing all paths (while taking loops only twice). Taking loops only twice suffices to detect in the first cycle the dependencies with in the loop. The second circle detect the dependency between the last and the first statement in the loop.

Note that Algorithm 3 does not resolve conflicting method calls. It just detects conflicting calls. An appropriate resolving strategy might be implemented manually by the programmer, or could be an additional parameter for the adapter generator. While the consumer producer example is a classic, well-known synchronization problem, here it may seem a little degenerated. Therefore we present a more abstract, but complicated example. In our second example components A and B wants to to use component C. We would now like to synchronise A and B calls to C's methods. In Figure 8 we see the C-FSMs of A and B, and the P-FSM of C.

The shuffle FSM of C-FSM$_A$ and C-FSM$_B$ is shown in Figure 9.

Finally, the FSM $(A+B)\times C$ is created to derive the annotations.

As a result, we have the annotations shown in table 1.

The complexity of this algorithm lies mainly in the construction of the shuffle-FSM and the cross product. Both constructions require maximum $|S_A|\cdot|S_B|\cdot \max(|I_A|,|I_B|)$ steps. (Since the input alphabets are overlapping we take their maximum instead of their sum.)

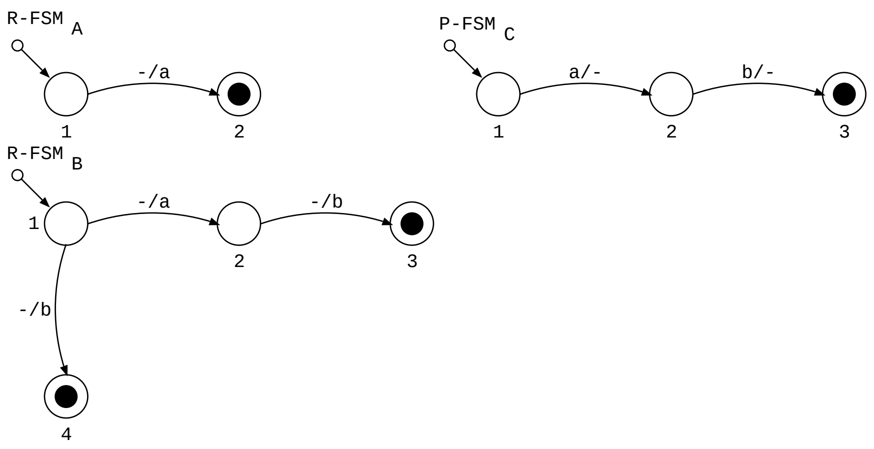


Fig. 8. C-FSM$_A$, C-FSM$_B$, and P-FSM$_C$ as input for the join-operator generation.

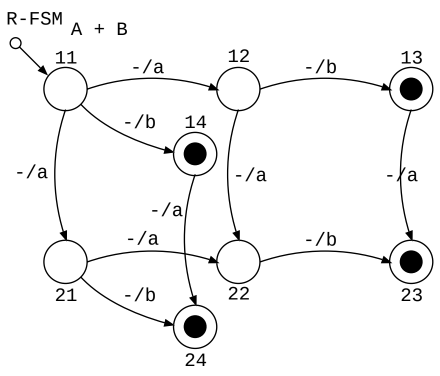


Fig. 9. $A + B$ as an intermediate result during the join-operator generation.

4.3 Protocol Changing Adapters

In the above section we concentrated on the synchronization of two (or in general several) components simultaneously using another component. All using components and the used component were given. We looked for the set of synchronization points (if existing). In this section we tackle the case where one component (A) uses another component (B), but the protocols C-FSM$_A$ and P-FSM$_B$ are not compatible. Because of simplicity, in the latter we refer with C and P to C-FSM$_A$ and P-FSM$_B$. In some cases we can compute a restriction of C's functionality (i.e., adaptation of P-FSM$_A$ [RH99]). But this works only if the intersection of the languages described by C and P is not empty. One interesting case of incompatible protocols (which results in an empty intersection) is that the method $P :: f$ called by C exists in principle, but is not yet ready in

