Modelling Parametric Contracts and the State Space of Composite Components by Graph Grammars

Ralf H. Reussner[1], Jens Happe[1], and Annegret Habel[2]

[1] Software Engineering Group (Fax ++49 441 9722 502)
[2] Formal Languages Group (Fax ++ 49 441 798 2965)
University of Oldenburg, Germany
{ralf.reussner, jens.happe,
annegret.habel}@informatik.uni-oldenburg.de

Abstract. Modeling the dependencies between provided and required services within a software component is necessary for several reasons, such as automated component adaptation and architectural dependency analysis. Parametric contracts for software components specify such dependencies and were successfully used for automated protocol adaptation and quality of service prediction. In this paper, a novel model for parametric contracts based on graph grammars is presented and a first definition of the compositionality of parametric contracts is given. Compared to the previously used finite state machine based formalism, the graph grammar formalism allows a more elegant formulation of parametric contract applications and considerably simpler implementations.

1 Introduction

Specifications should not be a means by themselves, but should have beneficial applications (besides of being a specification of something). Applications of software component specifications and software architecture specifications include automated test case generation, architectural dependency analysis [18] and component adaptation [14]. In any of these applications, additional information on a component (besides their interfaces) is beneficial which, on a first glance, seems to contradict the black-box use of components. However, the conflict between the need of additional information and black-box component (re-)use does not exist, as long as two conditions are fulfilled: Information on the component (beyond the interfaces) does not (a) have to be understood by human users, and (b) expose the intellectual property of the component creator. In addition, it is beneficial, if information on the component can be easily specified or even generated out of the component's code.

Parametric contracts [14, 15] support automated component protocol adaptation, quality of service prediction [16] and architectural dependency analysis by giving additional information on the inner structure of the component. In more detail, parametric contracts request so-called *service effect specifications* for specifying inner-component dependencies between provided and required services. These dependencies are simple to model as lists in case of signature-list interfaces (which required services are needed

M. Cerioli (Ed.): FASE 2005, LNCS 3442, pp. 80–95, 2005.

by a provided service). In case of protocol modelling interfaces things are more complicated, as one needs to specify sets of call sequences (which call sequences are needed to provide a service). As service effect specifications are an abstraction of a component's implementation's control-flow graph, it can automatically be extracted out of a component's code by control-flow analysis [10].

To make a component model compositional, all properties attached to a component should be present for a composite component as well (i.e., there is not difference between a basic and a composite component when neglecting the inner structure) and, in addition, the properties of a composite component should be derivable from the properties of the inner component plus the composition structure.

In this paper we discuss the compositionality of parametric contracts, in particular of parametric contracts used for protocol modeling interfaces. We use graph grammars to rewrite the graphical representation of a transition function of a finite state machine (FSM) modelling such sets of call sequences.

The contribution of this paper is twofold: Firstly, we show in detail how parametric contracts are modelled by a graph grammar. This is a novel contribution, as until now parametric contracts were described by a state machines and predicates. This directly leads to using graph grammars to component protocol adaptation. In particular, the graph grammar model leads to simpler implementations for the practically relevant adaptation of provides interfaces. Secondly, we show how service effect specifications of parametric contracts are composed by using graph grammars. This is the most important contribution with respect to applications, as until now, there was no compositional component model for components using parametric contracts. It should be emphasised that the contribution lies in the fruitful application of existing graph grammar formalisms to component based software engineering, not in the extension of the formalisms themselves.

This paper is organised as follows. In section 2 we review parametric contracts, their state machine models and give a brief introduction of the graph grammar notion used in this paper. In section 3 we show in general how graph grammars can be used to rewrite FSMs by interpreting their transition function as a graph. In section 4 we apply this idea to the main topic of this paper, namely protocol adaptation by parametric contracts and component state space composition. In section 5 we conclude by summarising the achievements of the paper, showing the limitations of our approach and discussing future work on open questions. Related work is discussed throughout the whole paper where appropriate.

2 Fundamentals

2.1 The Contractual Use of Components

The essence of design-by-contract[11] can be summarised as: If the client fulfils the precondition of the supplier, the supplier will fulfil its postcondition.

Much of the confusion about the term "contractual use" of a component comes from the double meaning of the term "use" of a component. The "use" of a component can mean either:

The usage of a component during run-time. This is, calling a service of a component. Therefore it should be evident that this type of contractual component use is nothing

different as using a method contractually. Thus this case should be called the use of a *component service* instead of the use of a *component*. As the contractual use of methods is well elaborated in literature [11], we do not consider this case here.
The usage of a component during composition time. This is, placing a component in a new reuse-context, like it happens when architecting systems, or reconfiguring existing systems (e.g., updating the component).

Depending on the above case, contracts play different roles. The usage of components at composition time is the actual important case when discussing the contractual use of components. Consider a component C acting as supplier, and the environment acting as client. The component offers services to the environment (i.e., the components connected to C's provides interface(s)). According to the above discussion of contracts, these offered services are the postcondition of the component, because this is, what the client can expect from a working component. According to Meyer's above description of contracts, the precondition specifies what the component expects from its environment to be provided in order to enable C to offer its services (as stated in its postcondition). Hence, the precondition of a component is stated in its requires interfaces.

Analogous to the above single sentence formulation of a contract, we can state:

> If the user of a component fulfils the components' requires interface (offers the right environment) the component will offer its services as described in the provides interface.

