1 Mismatch Feedback

1.1 Problem Statement

What if a compatibility check fails? The idea of this work was to guide the developer as much as possible during the composition process. Therefore we want to give feedback on where the mismatch occurred and if possible how to cure it. There are two main problems here.

The first problem is to define the kind of feedback we want. It can be argued that in the most general case it makes no sense to provide mismatch feedback except for a simple warning: "total mismatch". Suppose we try to match a component with a role in the composition pattern that does not fit at all. Do we really want feedback saying that we need to adapt the component such and such to make it work or do we want the algorithm to come back with an error saying that this component is not compatible with the selected role? And if so where do we draw the line? In Figure 1 gives an example of such a mismatch.

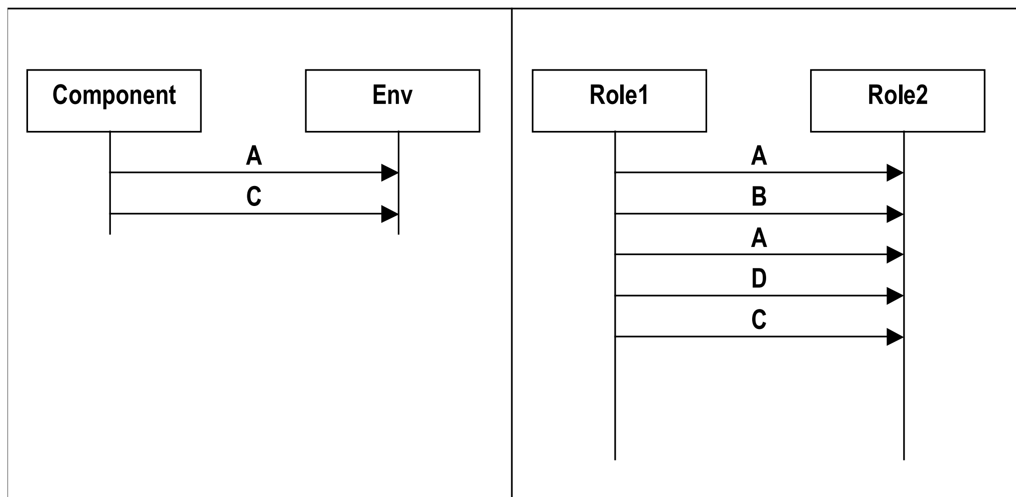

Figure 1:Indicating the mismatch?

How do we decide in this example if the message "A" of the component matches with the first or the second "A" of the composition pattern? And suppose it matches with the first "A" do we really want the program to show that everything between this first "A" and the message "C" is incompatible? Do we want feedback that is component centered or composition centered? I.e. do want to adapt the component to the composition or the composition to the component? Do we want incompatibilities to be shown on the component scenarios or on the composition scenarios? It is clear that there are no objective answers to this kind of questions.

1.2 Adapter Generation

In literature we find several tools that generate an adapter to make two state machines compatible. This is called adapter generation. There exist a whole field of research about adapter generation for finite state machines [***ref: Yellin and Strom, Reussner, Yaremski***]. We describe the results of Reussner [***ref***] as his solution is prototypical for the field and because he uses the asymmetric cross product to calculate his result. Next we introduce the adaptive programming library and we show how it can be used to generate adapters. This solution clearly outperforms all approaches we encountered in literature.

1.2.1 Reussner Adapter Generation

This algorithm is invented by Reussner [***Ref***]. We invited him to our lab for a couple of days and had a very fruitful conversation on adapter generation with him. There we noticed that he uses the asymmetric cross product to calculate what he calls: "a changing protocol adapter".

He often uses the example depicted in Figure 2.

