Automated Creation and Assessment of Component Adapters with Test Cases

Conference Paper · June 2010

DOI: 10.1007/978-3-642-13238-4_10 · Source: DBLP

2 authors:

Oliver Hummel
Karlsruhe Institute of Technology
56 PUBLICATIONS **704** CITATIONS
SEE PROFILE

Colin Atkinson
Universität Mannheim
212 PUBLICATIONS **5,724** CITATIONS
SEE PROFILE

Automated Creation and Assessment of Component Adapters with Test Cases

Oliver Hummel and Colin Atkinson

Software Engineering Group, University of Mannheim
68131 Mannheim, Germany
{hummel,atkinson}@informatik.uni-mannheim.de

Abstract. The composition of new applications from pre-existing parts has been one of the central notions in software reuse and component-based development for many years. Recent advances with component retrieval technologies and dynamically reconfiguring systems have brought the automated integration of components into systems into the focus of research. Even when a component offers all functionality needed by the using environment there is often a host of "syntactic obstacles" and to date there is no general solution available that can automatically address syntactic mismatches between components and their clients. In this paper we present an approach that automatically creates all syntactically feasible adapters for a given component-client constellation and selects the semantically correct one with the help of "ordinary" unit test cases. After explaining how our approach works algorithmically, we demonstrate that our prototype implementation is already able to solve a large fraction of the adaptation challenges previously identified in the literature fully automatically.

1 Introduction

As Brook's "No Silver Bullet" article [1] famously highlighted, software development deals with complex problems and is thus an inherently complex undertaking. Splitting software systems into more manageable subparts following the "divide and conquer" principle has hence been part of the toolset of software engineers for a long time [2]. Inspired by other engineering disciplines, McIlroy proposed the reuse of existing software artifacts as a means to reduce the effort involved in software development over four decades ago [3]. A number of years later, the ideas in his seminal paper not only gave rise to the notion of component-based development [4], they arguably also paved the way for the recent emergence of (web) services as a key technology of enterprise computing [5]. Similar to object-oriented software development, both approaches are based on the underlying metaphor of a jigsaw puzzle: the assembly of a "whole" based on smaller parts. And just like the pieces in a jigsaw puzzle, a number of objects, components or services needs to be placed together with their appropriate neighbors to yield the desired "whole". This process is known as composition in the component community and as orchestration in the web service community.

Generally speaking, in a software system, the individual connections between the pieces (i.e. between a client and a server component) are established through so-called interfaces comprising syntactic descriptions of operation signatures (comparable to the outer form of the puzzle pieces) and semantic descriptions of the functionality (comparable to the picture on the puzzle pieces if you will). Clearly, both descriptions can cause a mismatch when a part needs to be integrated into an existing system. A semantic mismatch occurs when a piece has the wrong picture (i.e. incorrect functionality) and can thus typically only be detected after the pieces have been put together and some test cases have been executed. Furthermore, it does not make much sense to attempt to perform larger modifications to the functionality of a part since it is more likely simply the wrong building block for the purpose in hand than a malformed part.

A syntactic mismatch, however, occurs when a piece has the right picture but the incorrect shape (i.e. incorrect operation signatures) and thus can be easily detected by the compiler or the runtime environment. Obviously, developers that need to deal with syntactic mismatches may alter the interface of either the server or the client component to fix the incompatibilities. However, this makes most sense early in the development process [6] and becomes much more expensive and difficult if, for example, a part needs to be replaced at run-time in a dynamically reconfigurable system.

Consequently, in this context, the non-invasive insertion of an adapter class in between two components treated as unmodifiable black boxes yields many advantages. It works analogously to the way in which power adapters are used to connect power-outlets with plugs from foreign countries. Of course, transferring this approach to software components is not a new idea: it has been around for such a long time that it is listed in practically every catalogue of development patterns such as for example the one by the Gang of Four [7]. As the comprehensive survey of Kell [8] underlines, adaptation is important for, and thus has been influenced by, a number of different software engineering research communities. Amongst developers, however, adaptation is often perceived as a tedious and error-prone activity that requires extensive testing, especially when interfaces are not well documented. Clearly, this perception and its impact make software adaptation a valuable target for automation.

The central theme of our article is thus to present a fully automated solution for overcoming syntactic mismatches that often arise when components are to be used and deployed in a new environment, whether this be in the context of reuse [3] or in the context of dynamically reconfigurable systems [22]. As we will discuss in more detail in section 5 we have found that it is feasible to use ordinary unit test cases to drive the automatic creation of adapters for software components. Before presenting this, we continue our article in section 2 by explaining important foundations of component-based software development and reuse. We especially elaborate on the test-driven reuse approach that initially trigged the development of our solution for automated adaptation. In section 3, following that, we explain how mismatching components are typically adapted for existing systems in order to present the required foundations for the automated adaptation approach we present in section 4. In section 5, we demonstrate the practical applicability of the tool we have implemented before we compare our approach to related works in section 6. A brief discussion of ongoing work and a summary of our contribution concludes our paper in section 7.