Checking the satisfaction of a requires interface includes checking for each required service whether its service contract is a a sub-contract of the service contracts of the corresponding provided service. Subcontracts are elobaroated in [12–p. 573].

2.2 Parametric Contracts

For a component developer it is hard to foresee all possible reuse contexts of a component in advance (i.e., during design-time). One of the severe consequences for component oriented programming is that one cannot provide the component with all the configuration possibilities which will be required for making the component fit into future reuse contexts. Coming back to our discussion about component contracts, this means, that in practice one single pre- and postcondition of a component will not be sufficient. Consider the following two scenarios:

1. the precondition of a component is not satisfied by a specific environment while the component itself would be able to provide a meaningful subset of its functionality.
2. a weaker postcondition of a component is sufficient in a specific reuse context (i.e., not the full functionality of a component will be used). Due to that, the component will itself require less functionality at its requires interface(s), i.e., will be satisfied by a weaker precondition.

As a consequence, we do not need statically fixed pre- and postconditions, but *parametric contracts* to be evaluated during deployment-time. In the first case a parametric contract computes the postcondition which is computed in dependency of the strongest precondition guaranteed by a specific reuse context (hence the postcondition is parameterised by the precondition). In the second case the parametric contract computes the

precondition in dependency of the post-condition (which acts as a parameter of the pre-condition). Due to this parametric mutual dependencies between the pre-condiction and the post-condition these contracts are called "parametric" contracts. For components a parametric contracts means, that provides- and requires-interfaces are not fixed. A provides interface is computed in dependency of the actual functionality a component receives at its requires interface, and a requires interface is computed in dependency of the functionality actually requested from a component in a specific reuse context. Hence, opposed to classical contracts, one can say:

> Parametric contracts link the provides- and requires-interface(s) of the same component. They have a range of possible results (i.e., new interfaces).

Interoperability is a special case now: if a component is interoperable with its environment, its provides interface will not change. If the interoperability check fails, a new provides interface will be computed.

Mathematically, parametric contracts are modelled by a function p mapping a provides interface P to the minimal requires interface $R = p(P) = \mathrm{R}_C$ specifying the needs of P. Hence p is a function of the set Prov_C of all possible provides interfaces of C to the set Req_C of all possible requires interfaces of C. A possible provides interface is any interface offering a subset of the functionality implemented in C, the set of all possible requires interfaces is the image of Prov_C under p. Note that p not necessarily is an injective function: several different provides interfaces may be mapped to the same requires interface. Consequently, the inverse mapping, associates to each requires interface of $R \in \mathrm{Req}_C$ a *set* of supported provides interfaces. To yield a single provides interface, we use the “maximum" element of this set. Formally, this element is the smallest upper bound of the set $p^{-1}(R)$. This smallest upper bound is the join of the elements of $p^{-1}(R)$ which exists because if provides interfaces P_1 and P_2 are elements of $p^{-1}(R)$ each of their elements (i.e., services, service call sequences, services with QoS annotations) is supported, consequently, the interface describing the set of all these elements is also itself element of $p^{-1}(R)$ ($P_1, P_2 \in p^{-1}(R) \Rightarrow P_1 \cup P_2 \in p^{-1}(R)$). For later use, we define the shorthand inv-$p : \mathrm{Req}_C \rightarrow \mathrm{Prov}_C$ as inv-$p(R) := \bigcup_{E \in p^{-1}(R)} E$. (Note that we use the more intuitive set-oriented notion of $\cup$ for the join-operator which is commonly referred to as $\sqcap$ in literature on lattices, etc.)

Like for a classical contracts, the actual parametric contract specification depends on the actual interface model[19] and should be statically computable. In any case, there's no need for the software developer to foresee possible reuse contexts. Only the specification of a bidirectional mapping between provides- and requires-interfaces is necessary.

2.3 Finite State Machines and Component Protocols

The protocol of the services offered by a component is defined as (a subset of) the set of valid call sequences. A *valid* call sequence is a call sequence which is actually supported by the component. For example for a file `open-read-close` is a valid call sequence, while `read-open` is not. The specified set of valid call sequences is called the provides-protocol.

Analogously, the protocol of the services required by a component is a set of call sequences by which the component calls external methods. This set of sequences of calls to external component services is called the requires-protocol.

The provides- and the requires-protocols are considered as sets of sequences. State machines are well-known notations for protocol specification [2, 9, 13, 20]. The benefits of a state machine specification of protocols are the representation of protocols in a compact and precise manner and the possibility of an efficient automatic formal analysis of protocols.

Definition 1 (Finite State Machine). *A finite state machine (FSM) is a system* $A = (I, Z, F, z_0, \delta)$ *where* I *is an input alphabet,* Z *is a finite set of states,* $F \subseteq Z$ *is a set of final states,* $z_0 \in Z$ *is a start state, and* $\delta \colon Z \times I \to Z$ *is a total transition function.*

Not that every partial transition function can be extended to a total one by adding a state $\perp$ and assigning $\perp$ whenever the partial function yields undefied or the state in consideration is $\perp$.

By P-FSM we denote the FSM specifying the provides protocol of a component, while the *component-requires-FSM* (CR-FSM) gives the requires protocol. The P-FSMs input alphabet is the set of methods provides by the component. In the reverse, the input alphabet of the the CR-FSM is the set of (external) methods required by the component. Since our implementation utilises a state-machine based approach we identify state-machines and protocols.