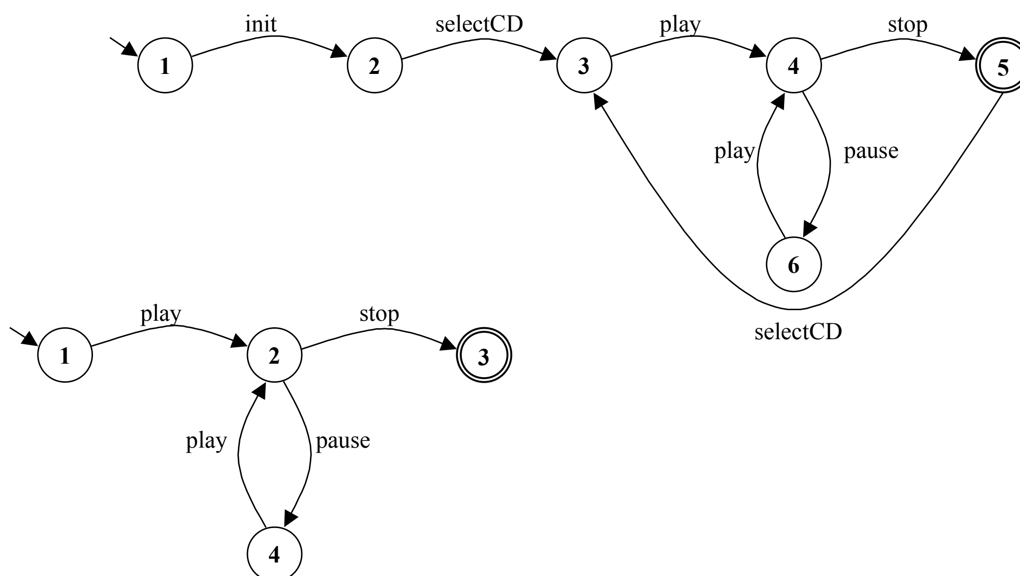

Figure 2: Adapting a simple CD player.

This example describes the usage behaviour of the GUI interface of two Compact Disc players. The first player is a more sophisticated player supporting multiple disc play, while the second player describes a standard CD player allowing only one CD to be played. Now suppose our composition pattern specifies the behaviour of the multiple disc player and our GUI component specifies a usage scenario corresponding with a single disc player. It is clear that we need to prefix the simple

component usage scenario with the "init" and "selectCD" transition to make it work. Reussner explains how he uses the asymmetric cross product to generate these prefixes.

The main step to compute these adaptations is to create the asymmetric cross product automaton. We start this algorithm with one "main" automaton and one "slave" automaton that will be adapted to the "main" automaton (in our case one component automaton that needs to be adapted to the composition automaton). The set of states in the resulting automaton is a subset of the Cartesian product of the states of the "main" automaton and the states of the "slave" automaton. The general idea is that the result contains two kinds of transitions: marked and unmarked transitions.

Marked transitions go from a state pair (*sm*, *sl*) containing one state from the "main" and one state from the "slave" with an input "i", where in the "main" the input "i" is handled in state *sm* and the "slave" automaton handles input "i" in state *sl*. In an unmarked transitions the input "i" is only handled in state *sm* (i.e. in the main automaton)

As we do not consider the case that inputs are accepted in the "slave" but not in the "main", we call this algorithm asymmetric. Figure 3 shows the result for the CD example.

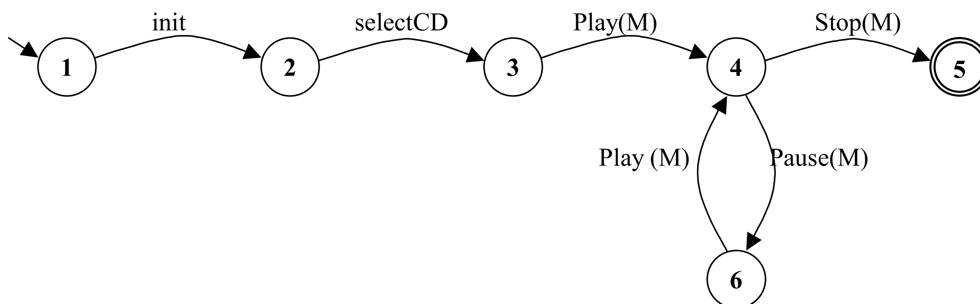

Figure 3: The generated adapter

For more details on this process see [***Ref Reussner***]. This algorithm thus generates a solution on how to adapt a component to a composition. (Swapping the role of the component and the composition renders a suggestion on how to adapt the composition pattern to the component.) However this is only one solution. It favours early matches over later matches (in Figure 1 this solutions assumes that the component matches with the first A in the composition pattern) and adapts one party to the other instead of adapting both parties. However it gives a good indication on what is missing to make them compatible and is thus useful as a feedback mechanism.