2 Component Software

As indicated in the introduction, the decomposition of large systems into more manageable parts is a common approach in software development. Still, the term “software component" (and especially its relation to "software object") is probably one of the most-discussed terms in software engineering. The first widely accepted definition was formulated at the Workshop on Component-Oriented Programming (WCOP) in 1996 [4]. Its main essence is that components have to hide their implementation behind contractually specified provided interfaces and may have explicit context dependencies (so-called required interfaces) only. However, the debate still continues as there are many open questions left. Today, for example, there are some widely accepted component technologies such as CORBA or EJB available, but interestingly they both use objects ("plain old Java objects" or POJOs in the latter case) as their underlying building blocks and do not fully match the component definition cited above for various reasons. To date, there is no genuine programming construct in object-oriented languages, and thus components can only be mimicked to a certain extend by using packages, (inner) classes and interfaces in order to achieve component-like behaviour. Only recently, some industry-driven efforts – such as OSGi [27] – have tried to mitigate this problem by defining deployment bundles that package together a number of Java classes and come closer to the above component definition. However, we do not want to pursue this discussion at this point - our goal with this brief outline is rather to motivate the understanding of the term that we will use throughout the remainder of this paper: We define a component as a software entity that offers its functionality only through a well-defined interface and thus is composable with other components without the need to disclose its underlying implementation. This definition clearly includes the common notions of objects and (web) services in use today and the ideas introduced in this paper can be applied to them as well as to other forms of modules. In fact, for the sake of simplicity and due to its high profile, we will use Java classes to illustrate our ideas in the following. We will explicitly mention other concepts only where specific differences arise.

2.1 Component Integration

Traditionally, one of the main drivers for component-based software development has been the reuse of existing software artefacts [3]. However, this is an area that long suffered from a lack of reusable material that prevented the creation of generally usable reuse systems. Only recently have some innovative approaches such as Test-Driven Reuse [11] taken advantage of the exploding amount of freely available open-source components and become able to deal with millions of components. However, even the latest component retrieval approaches need to live with the fact that increasing complexity of components reduces the likelihood of finding perfectly matching reuse candidates. Broadly speaking, as already explained in the introduction, two different criteria must be satisfied in order to integrate a component into a given environment, namely the component must match the needs of the environment syntactically and semantically. Although this distinction is already sufficient to understand the basic contribution of our approach, the more detailed set of criteria recently provided by Becker et al. [12] makes the goal of our approach clearer and will facilitate better comparability with other ideas later. The authors introduce a finer grained taxonomy that contains five distinct classes of integration mismatches, namely –

1. Technical mismatches
2. Signature mismatches
3. Protocol mismatches
4. Concept mismatches
5. Quality mismatches

These mismatches are ordered from top to bottom since a mismatch occurring in a higher class makes the consideration of lower classes immediately pointless. In other words, should a technical mismatch occur (i.e. component and target environment use different platforms) it does not make sense to check for a matching signature as the technical mismatch already prevents the components from functioning together. Signature (mis)matches in this classification are widely equivalent to what we described for syntactic matching earlier. Protocol mismatches focus on the invocation order of operations. For example, with a stack data type at least one element must have been pushed on the stack before its pop operation can be called successfully. Generally, these mismatches are a subclass of the semantic mismatches we introduced earlier and are related to concept (i.e. functionality) mismatches in the above classification as both classes are usually described using (semi-)formal pre- and postconditions [13]. However, since full formal descriptions of pre- and postconditions are often difficult if not impossible to check automatically (due to the halting problem), it makes sense to consider protocol matching separately as it can be investigated with simpler notations (such as state machines or petri nets). As the name implies, quality mismatches concentrate on non-functional issues, such as response time or reliability. While some, like response time, might be adaptable by the use of special mechanisms (such as caching in this case), other non-functional constraints such as the latter example are often not adaptable at all.

As mentioned before, the goal of the approach we present in this paper is the group of signature mismatches that has been further investigated by Becker et al. [12]. Based upon Zaremski and Wing's seminal work on signature matching [14] that we will introduce in more detail in section 3.1, Becker et al. identified a number of potential signature mismatches that need to be supported by an adaptation solution. We will utilize (and further explain) this collection in section 5.1 later in order to assess our prototypical implementation and to allow better evaluation of its capabilities.

3 Foundations of Component Adaptation

The adapter pattern described by the Gang of Four [7] as the archetype for adaptation comes in two forms – a static variant called the class adapter which is based on multiple inheritance and a dynamic variant based on delegation known as the object adapter. For today's most widespread object-oriented languages such as Java and C# that do not support multiple inheritance, the more appropriate variant is the object adapter which we will thus briefly explain in the following. The UML class diagram below depicts a situation where adaptation is required in order to make the `Client` class on the left-hand side work with the class shown on the right-hand side (the `Adaptee`). Unfortunately, the `Adapatee` provides an interface that is different to the specified (`Target`) interface required by the `Client`. Hence, the role of the

ObjectAdapter class is to implement Target by forwarding the requests it receives from the Client to the appropriate operation of the Adaptee. Ideally, of course, this has to happen transparently to both the Client and the Adaptee. In other words, neither the Client nor the Adaptee is aware of the fact that an adapter is "translating" the requests and responses between them. Obviously, for the sake of simplicity, the Target interface shown in Fig. 1 could be omitted and the Client could use the ObjectAdapter directly.