When modelling call sequences, we model for each state which methods are callable in this state. In many cases, a method call changes the state of the state machine, i.e., some other methods are callable after the call, while others, callable in the old state, are not callable in the new state. An example of the P-FSM of an exemplary video-stream component is shown in figure 1(a). The protocol described by this FSM represents the maximum functionality which can be offered by the video-stream. Note that the video-stream offers to manipulate the sound and the picture while playing and while pausing the video.

2.4 Graph Grammars

Graph transformation systems and graph grammars generalize string rewriting systems and Chomsky grammars, respectively: The objects are graphs, the rules are graph replacement rules, and the application of a rule to a graph yields a graph. Graph transformation systems and graph grammars are well-studied and applied to several areas of computer science (see, e.g. [17, 4, 5]). In the following, we provide the basic notions on graphs and graph grammars needed in the paper. Details and pointers to the literature can be found e.g. in [6, 8].

We consider directed, edge-labelled graphs with a finite set of nodes and edges. Source and target nodes of an edge are given by the source and target functions; the labelling of an edge is given by the labelling function.

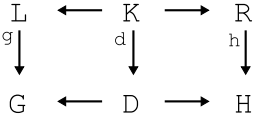
Definition 2 (Graph). *A graph over an alphabet* C *is a system* $G = (V, E, s, t, l)$ *consisting of two finite sets* V *and* E *of nodes (or vertices) and edges, two source and target functions* $s, t \colon E \to V$*, and a labelling function* $l \colon E \to C$*. The components of*

G are denoted by V_G, E_G, s_G, t_G, and l_G, respectively. The set of all graphs is over C is denoted by $\mathcal{G}_C$.

A graph morphism relates graphs. A graph morphism $g\colon G \to H$ between graphs G and H consists of two functions $g_V\colon V_G \to V_H$ and $g_E\colon E_G \to E_H$ that preserve sources, targets and labels, that is, $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, and $l_H \circ g = l_G$. A morphism g is an inclusion if g_V and g_E are inclusions and an isomorphism if g_V and g_E are injective and surjective. In the latter case, G and H are isomorphic denoted by $G \cong H$. A graph replacement rule consists of two graphs, the left-hand side and right-hand side of the rule. The left- and right-hand side are related by two inclusions from a common graph into the left- and the right-hand side.

Definition 3 (Rule). *A rule $r = \langle L \leftarrow K \rightarrow R \rangle$ consists of two graphs L and R, the left-hand side and the right-hand side of r, and two inclusions $K \rightarrow L$ and $K \rightarrow R$ from a common graph K. A rule r is an edge replacement rule if L is a graph with two nodes and a connecting edge and K is obtained from L by removing the connecting edge. The application of a rule r to a graph G amounts to the following steps:*

(1) Find a graph morphism $g\colon L \to G$ and check the following two conditions. Dangling condition: No edge in $G - g(L)$ is incident to a node in $g(L - K)$. Identification condition: For all distinct items $x, y \in L$, $g(x) = g(y)$ only if $x, y \in K$. (This condition is understood to hold separately for nodes and edges.)
(2) Remove $g(L - K)$ from G, yielding a graph $D = G - g(L - K)$.
(3) Add $R - K$ to tD, yielding a graph $H = D + (R - K)$.



An example of the application of a rule to a graph is given in figure 2.

The graph G directly derives H via r and g, denoted by $G \Rightarrow_{r,g} H$ or $G \Rightarrow H$. A sequence of direct derivations $G = G_0 \Rightarrow_{r_1,g_1} \ldots \Rightarrow_{r_n,g_n} G_n \cong H$ via $r_1, \ldots r_n \in \mathcal{R}$ is a derivation from G to H (of length n), denoted by $G \Rightarrow^*_{\mathcal{R}} H$. If the derivation is of length at least one, we also write $G \Rightarrow^+_{\mathcal{R}} H$.

A graph grammar consists of an alphabet, a nonterminal alphabet, a set of rules, and a start graph. The generated language consists of all graphs without nonterminal labels derivable from the start graph via the rules of the grammar.

Definition 4 (Graph Grammar). *A graph grammar is a system $\mathcal{G} = \langle C, N, \mathcal{R}, S \rangle$, where C and $N \subseteq C$ are alphabets, $\mathcal{R}$ is a finite set of rules, and S is a start graph. If $\mathcal{R}$ is a set of edge replacement rules, $\mathcal{G}$ is an edge replacement graph grammar. The graph language $L(\mathcal{G})$ generated by $\mathcal{G}$ consists of all graphs without nonterminal labels which can be derived from S by applying the rules of $\mathcal{R}$: $L(\mathcal{G}) = \{G \in \mathcal{G}_{C-N} \mid S \Rightarrow^*_{\mathcal{R}} G\}$.*

3 Refining Finite State Machines by Graph Grammars

3.1 Finite State Machines as Graphs

Usually, a finite state machine is drawn as a graph with additional information concerning the start and the final states. It can be represented as a graph by adding two nodes begin and end and edges with label Start and Final, respectively, from begin to the start state and the final states to end.