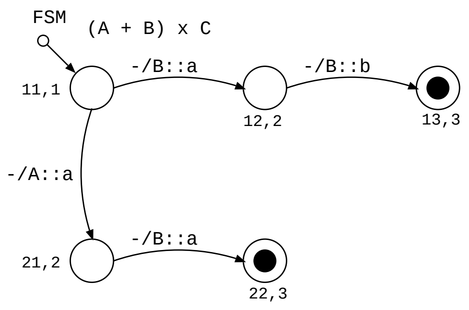


Fig. 10. $(A + B) \times C$ used to generate synchronising events for the join-operator.

the current state of P. Some such protocol incompatibilities can be resolved by 'prefixing' each call to P with a sequence of calls to P. These 'prefix calls' bring P into a state, in which the concerned method of P can be called. For example imagine a required gate of a simple CD player GUI, which only can start, stop, and pause the current CD. Now couple this to a more powerful provided `CDPlayer` gate additionally offering to select one of five CDs, before playing them. The C-FSM$_{\text{SimpleCDPlayer}}$ and the P-FSM$_{\text{CDPlayer}}$ are shown in Figure 11. In this example we need to prefix C-FSM$_{\text{SimpleCDPlayer}}$'s method `play` in state

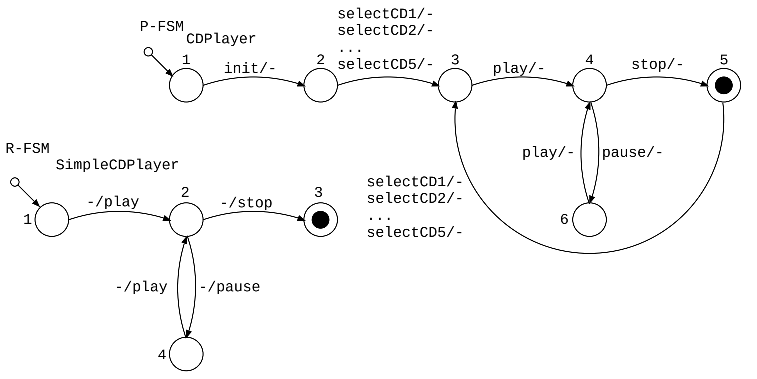


Fig. 11. C-FSM$_{\text{SimpleCDPlayer}}$ (left) and P-FSM$_{\text{CDPlayer}}$ (right)

one with calls to `init` and `selectCDn`. Here we can recognise two simple facts: (a) there may be several different possible prefixes. This ambiguity must be resolved by the programmer (here one might choose `selectCD1` for example). (b) not every call of `play` must be prefixed. Only calls to `play` must be prefixed, when P-FSM$_{\text{CDPlayer}}$ is in state one. (In general the prefix depends on the state of C, the state of P and the method of P to be called). One problem occurs: It is not sufficient to generate a prefix to bring P into a appropriate state (say s_1), where P can handle a call to a method (say m_1). We must also ensure that P in state s_1 can handle all possible sequences of calls to its methods that C can emit after the call to m_1. This clearly restricts the set of prefixes. Using prefixes means that a call to a method of P must first bring P into an appropriate state. After that call, P might be left in this state – yet this state is not a final state. In order to coerce P to move to a final state, some additional postfix transitions need to occur. It is noteworthy, that not all component incompatibilities can be resolved by prefixing or postfixing. A valid prefix or postfix may not exist.

Problem 3 (Initialising / Finalising problem)
Given a C-FSM$_A$ and a P-FSM$_B$, we look for a function `prefix` which given a triplet $(s_c, s_p, \text{method})$ returns a sequence of methods such that: (a) They are called in state s_p to drive P into a state enabling the method. (b) the methods of P that can be called from C after being in state s_c are also supported by P. Furthermore, we require a function `postfix`, which given a triplet $(s_c, s_p, \text{method})$ returns a sequences of method calls such that the sequence starts in s_p and takes P into a final state after method was called by C.

The main step to compute this functions, is to create the so-called *asymmetric shuffle-FSM*. The set of states of this FSM is a subset of the Cartesian product of the state set of C and P. The main idea is that this FSM contains two kinds of transitions: marked and unmarked transitions. Marked transitions go from a state pair (s_c, s_p) with an input i, where in both FSMs i is handled in state s_c (resp. s_p). In an unmarked transition, the input i is only handled in P, but not in C. (Since we do not consider the case, that inputs are accepted in C and not in P, we call this shuffle-FSM asymmetric.) Now we can look for a prefix as a path in this asymmetric shuffle-FSM from a state pair (s_c, s_p) to a marked transition i. Similarly the postfixes are defined as paths from $t((s_c, s_p), i)$ to a final state.

