GENERATING ADAPTERS FOR CONCURRENT COMPONENT PROTOCOL SYNCHRONISATION

Heinz W. Schmidt
School of Computer Science and Software Engineering
Monash University, Melbourne, Australia
hws@csse.monash.edu.au

Ralf H. Reussner
Distributed Systems Technology Center
Melbourne, Australia
rreussner@dstc.monash.edu.au

Abstract In general few components are reused as they are. Often, available components are incompatible with what is required. This necessitates component adaptations or the use of adapters between components. In this paper we develop algorithms for the synthesis of adapters, coercing incompatible components into meeting requirements. We concentrate on adapters for concurrent systems, where adapters are able to resolve synchronisation problems of concurrent components. A new interface model for components, which includes protocol information, allows us to generate these adapters semi-automatically.

Keywords: Component interfaces, interoperability, adapter generation, finite state machines

1. Introduction

One of the anticipated benefits of component oriented software development is, that a component can be deployed in several contexts. Unfortunately, practice shows, that components rarely can be deployed as they are. Usually they have to be adapted to achieve interoperability with the environment. Since components are reused mainly as black-boxes, the component user usually can only perform adaptations as configurations of the component, which must be foreseen by the component developer. But in practice, the component developer cannot foresee in advance all contexts of the component's future deployment. The middleware, in which the components are deployed, acts as a (more or less) transparent medium, while the actual interoperability problem occurs

between components working together via the middleware. One way to solve this interoperability problem is, that the middleware, in which the component is deployed, provides mechanisms to achieve interoperability between components automatically. This interoperability can be realised in two ways: (a) by automatic adaptations of the component (Reussner, 2001) or (b) by generating adaptors, which reside between a component and its environment (Yellin and Strom, 1997).

Adaptor generation requires a detailed information about component interactions. Unfortunately, the interface descriptions of common component models do not include a lot of such information. Whereas the IDL's known form Java or CORBA only provide method signatures (which can be seen a very rough contracts for methods deployment), interactions between methods are not described within these interface models. Especially, there is a demand for information concerning the valid sequences of method calls (Nierstrasz, 1993; Krämer, 1998; Vallecillo et al., 1999; Reussner, 2001; de Alfaro and Henzinger, 2001). For adaptor generation in particular, information about (a) the supported sequences of methods provided by a component and (b) the sequences of calls to external methods, which a component can perform, is required (Yellin and Strom, 1997). These protocols are formed by the application-level components and are independent of the underlying communication protocols (such as TCP/IP, UDP, etc.) In section 2.1 we propose an approach for modelling component interfaces with finite state machines. This enrichment of interface specifications can be used to (semi)-automatically generate adapters for bridging component protocol incompatibilities. Subsection 2.2 reviews the terms compatibility and conformance in the context of architectural connections. Three different kind of adapters to overcome common cases of component incompatibility are described in section 3: (A) One component uses two (or more) other components. (B) Two components simultaneously use a third one. Here the mediating adapter has to perform synchronisation between the two using components. (C) One component uses another one but with conflicting protocols. In section 4 we present related work concerning other architectural component models and protocol specification for components. Section 5 concludes and describes future work. More background information and details about the algorithms presented in section 3 is given in (Schmidt and Reussner, 2000).

2. Kens and Gates: Component Architectures and Interface Adapters

In our methods and tools we termed a self-contained component a *ken* (English: range of knowledge; Japanese: area (of local autonomy) (Schmidt and Chen 1995; Schmidt 1998)). Such a composite ken may be hierarchically de-

fined 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 contain a signature list (like classical interfaces do) and, additionally, a protocol specification, i.e., a description of a set of call sequences, as described below. Gates also may contain specifications of non-functional properties (i.e., quality of services), where appropriate and necessary. (In this paper we do not further pursue analyses using non-functional ken properties.) Using the term "ken" rather than component emphasises the usage of gates instead of classical interfaces.

2.1. Gate Behaviour: Recognisers and Generators

Gates tell us how to enter or interoperate with a ken. In this way, they enable a black-box view of kens.

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

In our architecture description, 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.

Each gate describes a component interface, by listing a method signature and defining the protocol for method calls. The protocol is specified as a finite state machine (FSM). The sequences of method calls accepted by this FSM are call sequences supported by the component.

An example of a valid call sequence 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.

Analogously to the provided gates, the required gates model the required external methods and components and all possible sequences of calls to these external methods. These sequences are again modelled in a FSM.

For short, the required gate FSM is abbreviated R-FSM. In the following we use the common notion of deterministic FSMs (e.g., like used in the UML).