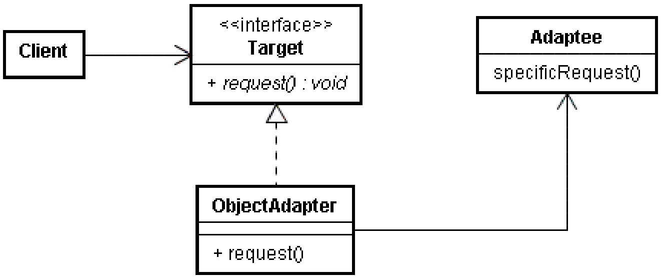

Fig. 1. Object adapter pattern as envisaged by the Gang of Four

The implementation of the ObjectAdapter class is straightforward. It needs to create an instance of the Adaptee during its own instantiation and forward all incoming requests to it as it executes. Once a response is returned from the Adaptee, it is passed on by the adapter to the Client. The challenge for a tool supposed to create adapters automatically is to figure out the internal "wiring" responsible for the forwarding of the adapter solely based on the interface and contract information provided by the Target and the Adaptee.

3.1 Signature Matching

The first fundamental prerequisite required for (automatic) adapter creation is to find out when two interfaces can be regarded as equivalent (or isomorphic) or when there is a chance they can potentially be made equivalent. In the reuse community, this process is usually called signature matching. Signature matching in its original form was defined by Zaremski and Wing [14] for the retrieval of functions and modules in functional languages from a component library and recognizes a match between two functions when they are identical in terms of the types they use in their signatures. In other words, the names of functions and parameters are fully ignored. More formally, this can be expressed as follows:

$$\textit{Signature Match}(q, M, C) = \{\mathrm{c} \in \mathrm{C} : \mathrm{M}(\mathrm{c}, \mathrm{q})\}$$

This means, a signature match requires a given query signature *q*, a match predicate *M* and a component library *C* in order to deliver a set of components where each one satisfies the match predicate. The signature of a function is definied as the list of types used as the function's input and output parameters and the exceptions it can throw. In addition to simple function matches, Zaremski and Wing also investigated module matches where a module is seen as a multiset of functions exhibited in the interface of the module. To our knowledge, only [15] has transferred these ideas to an object-oriented language, namely Ada, by condensing a collection of operation signatures into an object abstraction. We are not aware of any work in this direction for today's widely used object-oriented languages such as Java and C#. However, it is fairly straightforward to also apply these ideas to today's object-oriented languages and components as well as services.

Clearly, it is not necessary for the desired interface and the adaptee to be absolutely isomorphic, it is obviously sufficient if all operations of the adapter can be mapped to one operation of the adaptee; there can still be unused operations in the latter. However, it often happens that an operation signature can appear more than once within an adaptee which is a challenge not solvable purely by the means of signature matching anymore. Although the names of the operations might help in a case like this, in practice, there are often situations where establishing the right match becomes a tedious task even for a human developer. Consider, for example, the case in which operations are not well documented or not even well named (as is today often the case with web services). Further ideas developed by Zaremski and Wing include the use of so-called relaxed signature matches that, for instance, also allow different parameter orders to be accepted. Likewise, the idea of "relaxing" parameter or return types used with functional languages is also applicable for primitive types in object-oriented languages today. The general rule there is that preconditions cannot be strengthened and postconditions cannot be weakened for a subtype. Translated to parameters in operation signatures, this means that the "range" of a parameter in a reuse candidate can be increased (e.g. a `long` parameter on the adaptee side can also accept an `int` from the client or some reference type parameter can also accept objects of a subtype). Clearly, the inverse principle is valid for return types. For object types this can be based on the well-known Liskov Substitution Principle [16].

In order to conclude this subsection, we want to reiterate that signature matching is only able to determine whether two operation signatures can be considered equal, which is, of course, an important prerequisite for adaptation. However, it cannot be used to determine whether two operations are semantically adaptable or to derive the required mapping of operations and their parameters for the adaptation itself. We will discuss how to deal with this challenge in the next section.

4 Automating Adaptation

In this section we explain how the appropriate counterpart for a desired operation can be automatically identified in a candidate component. In other words, the challenge that we address here is finding the "correct" way of mapping the operations and the parameters of the desired component to those in an adaptee component. This is essentially a two-stage process:

First, based on signature matching, all potentially correct counterparts (i.e. all syntactically matching operations) need to be found. Details of the algorithms that create all valid permutations of the operation and parameter mappings are explained in the next subsection. Second, once all potential mappings have been established it is necessary to find the correct mapping for the adapter amongst the created permutations. For this purpose, it is necessary to have a specification of the functionality expected by the client at hand. However, specifying the functionality of software components is difficult and has consequently been an area of intensive research for decades. The commonly accepted approach today is the use of contracts [13] that specify pre- and post-conditions of operations in some typically (semi-) formal way. However, developers often perceive this as cumbersome since they need to learn an additional specification language and thus contracts are rarely used in practice. In Java, for example, this situation has been recently alleviated with the introduction of assertions that allow expressing pre- and postconditions in Java syntax. Nevertheless, due to the halting problem, the checking of assertions still requires the execution of code with concrete input values and is thus and closely related to the following idea from the reuse community. There, Podgurski and Pierce came up with the idea of using so-called samples (i.e. tuples of input and expected output values) to check the semantic fitness of operations [10]. About a decade ago, the test-driven development movement popularized the similar approach of using test cases created prior to the actual implementation as a specification for the required functionality [9]. Test cases have recently also been used successfully to implement so-called test-driven reuse [11] where they are used to evaluate the semantic fitness of reuse candidates. In fact, our adaptation solution was largely motivated by the need to adapt reusable Java classes in the context of our research in that area. Although test cases in this context naturally do not guarantee a full semantic assessment of the tested component, we have found them being a viable candidate for assessing the quality of adapters as we will demonstrate and discuss in section 5.

4.1 Permutation Creation

As indicated before, the first step required by our automated adaptation process is the creation of a table containing all possible syntactical adaptations for a given adaptee class and the desired interface of a client. Essentially, this is a four stage process based on two algorithms explained in the following. First, signature matches need to be established between all methods of the adapter and all matching methods of the candidate (i.e. the adaptee) according to the following Algorithm 1.

Algorithm 1. Discovering feasible method mappings

```
for each method in the adapter
  initialize empty List $list_m$ of method mappings
  for each method in the candidate
    if signatures match
      add method mapping to $list_m$
    endif
  endfor
endfor
```

For a better understanding of the algorithm (and the later evaluation of our prototypical implementation) we illustrate its application by using an adaptation challenge for a component performing simple mathematical matrix calculations. It is inspired by

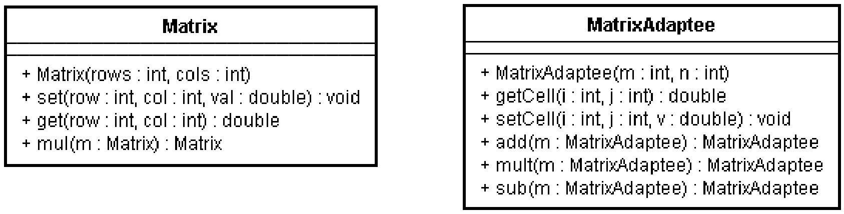
Matrix
+ Matrix(rows : int, cols : int)
+ set(row : int, col : int, val : double) : void
+ get(row : int, col : int) : double
+ mul(m : Matrix) : Matrix
MatrixAdaptee
+ MatrixAdaptee(m : int, n : int)
+ getCell(i : int, j : int) : double
+ setCell(i : int, j : int, v : double) : void
+ add(m : MatrixAdaptee) : MatrixAdaptee
+ mult(m : MatrixAdaptee) : MatrixAdaptee
+ sub(m : MatrixAdaptee) : MatrixAdaptee

Fig. 2. An exemplary adaptation challenge

an evaluation example used in [26] and illustrated in the figure below. The interface of the required matrix component is shown on the left-hand side and the one provided by the adaptee on the right-hand side.

For the sake of brevity we will omit a few methods in the following and merely consider the `set` and `mul` methods as they are sufficient to demonstrate the main challenges. As identified in previous work [17] the "translation" of a Matrix into a MatrixAdaptee required by the `mul` method is another challenge for adapter creation. We will discuss this issue in some more detail in section 5.1 and just assume it as solved for now. Thus, after Algorithm 1 has been executed, list$_m$ will contain the following entries:

```
set → setCell
mul → add
mul → mult
mul → sub
```

Here and in the following we use the right arrow to indicate an "is forwarded to" relationship. In other words, e.g. the `set` operation of the adapter (`Matrix`) forwards the request to `setCell` in the adaptee (`MatrixAdaptee`). Once these individual mappings have been established, they need to be combined for all methods contained in the `Matrix` component according to the algorithm shown below.

Algorithm 2. Combining method mappings for the whole class

```
initialize empty List list$_1$ of combinations
for each method in the adapter
  initialize empty List list$_2$ of combinations
  for each mapping in the list$_m$
    for each entry in list$_1$ or once if empty
      if candidate method not used in list$_2$ so far
        add method mapping to list$_2$
    endfor
  endfor
  list$_1$ = list$_2$
endfor
```

An important constraint in Algorithm 2 is that no method of the candidate may be addressed twice by adapter operations, which cannot happen in this simple example, however. The following list contains the three independently feasible internal wirings for the adapter class as obtained from the application of Algorithm 2:

```
set → setCell + mul → add
set → setCell + mul → mult
set → setCell + mul → sub
```

Furthermore, for each method, these mappings have to be combined with the feasible parameter permutations which can be derived in the next two stages of the permutation creation process using the same principles described in the two algorithms above. First, for each method mapping a list is created identifying which parameter in the adapter's method can be mapped to which parameter in the candidate method, e.g. for the set/setCell mapping:

```
set(int row, int col, double val)
                           → setCell(int i, int j, double v)
```