Let $A = (I, Z, F, z_0, \delta)$ be a finite state machine. Then $G(A) = (V, E, s, t, l)$ denotes the graph over the alphabet $C = I \cup \{\mathrm{Start}, \mathrm{Final}\}$ with node set $V = Z \cup \{\mathrm{begin}, \mathrm{end}\}$, edge set $E = Z \times I \cup \{e_{z_0}\} \cup \{e_f \mid f \in F\}$, and source, target, and labelling functions s, t, and l with

(1) $s(z,i) = z$, $t(z,i) = \delta(z,i)$, and $l(z,i) = i$ for all $z \in Z$ and $i \in I$,
(2) $l(e_{z_0}) = \mathrm{Start}$, $\mathrm{begin} = s(e_{z_0})$, and $t(e_{z_0}) = z_0$,
(3) $l(e_f) = \mathrm{Final}$, $f = s(e_f)$, and $t(e_f) = \mathrm{end}$ for all $f \in F$.

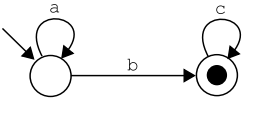
For convenience, start and final states are depicted in the classical way, that is, by an arrow pointing to the start state and a black dot inside each final state. The nodes begin and end are not drawn. An example of the representation of a FSM as a graph using the drawing conventions is given in figure 1(a).

3.2 Substuitution of Transitions in Finite State Machines

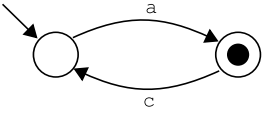
Substitutions of transitions in a finite state machine by finite state machines will be implemented by edge replacement graph grammars.

Example 1 (Substitution of Transitions by FSMs). Consider the FSM in figure 1(a). The transition b from the start state to the final state shall be replaced by the FSM given in figure 1(b).

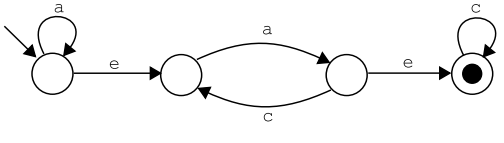
The result of the substitution is shown in figure 1(c). The ε-transitions are needed to maintain the structure of the original finite state machine. Without these ε-transitions



(a) Original FSM



(b) FSM for substitution



(c) FSM resulting from the substitution

Fig. 1. Substitution of transitions by FSMs

(by just linking the transitions connected to the start state with the source of transition b and the ones connected to the final state with the destination of transition b) it would be possible to go from the final state back to the start state. This contradicts the structure of the original FSM.

The implementation of the substitution of transitions by FSMs by an edge replacement graph grammar is done as follows: The FSMs are transformed into graphs and a set of substituting rules is defined.

Let A be a FSM, $I' \subseteq I$ a subset of input symbols selected for substitution, and $\mathrm{sub}\colon I' \to \mathcal{A}$ a mapping assigning a FSM of the set of FSMs $\mathcal{A}$ to each selected input symbol. For $i \in I'$, let $A_i = \mathrm{sub}(i)$ denote the FSM associated with the input symbol i. Then the edge replacement graph grammar $\mathcal{G} = \langle C, N, \mathcal{R}, S\rangle$ associated with A and sub is as follows: $S = G(A)$ is the start graph, $N = I'$ is the set of input symbols selected for substitution, and $\mathcal{R} = \{r_i \mid i \in I'\}$ is a set of rules where, for $i \in I'$, $r_i = \langle L_i \leftarrow K_i \rightarrow R_i\rangle$ is constructed as follows. L_i is the handle induced by i, that is the graph $(\{v_1, v_2\}, \{e\}, s, t, l)$ with $s(e) = v_1, t(e) = v_2$, and $l(e) = i$, K_i is obtained from L_i by removing the edge e, and R_i is the graph $G_\varepsilon(A_i)$ obtained form $G(A_i)$ by replacing the symbols Start and Final by the symbol ε. The label alphabet C consists of all symbols occurring in the start graph or some rules.

Example 2 (Application of Rules). The rule for the substitution in example in figure 1(b) is shown at the top of figure 2. The application of the rule to the graph of the FSM in figure 1(a) results in the direct derivation $G \Rightarrow H$ shown in figure 2.

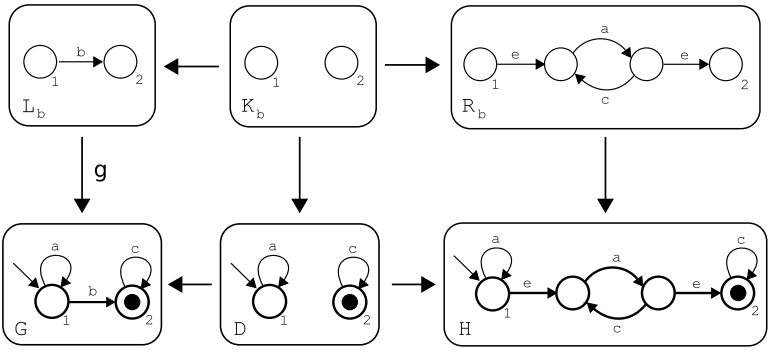


Fig. 2. Application of a rule

There is a close relationship between the iterated application of a substitution and the application of rules induced by the substitution [6]. We use the relationship for defining the iterated application of a substitution of transitions by FSMs.

Definition 5 (Substitution of Transitions by FSMs). *Let A be a FSM,* sub *a substitution, and $\mathcal{G} = \langle C, N, \mathcal{R}, G(A)\rangle$ the associated edge replacement graph grammar. Then the iterated application of* sub *to A is the graph language $L(\mathcal{G}) = \{H \in \mathcal{G}_{C-N} \mid G(A) \Rightarrow^*_{\mathcal{R}} H\}$.*

The edge replacement graph grammar starts in the graph of the FSM and replaces nonterminal labeled edges as long as possible. If the edge replacement graph grammar is non-recursive[3], then there are no infinite derivations from the start graph. Moreover, the rewrite relation $\Rightarrow_{\mathcal{R}}$ satisfies the diamond property, that is for every pair of direct derivation $G \Rightarrow_{\mathcal{R}} H_1$ and $G \Rightarrow_{\mathcal{R}} H_2$ with $H_1 \ncong H_2$ for some M there are direct derivations $H_1 \Rightarrow_{\mathcal{R}} M$ and $H_2 \Rightarrow_{\mathcal{R}} M$.