The asymmetric shuffle-FSM of our example is shown in Figure 12. As the re-

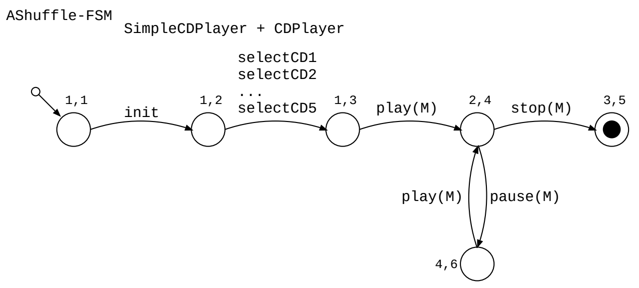


Fig. 12. The asymmetric shuffle-FSM of the `CDPlayer` and the `SimpleCDPlayer`

sult of our example the prefix for `CDPlayer:Play` in state 1 is: `init`, `SelectCD1`.

Note that the selection of the first CD is a choice of the programmer. The algorithm would present all possible CD's here (1–5).

The rest of the section describes the algorithm and argues why it solves the problem.

Before we can state the algorithm, we have to define three predicates. According to Kleene [Kle56], each finite FSM describes a regular language. This language clearly depends on the initial state of the FSM. When not assuming a fixed initial state, we can parameterize the language recognised by an FSM with the initial state. Let $L_C(s)$ denote the language recognised by FSM C, when state s is takes as its initial state. A finite FSM may contain λ transitions, i.e. transitions which do not consume any input symbol (and so are used non-deterministically). The set RL (restricted language in dependence of the initial state) is defined as $RL_{C,P}(s) := L_{C'}(s)$, where A' is the FSM C with every transition $t(s,i)$ replaced with an λ transition $t(s,\lambda)$ iff $i \in I_P$ Now we can state the predicate LC (language contained), used in the algorithm. $LC_{C,P}(s_c, s_p)$ is true iff the language $L_C(s_c)$ is contained in the language $RL_{P,C}(s_p)$.

Algorithm 4 (Construction of the Asymmetric Shuffle-FSM)
Given one component-FSM C and one provides-FSM P the resulting *asymmetric shuffle-FSM* $C \boxplus P = (I, S, F, e, s_0, t, M)$ *is defined as follows*

- the input alphabet I is I_P.
- the set of states S is a subset of the Cartesian product of the state sets S_C and S_P: $S := \{(s_c, s_p) | s_c \in S_C, s_p \in S_P\}$. After creating the transition function (as defined below) one has to check for each state (s_c, s_p) if $L(s_c) \subseteq RL(s_p)$ (predicates also defined below). In case this condition is not true, the state (s_c, s_p) is removed from the state set (and the transition function adapted accordingly).
- a state (s_c, s_p) is in the set of accepting states $C \subset S$, iff $s_c \in F_C$ *and* $s_p \in F_P$ and the predicate $LC_{C,P}(s_c, s_p)$ (defined below) is true. Note that the requirement that both states s_c and s_p are required to be final states. That differs from the definition of the 'symmetric' shuffle-FSM.
- the set of error-states is empty.
- the initial state is (s_{0C}, s_{0P}),
- and the transition function $t : S \times I \to S$ is defined

$$t((s_c, s_p), i) := \begin{cases} (t_C(s_C, i), t_P(s_P, i)) \text{ iff } i \in I_C \wedge i \in I_P \wedge \\ \quad t_C(s_c, i) \neq \text{undefined} \wedge \\ \quad t_P(s_p, i) \neq \text{undefined} \\ (s_c, t_P(s_p, i)) \text{ iff } i \in I_C \wedge t_P(s_p, i) \neq \text{undefined} \wedge \\ \quad t_C(s_c, i) = \text{undefined} \end{cases}$$

- the set M of marked transitions: a transition $t((s_c, s_p), i) \in M \Leftrightarrow i \in I_C \wedge i \in I_P \wedge t_C(s_c, i) \neq$ undefined $\wedge\, t_P(s_p, i) \neq$ undefined

After the construction of this FSM, one may have to remove unreachable or dead states. Now we define the predicates used in the asymmetric shuffle-FSM construction.