This yields:

```
row → i (int → int)
row → j (int → int)
col → i (int → int)
col → j (int → int)
val → v (double → double)
```

This list needs to be combined appropriately under the constraint that no parameter is used twice per method adaptation so that the resulting list of combinations has the following form:

```
set(row, col, val) → setCell(row, col, val)
set(row, col, val) → setCell(col, row, val)
```

Finally, we need to combine all method adaptations with their appropriate parameter permutations, which, for our example, leads to a total of twelve possible combinations of adaptations like the following:

```
set(row, col, val) → setCell(row, col, val)
mul(m) → add(m)
```

and

```
set(row, col, val) → setCell(col, row, val)
mul(m) → add(m)
```

And so on with `mul(m)` $\rightarrow$ `mult(m)` and `mul(m)` $\rightarrow$ `sub(m)`. Once all potential adaptations have been created like this, one configuration after the other can be checked for its fitness with the help of ordinary unit test cases typically created by the developers for the validation of a system's components anyway. We will explain this evaluation process in more detail in the next section.

5 Proof of Concept Implementation

The naive way to assess the adaptations created by the above algorithms with the help of test cases would be to submit an adapter class for each potential mapping to a testing environment along with the test case and the candidate class. However, this would involve a huge overhead since for every permutation a new adapter needs to be created, compiled, transferred, and executed. A more efficient solution that uses Java's reflection capabilities to lower the overhead to just one compilation run can be implemented as described in the following. The central idea is to not create new adapters at compile-time, but to interpose the permutation engine (the `Permutator` object in figure 3 below) in between the adapter and the candidate at run-time. This allows the switching to a new mapping within the adapter to happen more efficiently. The basic flow and the participants of this process are shown in the following sequence diagram and are explained in more detail thereafter.

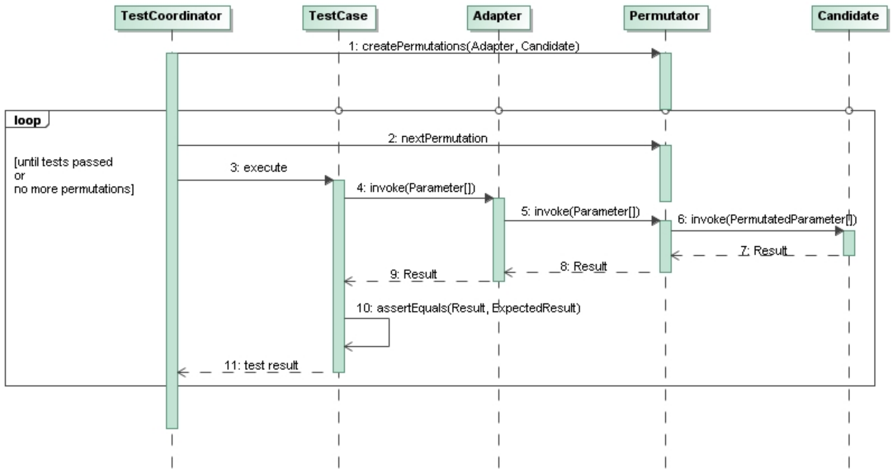

Fig. 3. Sequence diagram of the testing process

The `TestCoordinator` object on the left-hand side is responsible for managing the whole adaptation and testing process. Upon its invocation, it initializes the `Permutator` object and lets it create a lookup table that stores all possible permutations for the method and parameter mappings derived from the interface of the `Adapter` and `Candidate` (i.e. the adaptee) objects. After that, the engine is set up to carry out the permutation and testing cycle by executing the `TestCase`, which is a normal

JUnit test case. In order to provide the `TestCase` with the “illusion” of having an appropriate class under test, as discussed before, the `Adapter` is created according to the interface specified in the `TestCase`. The `Adapter` object, in turn, is created with the knowledge of the `Permutator` object and does not directly call the `Candidate` (i.e. the adaptee) as an adapter usually would, but rather forwards the parameters and an ID of the invoked method to the `Permutator`.

The `Permutator` internally keeps track of the state of permutations and is thus able to look up the relevant internal wiring for the current testing iteration. This allows it to invoke the actual operation of the `Candidate` with the appropriate parameter permutation. For the sake of clarity, we have depicted this scenario with just one invocation of the `Candidate` in figure 3. Of course, in real life, this needs to be done for every call from the `TestCase` to the `Candidate` – in other words, for one complete execution of the test case – with the same settings. As soon as one of these tests fails, the engine assumes that the current adaptation is not correct and a new permutation needs to be adjusted. Accordingly, the `TestCoordinator` notifies the `Permutator` to switch to the next permutation and the `TestCase` is executed once again. This surrounding loop is executed until either the complete test case has been passed without error or no further permutations are available. The former case obviously occurs for semantically acceptable reuse candidates and a correct adaptation while the latter indicates that the candidate is for some reason not reusable in the given context. This usually means that it does not offer the required functionality.