Fact 1. Let $\mathcal{G}$ be a non-recursive edge replacement graph grammar associated with a FSM and a substitution. Then $L(\mathcal{G})$ has exactly one element.

If the edge replacement graph grammar is recursive, then there is a symbol i with derivation $G \Rightarrow^+ H$ from the handle of i to a graph containing the symbol i. Since every symbol, in particular i, occurs in the start graph, there is an infinite derivation beginning with the start graph. Since there is exactly one rule for each nonterminal symbol, there is no chance to derive a terminal graph.

Fact 2. Let $\mathcal{G}$ be a recursive edge replacement graph grammar associated with a FSM and a substitution. Then $L(\mathcal{G})$ is empty.

For every non-recursive edge replacement graph grammar $\mathcal{G}$, we can find a linear ordering on the nonterminals such that every rule is strictly order-preserving, that is the symbol on the left-hand side is less that the nonterminal symbols on the right-hand side. Vice versa, if we can find such an ordering, then the grammar is non-recursive. Thus, we obtain the following.

Lemma 1. *It is decidable whether an edge replacement graph grammar is recursive.*

Proof. Similar to the proof for left-recursive context-free string grammars in [1]. If the edge replacement grammar is a non-recursive, then there is a linear order $<$ on the nonterminal symbols such that for every rule, the nonterminal symbol on the left-hand side is less than all nonterminal symbols on the right-hand side: Let $\lhd$ be the relation $A \lhd B$ if and only if $G \Rightarrow^* H$ where G is a handle with label A and H is a graph containing the label B. By definition of recursion, $\lhd$ is a partial order. (Transitivity is easy to show.) $\lhd$ can be extended to a linear order $<$ with the desired property. If $<$ is a linear order on the nonterminal symbols such that for every rule, the nonterminal symbol in the left-hand side is greater than all nonterminal symbols in the right-hand side, then there does not exist a nonterminal symbol A with derivation $G \Rightarrow^+ H$ from a handle G with label A to a graph H containing the nonterminal symbol A, i.e. the grammar is non-recursive.

If one considers a transition as function call, this problem can be seen as recursion of a function. In the context of function calls and parametric contracts it has been solved in [14].

[3] A symbol i in an edge replacement graph grammar is recursive if there exists a derivation $G \Rightarrow^+_{\mathcal{R}} H$ from the handle of i to a graph containing the symbol i. An edge replacement edge replacement graph grammar with at least one recursive symbol is recursive.

4 Applications to Protocol Adaptation with Parametric Contracts and Component Composition

Parametric contracts enable the deployer of a software component to determine the services a software component can provide in its current environment. Therefore, the requires interface is computed out of the components provides interfaces and service-effect-specifications. The result is intersected with the interfaces provided by the component's environment yielding an interface that contains only the services (of the component's requires interface) the environment can offer. This reduced requires interface is transformed into a reduced provides interface that includes only the services the component can provide in the current context.

It is also possible to compute the requirements of a component depending on the services needed by its environment. Therefore, the provides interfaces of a component are intersected with the (joined) interfaces required by its environment yielding a reduced provides interface that contains only the services that are needed by the environment. The result and the service-effect-specifications of the component are used to determine a reduced requires interface that asks only the services from the environment that are currently needed.

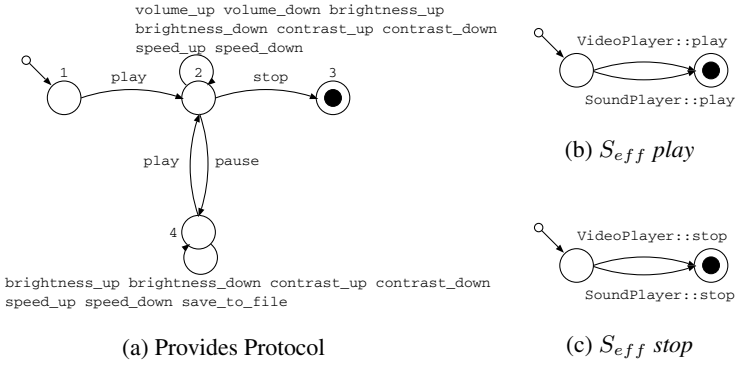
4.1 Computation of Requires Interfaces

The CR-FSM of a component can be derived from its P-FSM and its service-effect-specifications. Therefore, each transition in the P-FSM representing a service call is substituted by the associated service-effect-specification. This results in a new protocol consisting of the external services used by the component.

More formally speaking we have a P-FSM A_P and a set of service-effect-specifications $\mathcal{A}$, all given as finite state machines. Additionally, the function $v : I_P \rightarrow \mathcal{A}$ associates every input symbol of A_P with a service-effect-specification in $\mathcal{A}$.