Now we can state the function `Set:prefix(`s_c, s_p, i`)` which returns for a state in C and a state in P and for each input symbol $i \in I_C$ a (possibly empty) set of prefixes (method calls) which must be injected in P before method i can be called.

```
prefix $(s_c, s_p, i)$
return {pathes $p \in I_{C \boxplus P}$|
  starting from$(s_c, s_p)$ and ending in $(s'_c, s'_p)$|
  $t_{C \boxplus P}((s'_c, s'_p, i) \neq$ undefined }
```

The function `Set:postfix(`s_c, s_p, i`)` which returns for a *final* state in C and a state in P and for each input symbol $i \in I_C$ a (possibly empty) set of postfixes (i.e. a set of sequences of method calls) which must be injected in P after method i was called to bring P in final state. This function `postfix` is necessary, because when FSM C is in a final state, but P is not, we cannot wait on a next call of a method of P since C is in a final state.

```
postfix $(s_c, s_p, i)$
return {pathes $p \in I_{(C \boxplus P)}$|
  starting from$(s_c, s_p)$ and ending in $(s'_c, s'_p)|(s'_c, s'_p) \in F_{C \boxplus P}$}
```

As specified in the functions `prefix` and `postfix`, we are looking for paths to (resp. from) marked transitions, because a marked transition m originating from a state (s_c, s_p) is supported in state s_c by C, and in state s_p by P. Due to the construction of the asymmetric shuffle-FSM, a path from a state (s'_c, s'_p) to (s_c, s_p) is a sequence of method calls. This sequence must be called in P. It brings P to a state where the transition m is supported by P. (Similar reasoning holds for the `postfix` function).

When selecting a prefix, we must ensure that P-FSM$_P$ in state (s_c, s_p) is still able to accept (with possible further prefixing) all possible sequences, which C-FSM$_C$ can emit in state $t_C((,s)_c, m)$. This is ensured by predicate `LC`. (In fact, LC is to restrictive: the conversion of unknown method calls to λ-transitions only takes prefixes into account, which consist of methods of C-FSM$_C$ not contained in P-FSM$_P$.) This is ensured by predicate `LC`. Putting this together, we yield

Theorem 3. *The functions `prefix` and `postfix` solve the pre- and postfixing problem.*

The complexity of this algorithm lies mainly in the construction of the asymmetric shuffle-FSM and the cross product. Again, both constructions require maximum $|S_C| \cdot |S_P| \cdot \max(|I_C|, |I_P|)$ steps.

5 Conclusions

In this paper we presented a new method for specifying, analysing and adapting component interoperability. To this end we utilised an formal FSM based semantics for component interfaces in combination with an architectural definition of component configurations. While many of our constructions are very technical, the software engineer is not involved with the internal representation and algorithms. Our methods operate largely automatically. Where the software engineer is required to resolve any ambiguities, it is most in terms of the state models, provided by him or her.

Our methods allow us to locate incompatibilities for component bindings. Furthermore, from each black-box component specification, we automatically derive a single behaviour abstraction for a component as a new FSM. This permits various consistency checks but also the computation and simulation of the effects of changes at either the provided or required interfaces of the component in consideration.

We then presented three different kind of adapters to overcome common cases of component incompatibility: (A) One component uses two (or more) other components. (B) Two components simultaneously use a third one. Here the mediating adapter has to perform synchronization between the two using components. (C) One component uses another one but with conflicting protocols. In this case the mediating adapter has to present the functionality of the used component in another (fitting) protocol. For each case (A)–(C) algorithms were presented for the semi-automatic adapter generation. Furthermore the correctness of some of the algorithms was shown.

The approach presented supports an architectural design process oriented towards reuse. The algorithms partly automate design steps and partly support the software architect in decision making.

Open issues are related to: (1) parameter handling: the generation of adapters is semi-automatic; it would be interesting to develop skeleton adaptor generation; also an integration of Yellin and Stroms approach [YS97] is promising. (2) the presented interface model includes signature lists and protocol information (constraints on calling sequences). An extension of that model to include and reason about component qualities is sorely missing.