Definition 1 (Finite State Machine)
A FSM $A = (I, S, F, s_0, t)$ *comprises an input alphabet (I), a set of states (S), a set of final states (*$F \subseteq S$*), a start state (*$s_0 \in S$*), and a total transition function* $t : S \times I \to S$.

The P-FSMs input alphabet is the set of methods provided by the component. In the reverse, the input alphabet of the the R-FSM is the set of (external) methods required by the component.

2.2. Compatibility

To introduce the usage of adapters a proper architectural context, first we have to discuss how kens are related to their environment. Therefore, we review the terms compatibility and conformance.

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

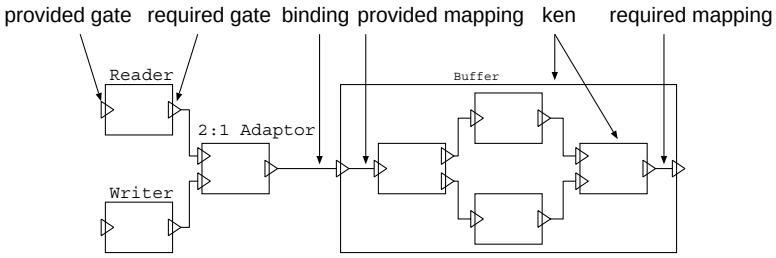

Figure 1. Example: Kens, Gates, Bindings and Mappings

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 num-

ber of other aspects including compliance to standard notations for its precode, standard protocols, possibly domain-specific, etc.

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* (see fig. 1). 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 (Frick et al., 1996)): *Conformance* demands that

1. in each provided mapping, the interior gate conforms to the exterior gate (contravariant conformance);
2. in each required mapping, the exterior gate conforms to the interior gate (covariant conformance);
3. there may be unmapped interior provided gates;
4. there may be unmapped exterior required gates;

A weaker variant is *partial (or context-dependent) conformance* which is like conformance except that

1. there may be unmapped exterior provided gates, if these are not used in the encompassing ken's context;
2. there may be unmapped interior required gates, if these cannot be reached from required gates;

Hence, when ken A is to replace ken B one has to check (a) whether the provides interface of A includes the provides interface of B and (b), whether the requires interface of A is included in the requires interface of B. Checking whether A is bound correctly to B (i.e., whether A can use B) is realised by testing whether the requires interface of A is included in the provides interface of B.

These tests are applicable in general; they must hold for normal signature-based interface models as for protocol-modelling interfaces. For the latter, the inclusion of the languages, described by the interface FSMs must be checked.

3. Adapters

Most commonly synchronisation problems arise (a) when one component uses two other components or (b) through concurrent use of a component by two other components. For this purpose a *split-operator* (handling case (a)) and a *join-operator* (for case (b)) is introduced. The former provides a single gate and requires two gates to which incoming calls are dispatched appropriately. The latter, conversely, joins the incoming call streams of two provided gates and channels them 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. These adapters also lead to an architectural simplification. This simplification normalises connections such that each gate has a unique binding or mapping.

The last adapter we present adapters for a specific class of protocol incompatibilities where a component's protocol needs some extra methods calls which are not performed by the using component.

3.1. 1:n-Adapter

In this subsection we describe the generation of an adapter between a ken A and two kens B and C which are used by A. The adapter dispatches the calls of A to the right ken (B or C). The benefit of this adapter is the presentation of B's and C's provides interfaces in a single interface. That allows interoperability checking, like described in section 2.2.

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 apply the construction of the *shuffle-FSM* $P_1 + P_2$. The concept of a shuffle-FSM is, that both constituent FSMs (in our case the P-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 (Shaw, 1978)). Hence, the general concept of a shuffle-FSM is applicable to our problem. Applied to our problem, the constrcution works as follows.

Algorithm 1 (Construction of Shuffle-FSM)
Given two FSMs A and B the resulting shuffle-FSM $A + B = (I, S, F, s_0, t)$ *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 $s_b \in F_B$.*
- *the start-state is (s_{0A}, s_{0B}),*
- *and the transition function $t : S \times I \to S$ 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. From the construction of the transition function of the shuffle-FSM as defined in equation 1 it follow:

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

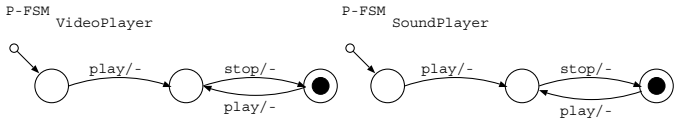

Figure 2. P-FSM$_{\text{VideoPlayer}}$ (left) and P-FSM$_{\text{SoundPlayer}}$ (right)

Figure 3 shows an example, where the shuffle-FSM of the provided gate of the `VideoPlayer` ken (Figure 2, left) and the provided gate of a `SoundPlayer` (Figure 2,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 of transitions. The number of resulting transitions is never larger than $|S_A| \cdot |S_B| \cdot (|I_A| + |I_B|)$. The algorithm produces an `1:2`-adaptor, which can be applied associatively multiple times to create an `1:n`-adaptor.