1.2.2 The Adaptive Programming Library

1.2.2.1 Introduction

In this section we show the connection between the adaptive programming library and adapter generation.

In the local check we try to check if a given component C can be used to implement a given role R. Both the component and the role are documented with a state machine that represents the state transitions of the component and the role. The component is compatible with the role if the component have common behaviour, i.e. they need to share at least one trace from a start state to a stop state that is exactly the same. Typical mismatches occur when the component does have such a trace except that it sends one or more message in between (so it has one or more transition that is not found in the role description, but it does have all the transitions as specified by the role). The traversal graph approach in [***ref strategies Lieberherr ***] recognizes all these situations and builds a state machine describing all possible adaptations for the component so that it becomes compatible with the role or renders an empty automaton if it cannot be done. To do this we consider the component state machine as the strategy graph and the role state machine as the class graph. The traversal graph is empty if there is no adaptation possible to make the component compatible. Otherwise the traversal graph contains all possible adaptations. To fully understand the analogy we need to go into more detail on the connection between NDFA intersection and the calculation of a traversal graph.

1.2.2.2 Connection calculation Traversal Graph with the intersection of NDFA's

During a discussion at ICSE 2000 Karl Lieberherr mentioned the connection between the calculation of the traversal graph (called algorithm 1 in [**ref**]) and the intersection of two NDFA's. A slide show explaining this connection can be found at [*** ref URL***]

The idea behind calculating the traversal graph is to check if a start-stop path specified by the strategy graph also exists in the class graph, allowing the class graph to use more internal transitions. I.e. a strategy specifies where to start, where to stop and what transitions should certainly be passed going from start to stop. This corresponds to traversal specifications following the template: FROM x_1 VIA $x_2 \ldots x_{n-1}$ TO x_n. Note that this is a restriction of the general traversal specification as defined by Lieberherr en co. They also allow specifying what transitions are *not* allowed. We made this restriction on purpose, because only this subset of traversal specifications has a counterpart in the intersection of NDFA calculation (*** check is this right ? ***).

This means that after each transition as specified by the strategy graph, we can have any number and any kind of transitions in the class graph as long as we go on with the transitions as specified by the strategy. An example makes this clear. Take the class graph and the strategy as specified in the left hand side of Figure 4 (and assume the name map to be identity). The strategy graph means: Traverse FROM a VIA c TO e. It is clear that the class graph supports this strategy. The resulting traversal graph is identical with the class graph.

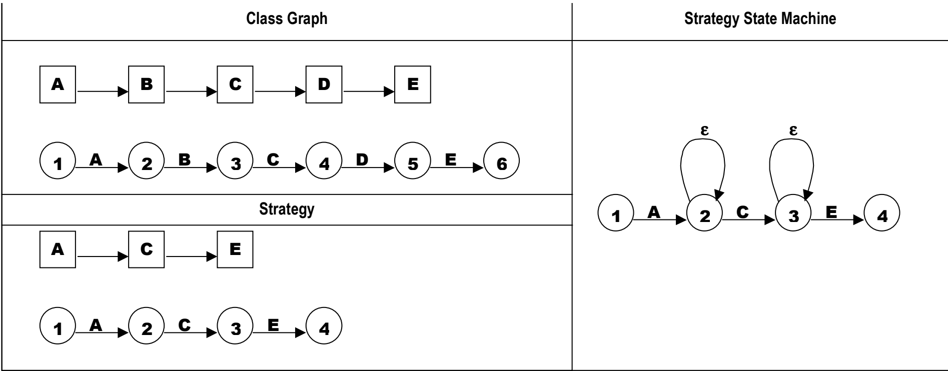

Figure 4: Class Graph and Strategy Graph with Corresponding State Machines

To see the connection with NDFA intersection I depicted the corresponding state machines for the class graph and the strategy graph just below them in Figure 4 (the correspondence is easy to see if you compare the strategy state machine with the specification FROM a VIA c TO e). Just calculating the intersection between the class graph state machine and the strategy graph state machine as specified on the left hand side returns an empty automaton. Remind now that the strategy graph allows more internal transitions in the class graph. This means that during the calculation of the intersection we should be able to proceed at wish in the class graph state machine until we find a common transition again. This is easily accomplished by adding epsilon transitions loops after every internal transition in the strategy graph state machine. The result is depicted at the right hand side of Figure 4.