References

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[BBB+98] R. Balter, L. Bellisard, F. Boyer, M. Riveill, and J.-Y. Vian-Dury. Architecturing and configuring distributed applications with olan. In *Proceedings of IFIP International Conference on Distriubted Systems Platforms and Open Distributed Processing (Middleware)*, pages 15–18, 1998.

[BCG+82] 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.

[Bos00] Jan Bosch. *Design and Use of Software Architectures – Adopting and evolving a product-line approach.* Addison Wesley, Reading, MA, 2000.

[BR88] V. R. Basili and H. D. Rombach. Towards a comprehensive framework for reuse: A reuse-enabling software evolution environment. Technical Report UMIACS-TR-88-92, CS-TR-2158, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, MD 20742, December 1988.

[CE00] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming.* Addison Wesley, Reading, MA, 2000.

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

[DK76] Frank DeRemer and Hans H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, 2(2):80–86, June 1976.

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

[FS96] H. Fossa and M. Sloman. “Implementing Interactive Configuration Management for Distributed Systems”, Proc. Intl. Conf. on Configurable Distributed Systems, Annapolis, Maryland, IEEE, 1996.

[FZZ96] Arne Frick, Walter Zimmer, and Wolf Zimmermann. Konstruktion robuster und flexibler Klassenbibliotheken. *Informatik, Forschung und Entwicklung*, 11(4):168–178, November 1996.

[GS94] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CS-94-166, Carnegie Mellon University, School of Computer Science, January 1994.

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

[JRvdLvdL00] Mehdi Jazayeri, Alexander Ran, Fran van der Linden, and Philip van der Linden. *Software Architecture for Product Families: Principles and Practice*. Addison-Wesley, Reading, MA, 2000.

[Kle56] 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.

[KMN89] J. Kramer, J. Magee, and K. Ng. Graphical configuration programming. *Computer*, 22(10):53–58, 62–65, October 1989.

[KS87] B. Krämer and H. 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 - Heidelberg - New York, 1987.

[KS98] V. Kashyap and A. Sheth. Semantic heterogenity in global information systems: the role of metadata, context, and ontologies. In M. Papazoglou and S. Dcglageter, editors, *Cooperative Information Systems.* Academic Press, 1998.

[LSF00] S. Ling, H.W. Schmidt, and R. Fletcher. Constructing interoperable components in distributed systems. In *IEEE Proceedings of TOOLS Pacific '99, Melbourne*, pages 274–284. IEEE Computer Society Press, 2000.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. *Lecture Notes in Computer Science*, 989:137–155, 1995.

[Mil80] R. Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[MS89] D. R. Musser and A. A. Stepanov. *The Ada Generic Library: Linear List Processing Packages.* Springer Verlag, New York, 1989.

[MS96] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide.* Addison Wesley, Reading, MA, 1996.

[Nel68] R. J. Nelson. *Introduction to Automata.* John Wiley & Sons, New York, NY, 1968.

[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.

[Pre95] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development.* Addison Wesley, Reading, MA, 1995.

[RE96a] M. Radestock, S. Eisenbach. "Formalizing System Structure", In Proc. Int. Workshop on Software Specification and Design, IEEE, pp. 95–104, 1996.

[RE96b] M. Radestock, S. Eisenbach. "Semantics of a Higher-Order Coordination Language", In Proc. Conf. Coordination Languages and Models, Springer, 1996.

[Reu00] Ralf H. Reussner. Formal Foundations of Dynamic Types for Software Components. Technical Report 08/2000, Department of Informatics, Universität Karlsruhe, Am Fasanengarten 5, D-76128 Karlsruhe, Germany, 2000.

[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.

[SC95] H.W. Schmidt and J. Chen. "Reasoning About Concurrent Objects", in IEEE Proc. Asia-Pacific Software Engineering Conf. (APSEC '95), Brisbane, pp. 86–95, 1995.

[Sch98] H.W. Schmidt. "Compatibility of interoperable objects", in: B. Kraemer, M. Papazoglou, H. Schmidt (eds), Information Systems Interoperability Research Studies Press, 1998.

[Sha78] Alan C. Shaw. Software descriptions with flow expressions. *IEEE Transactions on Software Engineering*, 4(3):242–254, May 1978.

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

[YS94] D. Yellin and R. Strom. Interfaces, Protocols and the Semiautomatic Construction of Software Adaptors. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, volume 29:10 of *ACM Sigplan Notices*, pages 176–190, 1994.

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