5.1 Evaluation

In order to assess the capabilities of our prototype we designed a complex adaptation challenge for a comprehensive “in vitro” evaluation. The two interfaces shown in figure 4 below contain the adaptation aspects currently supported by our system, namely – a constructor with a parameter that needs to be stored in an object variable, one method that accesses this variable, one method that changes this variable and various methods with multiple parameters. For this task, we have defined a simple JUnit test that specifies the interface of the class shown on the left-hand side of the figure and prepared an adaptee with the interface shown on the right-hand side. In order to make the challenge more expressive, the signature of each method appears twice to demonstrate that the tool is not only capable of finding the correct order of parameters, but identifying the correct operation as well. The “doNothing” operations contained in the adaptee are those that are meant to return a value that leads to a failed test. In total this challenge yielded 12,288 possible permutations and the small blue digits in figure 4 indicate the amount of possible permutations per method for the given example. The correct permutation was the 1,544th, which was discovered after roughly seven and a half minutes of our test system, which was a 2.0 GHz single-core notebook with 1.5 GB RAM running Windows XP.

This example also covers the most relevant challenges for an adapter generator based on the types of mismatched recognized by Becker et al. [12]. The table following below lists each one, shows how far our prototype supports it, refers to an example adaptation from the above challenge and contains a brief explanation for each.

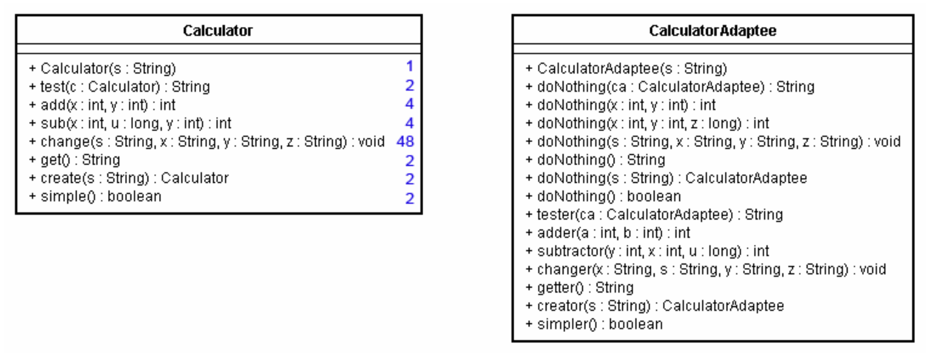
Calculator
+ Calculator(s : String) 1
+ test(c : Calculator) : String 2
+ add(x : int, y : int) : int 4
+ sub(x : int, u : long, y : int) : int 4
+ change(s : String, x : String, y : String, z : String) : void 48
+ get() : String 2
+ create(s : String) : Calculator 2
+ simple() : boolean 2
CalculatorAdaptee
+ CalculatorAdaptee(s : String)
+ doNothing(ca : CalculatorAdaptee) : String
+ doNothing(x : int, y : int) : int
+ doNothing(x : int, y : int, z : long) : int
+ doNothing(s : String, x : String, y : String, z : String) : void
+ doNothing() : String
+ doNothing(s : String) : CalculatorAdaptee
+ doNothing() : boolean
+ tester(ca : CalculatorAdaptee) : String
+ adder(a : int, b : int) : int
+ subtractor(y : int, x : int, u : long) : int
+ changer(x : String, s : String, y : String, z : String) : void
+ getter() : String
+ creator(s : String) : CalculatorAdaptee
+ simpler() : boolean

Fig. 4. Adaptation challenge built to check the features of prototype implementation

Table 1. Overview of adaptation challenges and how far they are supported by the prototype implementation

Mismatch	Supported	Example	Brief Explanation
1. Naming of methods	yes	noParam -> check	through evaluation of possible permutations
2. Naming of parameters	implicitly	add -> adder	with permutations
3. Naming of types	only for the adaptee itself	Calculator -> CalculatorAdaptee	used types are identical but have different names
4. Structuring of (used) complex types	indirectly	used types can be adapted separately	used types require adaptation as well
5. Naming of exceptions	no		exceptions with a different name can be adapted
6. Typing of methods	no		returned values can be of a subtype
7. Typing of parameters	no		submitted parameters can be of a subtype
8. Typing of exceptions	n.a.		essentially identical with 5 in Java
9. Ordering of parameters	yes	sub -> subtractor	through permutations
10. Number of parameters	no		number of parameters can vary e.g. due to constant or empty parameters
11. Return values of own type	yes	create -> creator	see above
12. Parameters of own type	yes	test -> tester	see above

Rows 11 and 12 are not contained in the reference publication and have been added by us. They are referring to the previously mentioned "translation problem" that occurs when a class uses objects of its own type as parameters or return values such as the `test` method in figure 4 (or the `mul` operation from figure 2). Since the `CalculatorAdaptee` expects an object of its own type it is not possible to simply forward the calculator instance in this case. Rather it must be replaced by the adapter with the appropriate adaptee instance. A solution for this issue is discussed in a previous publication [17] in more detail.