It is important to note here that the traversal graph calculation algorithm does not perform a general intersection of two NDFA's. The correspondence works the other way round. I.e. it is possible to translate the traversal graph and the class graph (for the restricted class of strategies defined above) to NDFA's where the intersection of these NDFA's corresponds with the traversal graph obtained by the algorithm as defined in the Adaptive Programming library. It is in general not proven that two NDFA's can be converted to a class graph and a strategy graph so that their traversal graph corresponds with the intersection of these NDFA's.

1.2.2.3 Calculating Adapters

It is clear from the previous that the traversal graph algorithm calculates the intersection between a specific subset of NDFA's. So what does it mean if we just convert the role state machine to a strategy graph and the component specification to a class graph? The AP library first insert ε-transition loops after every internal transition in the class graph state machine (now corresponding with the role specification). This allows the component to proceed until it finds a compatible transition in the role specification. This allows for example the component and role R1 as specified in Figure 5 to be compatible. Indeed considering the role specification as the strategy means that we go from A via B to D. Our component specification allows this but also traverses C.

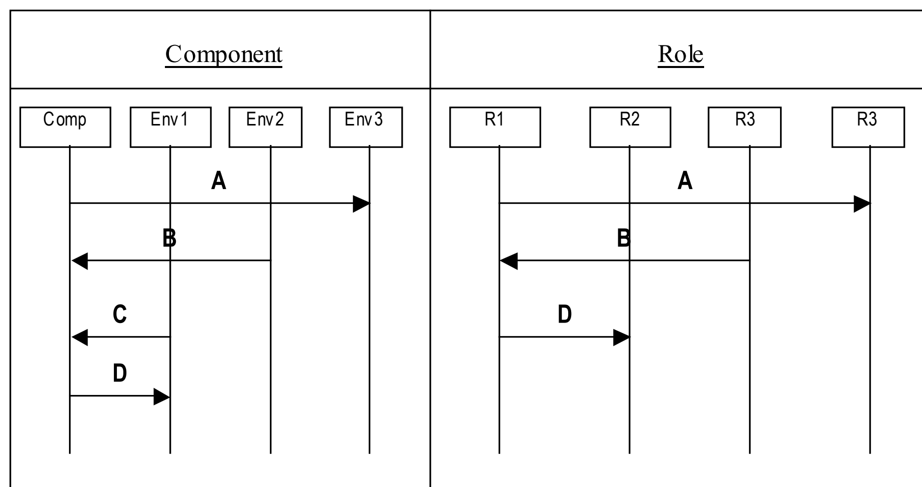

Figure 5: Using the AP library renders this component to be compatible with role R1

Thus we only need to convert the state machines of the component and the role to a class graph and a strategy and the traversal graph algorithm of the AP library calculates a traversal graph that corresponds with a state machine describing all traces that renders the component to be compatible with the role. Any trace that is found in the traversal graph corresponds with a solution of how we can adapt the component to the role. It is clear that switching the inputs returns all possible adaptations of the role to the component.

1.2.3 Conclusion

We implemented both algorithms in our prototype. While the algorithm of Reussner returns one clear solution, it can be argued that this is not always a sensible solution (see problem statement). It has however the virtue of being easy to use. The algorithm using the adaptive programming library is as far as we know a new idea and has the advantage of generating *all* possible adaptations. However the user needs to select the wanted solution manually making this algorithm more difficult to use. Nonetheless is this algorithm the best feedback tool as it clearly indicates what has gone wrong and how it can be fixed.