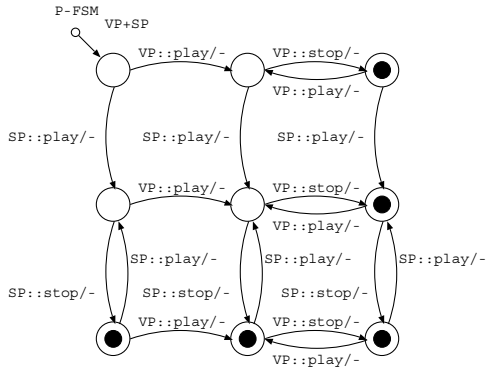

Figure 3. P-FSM$_{\text{VideoPlayer+SoundPlayer}}$

3.2. n:1-Adapter

Fig. 6 shows a producer-consumer system 4. 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 synchronisation 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.

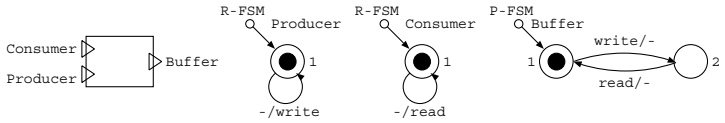

Figure 4. `Writer` as producer, `Reader` as Consumer and `Buffer` as input for the join-operator generation.

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.*

The algorithm to find these synchronisation points works as follows:

Algorithm 2

1. *Compute the shuffle-FSM $A + B$ as defined in algorithm 1 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 synchronisation information from $((A + B) \times C)$ and A:*

```
for each path p in ((A + B) × C) from s_((A+B)×C) to an accepting state
do
    ⟨in paths with circles, circle only twice⟩
    e_old ←null;
    for each edge e ∈ p do
        annotate Π(e) with excludes E(e);
        if e_old ≠ null then
            if owner(e_old) ≠ owner(e) then
                annotate Π(e_old) with "-, enables Π(e)";
                ⟨enabling the other transition⟩
                annotate Π(e) with "enables Π(e),-";
                ⟨waiting on the other transition⟩
            fi
        fi
        e_old ←e;
    od
od
```

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

The intermediate Writer + Reader and the (Writer + Reader) × Buffer are shown in Figure 5. The annotations are given in statechart event syntax (Harel,

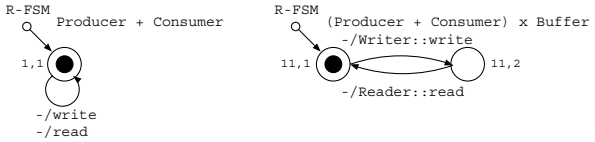

Figure 5. Intermediate FSM constructions: Writer + Reader and (Writer + Reader) × Buffer.

1987), 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` R-FSM$_{\text{Writer}}$`::write_1`" and
"`enables` R-FSM$_{\text{Reader}}$`::read_1/-`" for the `read` operation. Both annotations can be combined to
"`enables` R-FSM$_{\text{Reader}}$`::read_1/ enables` R-FSM$_{\text{Writer}}$`::write_1`".

Similarly, the result for the R-FSM$_{\text{Writer}}$`:write` operation is
"`enables` R-FSM$_{\text{Writer}}$`:write_1/enables` R-FSM$_{\text{Reader}}$`: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 the following lemma

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

Proof 1
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 1
Algorithm 2 solves the synchronisation problem 2.

Proof 2
From lemma 2 we know that $((A + B) \times C)$ describes all possible sequences of calls from A and B to the ken 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 2 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.

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.) Again, this algorithm produces an `2:1`-adaptor, which can be applied associatively multiple times to create an `n:1`-adaptor for an arbitrary `n`.

3.3. Protocol Changing Adapters

In the above section we concentrated on the synchronisation 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 synchronisation points (if existing). In this section we tackle the case where one ken (A) uses another ken (B), but the protocols R-FSM$_A$ and P-FSM$_B$ are not compatible. Because of simplicity, in the latter we refer with C and P to R-FSM$_A$ and P-FSM$_B$. In some cases we can compute a restriction of C's functionality (i.e., adaptation of P-FSM$_A$ (Reussner, 2001)). 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 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 R-FSM$_{\text{SimpleCDPlayer}}$ and the P-FSM$_{\text{CDPlayer}}$ are shown in Figure 6.