In addition to the example constructed above, we also applied our system to the matrix adaptation challenge from figure 2 after we had created a simple test case for it. Out of 24 feasible permutations the correct one was chosen in less than five 5 seconds. In order to have our system undergo another more practically relevant evaluation we used some further Matrix components we had previously retrieved from the merobase.com component search engine in summer 2007. In total, this set comprises 137 potentially reusable candidates out of which 10 have exactly identical method names with the interface we specified on the left-hand side of figure 2. Out of these 10 components only two candidates could be directly executed without adaptation or any other modification. Of course, our test case did not even compile successfully with the other 127 candidates due to deviating class or operation names. However, once we included our automatic adaptation creation into the testing process, we were able to test 26 out of these 137 candidates successfully. No false positives were detected amongst them in a manual inspection. There were in fact some false negatives, i.e. classes that seem to offer the right functionality but which our prototype was not able to adapt. The main reasons for this were lacking dependencies and thrown exceptions that our tool is not able to adapt yet. For adapting and assessing all 137 components our prototype requires about 5 minutes and 30 seconds on our test system.

5.2 Discussion

As the above evaluations reveal, the automated adaptation engine described in this paper not only works in a controlled laboratory environment, but also demonstrated its robustness with real world reuse candidates arbitrarily downloaded from the web. In this context it considerably increases the probability of finding appropriate reuse candidates. Interestingly, the applicability of automatic adaptation goes far beyond plain reuse. For example, it seems feasible to extend recent research on self-testing components [22] with an automated adaptation engine and thus to create self-adapting components [24] that can be used in dynamically reconfiguring systems. However, in order to allow the application of our approach in a practical environment with perhaps even more complex components, we still need to overcome a number of challenges. As mentioned before, it is obvious that test cases are by no means a complete specification of the behaviour of components. They can merely sample it and the reliability of reuse candidates retrieved with a test-driven reuse system and adapters generated by our prototype is of course closely correlated with the quality of the tests employed. Our previous experience with test-driven reuse [11] nevertheless indicates that already quite simple test cases created with common practices rule out false positives with high confidence. Clearly, establishing concrete measures and guidelines in this context is another interesting area for future research.

In order to conclude the discussion of our approach we briefly need to come back to the initial comparison of components and objects as this is certainly an important issue for its scalability. As we have demonstrated in the evaluation section, our prototype is able to adapt the interfaces of classes with significant complexity and since a well defined component is supposed to hide its implementation behind an interface of identical style our approach is applicable in that case as well. However, as our current "in vitro" implementation is still based on a brute-force assessment of all possible adapters its application can become time consuming with increasing interface sizes.

Thus, we are currently exploring potential optimization strategies for the testing process. Currently, the most promising idea to speed up the permutation evaluation is reusing test results of already tested adapter configuration for operations. In other words, the tool remembers previous test results and therefore does not need to process the adaptation of an operation again once it has been tested with the same adaptee method and parameter permutation. However, as their might exist subtle dependencies between operations we still assume that we need to have a fallback to the brute force variant in order to be sure not to miss a working adapter configuration. Nevertheless, we expect this solution will improve the scalability of our approach considerably. Scalability, however, is often confused with the ability to deal with more than one adaptee at a time in the context of components and class assemblies. However, adaptation per se is defined as a one to one mapping between adapter and adaptee [7] and composing a number of classes or components beyond that notion is rather an orchestration challenge (based on the facade pattern [7]) as currently under intensive investigation in the web service community. Nevertheless, the approach just presented might be helpful to find a general solution to this group of problems as well, but this has yet to be investigated as well.

6 Related Work

We have already referred the reader to [8] for a comprehensive overview of general adaptation techniques. This article lists a tremendous amount of literature that offers a wide variety of approaches for the integration of components into a system from a large number of communities. We, however, focus on those previous approaches that aimed to automate the adaptation process in the remainder of this subsection.

To our knowledge, Penix and Alexander [18] were the first researchers that sketched a solution for this challenge. They grounded their proposal in formal component specifications. However, they merely described some theoretical foundations, but provided neither concrete algorithms nor a practical implementation. More recently, Haak et al. [19] proposed a similar approach for automated adaptation of a subset of Standard ML and claim to have a working solution for simple modules enriched with machine-readable semantic specifications. However, neither a proof of concept nor an evaluation is provided. Furthermore, such semantic specifications impose additional effort on the developers and are not likely to be created for reusable components. Bracciali et al. [20] developed a methodology that comprises a small language for adapter specifications from which adapters can be automatically derived. However, it suffers from a similar shortcoming as the previous approaches since the specification of the adapters needs to be figured out by a human developer. Gschwind has also worked on the automation of component adaptation and proposed the use of an adapter repository where adapters can be stored and selected automatically [21]. However, the content of the repository (i.e. the adapters) need to be generated by humans again. Other more recent efforts to automate adaptation such as [23] or [25] also present interesting ideas, supporting the semi-automated generation of adapters for web services or the automated creation of adaptation contracts, but neither of them is able to fully support the whole process of adapter creation without human support.