Informally, we proceed as follows. Firstly, a graph grammar $\mathcal{G}$ is defined substituting each transition with the associated service-effect-specification. (The substitution without graph grammars is described in [14, 15].) The definition of $\mathcal{G}$ is similar to the one given in section 3 except that no ϵ-transitions are used for integrating the FSM into its new context, but special calling and return transitions. The requires protocol is a projection of this protocol where the calling and return transitions will be replaced by ϵ-symbols. But for the moment, they are needed for the construction of the adapted provides interface as described in section 4.2. Secondly, we apply this graph-grammar $\mathcal{G}$ to the provides protocol. The result of this application forms the requires protocol. Hence, the "algorithm" of computing the requires protocol is simply the application of $\mathcal{G}$ to the provides protocol. This is the minimal requires interface, as no substitution performed by $\mathcal{G}$ is superfluous. The time-complexity of this approach is bounded by the number of transitions of the P-FSM.

More formally, let $\mathcal{G}$ be a graph grammar according to section 3 with the original FSM A_p, the FSM set for substitution $\mathcal{A}$, and the mapping function v. Instead of defining E_s and E_f for the right-hand side of rule r_i (the rule associated with input symbol i in I_P) as above, set $E_s = \{e | s(e) = n_1, {l_V}'(t(e)) = Start, l_E(e) = i'\}$ and $E_f = \{e | {l_V}'(s_{R_i}(e)) = Final, t(e) = n_2, l_E(e) = return\}$, where $i', return \notin$



(a) Provides Protocol

(b) S_{eff} *play*

(c) S_{eff} *stop*

Fig. 3. Substitution of transitions by FSMs

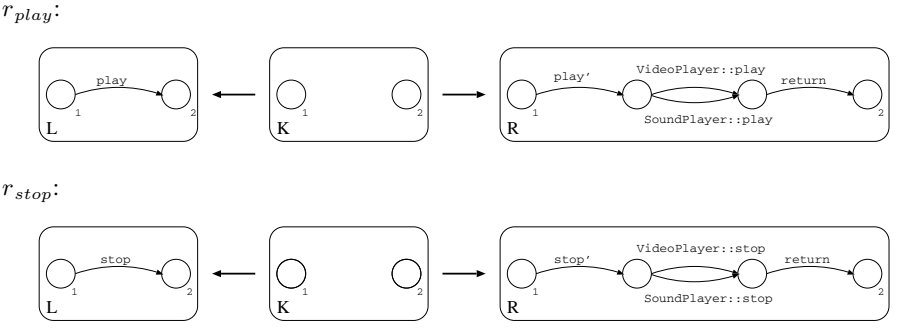


Fig. 4. Rules for substitution

$I_P \bigcup_{A_i \in \mathcal{A}_i} I_i$ are new edge labels and $l_V{}'$ is the node label function of G_{A_i}. E_s contains exactly one edge from the source of the substituted edge i to the node associated with the start state of A_i. Instead of labelling it with ϵ as above, a new edge label i' is defined marking the transition as a service call. E_f consist of edges from all nodes corresponding to the final states of A_i to the destination the substituted edge. The ϵ label used in section 3 is replaced by a general $return$ label tagging the edge as a return-transition from a called service.

As an example of the approach described above, consider the provides automaton of a video-stream component shown in figure 3(a). The video-stream component maps its provided services on a video-player and sound-player component. The service-effect-specifications for `play` and `stop` are given in figures 3(b) and 3(c).

Now, we create a graph grammar $\mathcal{G}$ that substitutes the transitions of the P-FSM as described above. Therefore, the start graph S of $\mathcal{G}$ is set to the graph representation of the provides automaton A_P shown in figure 3(a) and the non terminal alphabet N_E is set to the input alphabet $I_P = \{\texttt{play, stop, ...}\}$ of A_P. The total label alphabet contains I_P, the input symbols of the service-effect-specifications, and a set of new

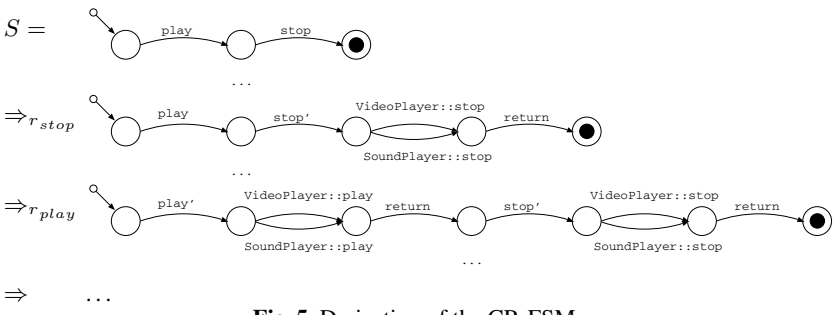


Fig. 5. Derivation of the CR-FSM

input symbols $I'_P = \{$`return, play', stop' ...`$\}$. Last but not least, the rules are created as shown in figure 4. For rule r_{play} the `play`-transition is replaced by the service-effect-specification of play. The transition `play'` on the right-hand side of the rule indicates that the service called `play` is executed. After this transition follows the start-state of the service-effect-specification of `play`, the calls to the video player and sound player components and the final-state which is connected to the target of the `play`-transition with a `return`-transition. The structure of r_{stop} and all other rules is analogous.

The computation of the requires protocol from the start-graph of $\mathcal{G}$ amounts in the derivation shown in figure 5. First, rule r_{stop} is applied on start-graph S, the graph representation of the provides automaton (it is only partially depicted) and the transition `stop` is substituted by the corresponding service-effect-specification. Next, rule r_{play} is applied. This process is continued until all provided services are replaced by its service-effect-specifications.