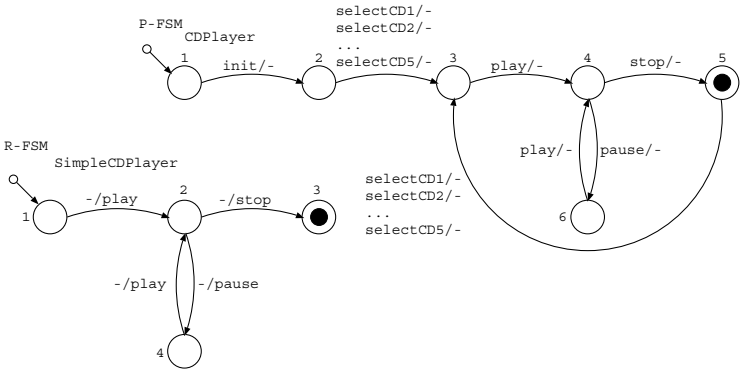

Figure 6. R-FSM$_{\text{SimpleCDPlayer}}$ (left) and P-FSM$_{\text{CDPlayer}}$ (right)

In this example we need to prefix R-FSM$_{\text{SimpleCDPlayer}}$'s method `play` in state one with calls to `init` and `selectCDn`. Here we can recognise two simple facts: (a) there may be several different possible prefixes. This ambi-

guity must be resolved by the software designer (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 R-FSM$_A$ and a P-FSM$_B$, we look for a function `prefix` which given a triplet $(s_c, s_p, \textit{method})$ 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 $(s_c, s_p, \textit{method})$ 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 a 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 tuple (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 tuple (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 7. As the result 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 (Kleene, 1956), each finite FSM describes a regular lan-

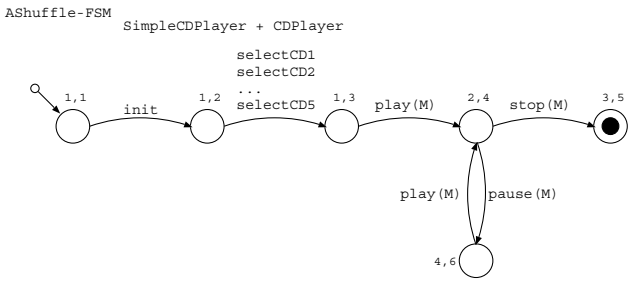

Figure 7. The asymmetric shuffle-FSM of the `CDPlayer` and the `SimpleCDPlayer`

guage. This language clearly depends on the start state of the FSM. When not assuming a fixed start state, we can parameterise the language recognised by an FSM with the start state. Let $L_C(s)$ denote the language recognised by FSM C, when state s is takes as its start state. A finite FSM may contain λ transitions, i.e. transitions which do not consume any input symbol (and so are used non-deterministicly). The set RL (restricted language in dependence of the start 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 3 (Construction of the Asymmetric Shuffle-FSM)
Given one requires-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 start-state is (s_{0C}, s_{0P}),*

- *and the transition function* $t : S \times I \to S$ *is defined*

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

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

After the construction of this FSM, one may have to remove unreachable or dead states. The function `Set:prefix` (s_c, s_p, i) 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 }

Likewise, the function `Set:postfix` (s_c, s_p, i) 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). Formal definitions of the functions `prefix` and `postfix` are given in (Schmidt and Reussner, 2000). When selecting a prefix in state (s_c, s_p), we must ensure that P can accept all possible sequences which C can emit. If necessary we use further prefixing to ensure the acceptance. This is ensured by predicate `LC`. Putting this together, the functions `prefix` and `postfix` solve the initialising / finalising problem.

The complexity of this algorithm lies mainly in the construction of the asymmetric shuffle-FSM and the cross product ($\mathbf{O}(|S_C| \cdot |S_P| \cdot \max(|I_C|, |I_P|))$).

4. Related Work

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 (Balter et al., 1998), or in DARWIN (e.g., Magee et al., 1995; Radestock and Eisenbach, 1996a; Radestock and Eisenbach, 1996b), and in our own DARWIN extension (Schmidt 1998; Ling et al., 1999). A general overview over ADLs is given in (Jazayeri et al., 2000). 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., (DeRemer and Kron, 1976).