7 Conclusion

The wide variety of existing articles discussing the automation of component adaptation demonstrates the importance of this topic. However, to the best of our knowledge, so far there exists no approach with appropriate tool support that would be able to automatically deliver practically usable syntactic adaptations for components in popular mainstream programming languages such as Java. We have experienced the necessity of such a technology during research we conducted for a test-driven component retrieval system and found that the underlying testing engine could also be used to assess the quality of automatically created adapters. Thus, we developed a so-called permutation engine that is able to derive all syntactically feasible adaptations between a specified interface and a syntactically mismatching component. Together with the testing engine this yields a system that is usually able to generate a working adapter for average size components completely without human intervention in just a few seconds. No previous approach has been able to offer such a large degree of automation including syntactic and protocol adaptation as our test-driven adaptation engine.

Another big advantage of our approach is the fact that it is solely based on artefacts (namely the test cases) that are normally created during the development process of a system and does not require any additional specification effort or the learning of a potentially complex formal specification language. Together with performance optimizations and coverage improvements currently under development it opens a host of interesting research possibilities that, in the future, promise to facilitate not only the composition of complex applications from components, but also the orchestration of web services.

References

1. Brooks, F.P.: No silver bullet: Essence and accidents of software engineering. IEEE Computer 20(4) (1987)
2. Parnas, D.L.: On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM 15(12) (1972)
3. McIlroy, D.: Mass-Produced Software Components. In: Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany (1968)
4. Szyperski, C.: Component Software, 2nd edn. Addison-Wesley, Reading (2002)
5. Erl, T.: Service-oriented architecture: concepts, technology and design. Prentice-Hall, Englewood Cliffs (2005)
6. Crnkovic, I., Chaudron, M., Larsson, S.: Component-based Development Process and Component Lifecycle. In: Proc. of the Intern. Conf. on Software Engin. Advances (2006)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
8. Kell, S.: A Survey of Practical Software Adaptation Techniques. Journal of Universal Computer Science 14(13) (2008)
9. Beck, K.: Extreme programming explained: embrace change. Addison-Wesley, Reading (2004)
10. Podgurski, A., Pierce, L.: Retrieving Reusable Software by Sampling Behavior. ACM Transactions on Software Engineering and Methodology 2(3) (1993)

11. Hummel, O., Janjic, W., Atkinson, C.: Code Conjurer: Pulling Reusable Software out of Thin Air. IEEE Software 25(5) (2008)
12. Becker, S., Brogi, A., Gorton, I., Overhage, S., Romanovsky, A., Tivoli, M.: Towards an Engineering Approach to Component Adaptation. In: Reussner, R., Stafford, J.A., Szyperski, C. (eds.) Architecting Systems with Trustworthy Components. LNCS, vol. 3938. Springer, Heidelberg (2006)
13. Meyer, B.: Applying Design by Contract. IEEE Computer 25(10) (1992)
14. Zaremski, A.M., Wing, J.M.: Signature Matching: A Tool for Using Software Libraries. ACM Transactions on Software Engineering and Methodology 4(2) (1995)
15. Stringer-Calvert, D.W.J.: Signature Matching for Ada Software Reuse. Master's Thesis, University of York (1994)
16. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Transaction on Programming Languages and Systems 16(6) (1994)
17. Hummel, O., Atkinson, C.: The Managed Adapter Pattern: Facilitating Glue Code Generation for Component Reuse. In: Edwards, S.H., Kulczycki, G. (eds.) ICSR 2009. LNCS, vol. 5791. Springer, Heidelberg (2009)
18. Penix, J., Alexander, P.: Towards Automated Component Adaptation. In: Proceedings of the International Conference on Software Engineering and Knowledge Engineering (1997)
19. Haack, C., Howard, B., Stoughton, A., Wells, J.B.: Fully automatic adaptation of software components based on semantic specifications. In: Kirchner, H., Ringeissen, C. (eds.) AMAST 2002. LNCS, vol. 2422. Springer, Heidelberg (2002)
20. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. The Journal of Systems and Software 74(1) (2005)
21. Gschwind, T.: Adaptation and Composition Techniques for Component-Based Software Engineering, PhD thesis, Technical University of Vienna (2002)
22. Brenner, D., Atkinson, C., Malaka, R., Merdes, M., Suliman, D., Paech, B.: Reducing Verification Effort in Component-Based Software Engineering through Built-In Testing. In: Information Systems Frontiers, vol. 9(2). Springer, Heidelberg (2007)
23. Motahari Nezhad, H.R., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-automated adaptation of service interactions. In: Proceedings of the International Conference on the World Wide Web (2007)
24. Atkinson, C., Hummel, O.: Reconciling Reuse and Trustworthiness through Self-Adapting Components. In: Proceedings of the International Workshop on Component-Oriented Programming (2009)
25. Martin, J.A., Pimentel, E.: Automatic Generation of Adaptation Contracts. Electronic Notes on Theoretical Computer Science, vol. 229, p. 2 (2009)
26. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading (2000)
27. O.S.G. Alliance: OSGi Service Platform Core Specification. Release 4. OSGi (2007)