The result of the application of $\mathcal{G}$ on S is a FSM that can be easily transformed to the requires protocol of the video stream component. Therefore, the symbols introduced above ($I'_P = \{$`return, play', stop' ...`$\}$) are substituted by ϵ. The resulting FSM describes the service calls made by the component to its environment. Transforming this ϵ-FSM into a deterministic FSM and minimising the result leads to a good representation of the call sequences used by the component. For parametric contracts, the next step is to intersect the result with the P-FSMs of the component's environment as described in the beginning of this section.

4.2 Computation of Provides Interfaces

We are not only interested in the services required by a component depending on its environment, but also in the services that can be provided in the current environment. Therefore, we need to determine the P-FSM of the component depending on the protocols and services offered by its environment. This is done by intersecting the CR-FSM of the component with P-FSMs of its environment. The result is a CR-FSM containing only those services that are required by the component **and** can be provided by its environment. This reduced CR-FSM is used to determine a reduced P-FSM containing only those call sequences that can by provided by the component in the current context.

Therefore, we need to reconstruct a graph that has a structure similar to the result of our graph grammar $\mathcal{G}$ given in section 4.1. We can use this graph to apply the inverse rules

of $\mathcal{G}$ and derive the reduced P-FSM of the component. Finally, we need to intersect the result with the original P-FSM to clean up all service-effect-specifications that could not be transformed back to a single service call. This is required since during the intersection with the environment some states and transitions of the CR-FSM are removed and thus, some service-effect-specifications are incomplete and the inverse rules of $\mathcal{G}$ cannot be applied.

For the first step we define an *asymmetric intersection* between two finite state machines:

Definition 1 (Asymmetric Intersection)
Let $A = (I_A, Z_A, F_A, z_{0\,A}, \delta_A)$ and $B = (I_B, Z_B, F_B, z_{0\,B}, \delta_B)$ be two finite state machines with $I_A \subseteq I_B$, and $I'_B \subseteq I_B - I_A$. The asymmetric intersection of A and B is given by $A \times B = (I_B, Z_A \times Z_B, F, (z_{0\,A}, z_{0\,B}), \delta_{A\times B})$. Where $(z_1, z_2) \in F$, if $z_1 \in F_A$ and $z_2 \in F_B$. The state transition function $\delta_{A\times B}$ is given by

$$\delta_{A\times B}((z_1, z_2), i) = \begin{cases} (\delta_A(z_1, i), \delta_B(z_2, i)), & \text{if } i \in I_A \\ (z_1, \delta_B(z_2, i)), & \text{if } i \in I'_B \\ undefined, & \text{otherwise} \end{cases}$$

The asymmetric intersection creates a new FSM whose accepted language $L(A \times B)$ is a subset of $L(B)$ but (usually) not of $L(A)$. One can consider it as a finite state machine accepting the common language of A and B while ignoring all input symbols in I'_B for automaton A. Note, if I'_B is the empty set, the asymmetric intersection matches the regular FSM intersection.

Let G_{Req} be the result of the application of $\mathcal{G}$, G'_{Req} the result of the intersection, and A_{Req} and A'_{Req} the corresponding FSMs. Then, set $A = A'_{Req}$, $B = A_{Req}$ and I'_B to the symbols newly introduced by the construction of $\mathcal{G}$ (for example `return, play'` and `stop'` for the video-stream component). Then the result of the asymmetric intersection is structural similar to the result of the application of $\mathcal{G}$, but does contain only the service calls that are supported by the environment. So, we can use the inverse graph grammar $\mathcal{G}^{-1}$ of $\mathcal{G}$ for the construction of the reduced P-FSM.

The inverse graph grammar $\mathcal{G}^{-1}$ of $\mathcal{G}$ is constructed by inverting all rules of $\mathcal{G}$. The inverse rule r^{-1} of r is given by $L^{-1} = R$, $K^{-1} = K$, and $R^{-1} = L$. So, only the left-hand and right-hand side of the rules are exchanged.

The application of $\mathcal{G}^{-1}$ on the result of the asymmetric intersection is a graph whose complete service-effect-specifications have been substituted by the corresponding service calls and whose incomplete ones still exist. Hence, the final step is to clean up the result by intersecting it with the original P-FSM. This yields a reduced P-FSM containing only the call sequences that can be provided by the component in its current context.

The resulting P-FSM is maximal, as all provided service sequences are included. This is because all possible substitutions are performed by $\mathcal{G}^{-1}$ which means that an implemented services is available in all states where its requires call sequences (i.e., the left-hand side of the rule in $\mathcal{G}^{-1}$ corresponding to the service) are present in the CR-FSM.

The time-complexity is bounded by the number of substitutions to be performed which is approximately the number of transitions of the CR-FSM divided the average

number of edges ("transitions") of the left-hand sides of the rules of $\mathcal{G}^{-1}$. (This is in principle the same time-complexity as the computation of the requires-protocol, however, in the latter case the number of edges in the left-hand sides of the rules in $\mathcal{G}$ is is one, while the number of edges in the left-hand sides of the rules in $\mathcal{G}^{-1}$ is higher than one, as the left-hand sides in $\mathcal{G}^{-1}$ are service effect automata.