Current industrial component models, such as Microsoft's (D)COM(+), Sun Microsystems' and IBM's EJB 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 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 specifications by petri-nets, automata-based models, or process-algebra (e.g., Vallecillo et al., 1999). Automata and automata based calculi have been used widely for protocol verification and testing in telecommunication and real-time component systems (Milner, 1980; Krämer and Schnmidt, 1987; Bochmann et al., 1982; Harel, 1987; de Alfaro and Henzinger, 2001). Nierstrasz proposes the modelling of the provided services with a nondeterministic finite state machine (Nierstrasz, 1993). Henzinger and Alfaro use finite state machines to describe interfaces and define a refinement cacluli (de Alfaro and Henzinger, 2001). Frolund developed contracts for specifying non-functional properties and requirements of components (Frolund and Koistinen, 1998). Yellin and Strom describe the protocol of offered and required services in one finite state machine, and use this protocol information to generate adapters and to support to bridge incompatible signatures (Yellin and Strom, 1997). While their work includes adapters which are special cases of the here presented initialising / finalising -adapter, they are less concerned with adapters specific for concurrent systems. Since they specify provides and requires protocols in a single finite-state machine, their adaptors can bridge incompatibilities for alternatingly interacting components. But their adaptor do not bridge incompatibilities which can only be solved by injecting somplete sequences, as initialising/finalising-adapter does.

5. Conclusions

In this paper we utilised an formal FSM based semantics for component interfaces in combination with an architectural definition of component configurations to generate adapters to overcome three 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 synchronisation 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. Open issues are related to: (1) parameter handling: the generation of adapters is semi-automatic; it would be interesting to develop skeleton adapter generation; also an integration of Yellin and Stroms approach (Yellin and Strom, 1997) 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.

Acknowledgements

The authors would like to thank Iman Poernomo for fruitful discussions.

References

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

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

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

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

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

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

Frick, A., Zimmer, W., and Zimmermann, W. (1996). Konstruktion robuster und flexibler Klassenbibliotheken. *Informatik, Forschung und Entwicklung*, 11(4):168–178.

Frolund, S. and Koistinen, J. (1998). Quality-of-service specification in distributed object systems. Technical Report HPL-98-159, Hewlett Packard, Software Technology Laboratory.

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

Jazayeri, M., Ran, A., and van der Linden, F. (2000). *Software Architecture for Product Families: Principles and Practice*. Addison-Wesley, Reading, MA, USA.

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

Krämer, B. (1998). Synchronization constraints in object interfaces. In: B. Kraemer, M. Papazoglou, H. Schmidt (eds), *Information Systems Interoperability*, Research Studies Press, Singapore.

Krämer, B. and Schnmidt, H. W. (1987). Types and modules for net specifications. In Voss, K., Genrich, H. J., and Rozenberg, G., editors, *Concurrency and Nets*, pages 269–286. Springer-Verlag, Berlin.

Ling, S., Schmidt, H., and Fletcher, R. (1999). Constructing interoperable components in distributed systems. In *IEEE Proceedings of TOOLS Pacific '99, Melbourne*, pages 274–284. IEEE Computer Society Press.

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

Milner, R. (1980). A calculus of communicating systems. number 92 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin.

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

M. Radestock, S. Eisenbach (1996). Formalizing System Structure, In *Proc. Int. Workshop on Software Specification and Design*, IEEE, pp. 95–104.

M. Radestock, S. Eisenbach (1996). Semantics of a Higher-Order Coordination Language, In *Proc. Conf. Coordination Languages and Models*, Springer-Verlag, Berlin.

Reussner, R. H. (2001). The use of parameterised contracts for architecting systems with software components. In Weck, W., Bosch, J., and Szyperski, C., editors, *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP'01)*.

Rumbaugh, J., Jacobson, I., and Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, MA, USA.

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

Schmidt, H. W. (1998). Compatibility of interoperable objects, In: B. Kraemer, M. Papazoglou, H. Schmidt (eds), *Information Systems Interoperability*, Research Studies Press, Singapore.

Schmidt, H. W. and Reussner, R. H. (2000). Automatic Component Adaptation By Concurrent State Machine Retrofitting. Technical Report No. 2000/81 of the School of Computer Science and Software Engineering, Monash University, Melbourne.

Shaw, A. C. (1978). Software descriptions with flow expressions. *IEEE Transactions on Software Engineering*, 4(3):242–254.

Szyperski, C. (1998). *Component Software: Beyond Object-Oriented Programming*. ACM Press, Addison-Wesley, Reading, MA, USA.

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

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