4.3 Compositional State Spaces

As mentioned, service effect automata are an abstraction of a component's internal state space. As transitions model calls to external services, all internal computations in-between two calls of an external service, are modelled by a single state of the service effect automaton. As service effect automata are part of our component model and component models should be compositional, in the following we will apply the above described mechanism of substituting a transition of a FSM by another FSM to the compositionality of components with service effect automata. Compositionality relates to a composition operator $\mathcal{O}$ taking two or more components and compsiting it to a new (composed) component. We consider as composition operator the use-relationship. We denote a component using others as C_1 and set set of components $C_2 \dots C_n$ *directly* connected to its requires interface(s) as $\mathcal{C}$. Hence, $\mathcal{O}(C_1, \mathcal{C}) =: C_c$ denotes the composition of $C_1 \dots C_n$. As this composition is again a component, it can be itself a parameter to the composition operator which if one wishes to include components indirectly used by C_1 in the composition. As C_c offers the same services as C_1 does, we are now interested in the service effect automata of C_c for these services. Formally speaking we have a service effect automaton A_P of a service s of component C_1 and a set of service-effect-specifications $\mathcal{A}$, all describing the bahaviour of services implemented by components $\mathcal{C}$. We are now interested in the service effect automaton of service s of the composed component C_c. Like in the other applications of edge substitution, the function $v : I_P \rightarrow \mathcal{A}$ associates every input symbol of A_P with a service-effect-specification in $\mathcal{A}$. Now we proceed like on the above construction of requires interfaces (section 4.1) with the only difference that A_P denotes a service effect automaton (and not a provides protocol) and the result of the substitution is the service effect automaton of service s of component C_c. For computing the requires protocol of component C_c, we use the provides protocol of C_c (which is per definition identical to the provides protocol of C_1) and all service effect automata of C_c (constructed as described above) and proceed as shown in section 4.1.

Note that the here presented approach to compositionality is not restricted to service effect automata. Muchmore, any state model with a partial function from transitions to a set of state models to be substituted can be composed by the approach described.

5 Conclusion and Future Work

We presented three applications of a graph grammar approach to finite state machine rewriting, namely (a) computation of requires protocols in dependency of provides protocols given invariant service effect automata, (b) the inverse: computation of provides protocols in dependency of requires protocols given invariant service effect automata and (c) the composition of service effect automata of components connected by a direct

use-relationship. These computations form the base for various applications in component based software engineering, such as automated component adaptation [14] and analyses of component based architectures [18]. The main benefits of using edge replacenent graph grammars are (a) a unified formal base of the above computations, (b) an important theory comparable with the theory on context-free string grammars [6], and (c) its simplicity (compared to the existing state machine based approach [14, 15]). In the future, we plan to explore the application of using hierarchical graph transformation [3] to model recursively inserted service effect automata.

References

1. Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing*. Prentice-Hall, Englewood-Cliffs, New Jersey, 1972.
2. Luca de Alfaro and Thomas A. Henzinger. Interface automata. In Volker Gruhn, editor, *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01)*, volume 26, 5 of *SOFTWARE ENGINEERING NOTES*, pages 109–120, New York, September 10–14 2001. ACM Press.
3. Frank Drewes, Berthold Hoffmann, and Detlef Plump. Hierarchical graph transformation. *Journal of Computer and System Sciences*, 64:249–283, 2002.
4. Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2: Applications, Languages and Tools. World Scientific, 1999.
5. Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 3: Concurrency, Parallelism, and Distribution. World Scientific, 1999.
6. Annegret Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1992.
7. Annegret Habel. Hypergraph grammars: Transformational and algorithmic aspects. *Journal of Information Processing and Cybernetics EIK*, 28:241–277, 1992.
8. Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
9. Gerald J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, USA, 1991.
10. Gunnar Hunzelmann. Generierung von Protokollinformation für Softwarekomponentenschnittstellen aus annotiertem Java-Code. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe (TH), Germany, April 2001.
11. Bertrand Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, October 1992.
12. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, USA, 2 edition, 1997.
13. Oscar Nierstrasz. Regular types for active objects. In *Proceedings of the 8th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-93)*, volume 28, 10 of *ACM SIGPLAN Notices*, pages 1–15, October 1993.
14. Ralf H. Reussner. *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Logos Verlag, Berlin, 2001.
15. Ralf H. Reussner. Automatic Component Protocol Adaptation with the CoCoNut Tool Suite. *Future Generation Computer Systems*, 19:627–639, July 2003.

16. Ralf H. Reussner, Iman H. Poernomo, and Heinz W. Schmidt. Reasoning on software architectures with contractually specified components. In A. Cechich, M. Piattini, and A. Vallecillo, editors, *Component-Based Software Quality: Methods and Techniques*, number 2693 in LNCS, pages 287–325. Springer-Verlag, Berlin, Germany, 2003.
17. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1: Foundations. World Scientific, 1997.
18. Judith A. Stafford, Alexander L. Wolf, and Mauro Capuroscio. The application of dependence analysis to software architecture descriptions. In M. Bernardo and P. Inverardi, editors, *Formal Methods to Software Architects*, volume 2804 of *Lecture Notes in Computer Science*, pages 52–62. Springer-Verlag, Berlin, Germany, 2003.
19. A. Vallecillo, J. Hernández, and J.M. Troya. Object interoperability. In A. Moreira and S. Demeyer, editors, *Object Oriented Technology – ECOOP '99 Workshop Reader*, number 1743 in LNCS, pages 1–21. Springer-Verlag, Berlin, Germany, 1999.
20. D. Yellin and R. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.