Refactoring-Based Adaptation of Adaptation Specifications

Ilie Şavga and Michael Rudolf

Institut für Software- und Multimediatechologie, Technische Universität Dresden, Germany, {is13|s0600108}@inf.tu-dresden.de

Summary. When a new release of a component replaces one of its older versions (*component upgrade*), changes to its interface may invalidate existing component-based applications and require adaptation. To automate the latter, developers usually need to provide *adaptation specifications.* Whereas writing such specifications is cumbersome and error-prone, maintaining them is even harder, because any evolutionary component change may invalidate existing specifications. We show how the use of a history of structural component changes (*refactorings*) enables automatic adaptation of existing adaptation specifications; the latter are written once and need not be maintained.

1 Introduction

A software framework is a software component that embodies a skeleton solution for a family of related software products [17]. To instantiate it to a concrete application, developers write custom-specific modules (*plugins*) that subclass and instantiate public types of the framework's Application Programming Interface (API). Frameworks are software artifacts, which evolve considerably due to new or changed requirements, bug fixing, or quality improvement. If changes affect their API, they may invalidate existing plugins; that is, plugin sources cannot be recompiled or plugin binaries cannot be linked and run with a new framework release. When upgrading to a new framework version, developers are forced to either manually adapt plugins or write update patches. Both tasks are usually error-prone and expensive, the costs often becoming unacceptable in case of large and complex frameworks.

To reduce the costs of component upgrade, a number of techniques propose to, at least partially, automate component adaptation [7, 9, 11, 16, 18, 21, 24]. These approaches rely on and require additional *adaptation specifications* that are used to intrusively update components' sources or binaries. Component developers have to manually provide such specifications either as different annotations within the component's source code [7, 11, 21] or in a separate specification [9, 16, 18, 24] used by a transformation engine to adapt the old application code. However, in case of large and complex legacy applications the cumbersome task of writing specifications is expensive and error-prone.

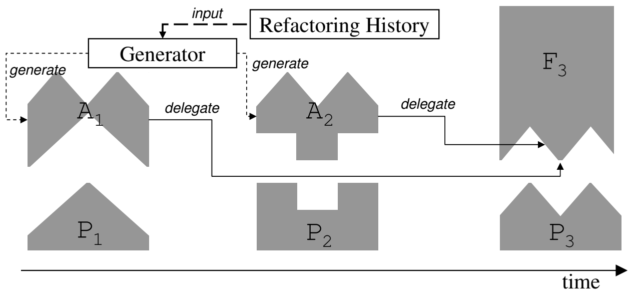

Fig. 1. Refactoring-based plugin adaptation. While a new plugin P_3 can be developed using the latest framework version F_3, existing plugins P_1 and P_2 are protected by adapters A_1 and A_2.

To alleviate the task of writing adaptation specifications, Dig and Johnson [13] suggest to reuse the information about the change to automatically perform adaptation. They investigated the evolution of five big software components, such as Eclipse [3], and discovered that most (from 81% up to 97%) problem-causing API changes were *refactorings* – behavior-preserving source transformations [19]. Common examples of refactorings include renaming classes and members to better reflect their meaning, moving members to decrease coupling and increase cohesion, and adding new and removing unused members. As at the time of framework refactoring its plugins are usually not available for update, they are invalidated by refactoring.

Based on the results of [13], two techniques proposed independently by Dig et al. in [14] and by us in [22] use refactoring information to *insulate* plugins from framework changes. The refactoring information required can be logged automatically (e.g., in Eclipse [3] or JBuilder [1]) or harvested semi-automatically [12] and thus does not require additional specifications from developers. Figure 1 shows, how we use the refactoring history to create adapters between a framework and some of its plugins upon the release of a new framework version. The adapters then shield the plugins by representing the public types of the old framework version, while delegating to the latest one.

Although in general the refactoring-based adaptation is capable of adapting more than 80% of problem-causing API changes, its restriction to pure structural changes also entails its limitations. Other changes affecting existing plugins require either manual or specification-based adaptation. For example, in a new framework release developers may change the way message exchanges between the framework and its plugins are performed. They introduce a new framework method and require calling it (possibly, with a default parameter) from plugins. This change leads to a *protocol mismatch* characterized by improper message exchange among components [8], in this case between the new framework and its old plugins that do not call the new method. To bridge protocol mismatches, developers have to specify messages that components may send and receive as well as the valid message exchange. Basing on these specifications *protocol adapters* that support proper intercomponent communication are generated (e.g., [20, 24]).

However, once such specifications are provided, component evolution unavoidably demands their *maintenance*, because evolutionary changes may alter component parts on which existing specifications rely, rendering the latter useless. For instance, if a framework type that is referred to in a specification is renamed, the specification is no longer valid. In such cases, specifications must be updated along with the change of the involved components, which raises the complexity and costs of component adaptation.

Extending our refactoring-based approach [22], we want to maximally ease the task of adapting remaining changes. For protocol changes, the corresponding specifications must be undemanding to write and not require maintenance throughout subsequent component evolution. The main contribution of this paper is in enabling component developers to write protocol adaptation specifications that are:

- in-time: specified at the actual time of component change. It is inherently easier to specify the adaptation of small incremental changes upon their application than to write large specifications involving complex component and change dependencies in one big step upon a new component release.
- durable: valid at the time of their execution regardless of any component change applied after the specification was written. This eliminates the need to maintain specifications.

Given a protocol adaptation specification, we use the information about subsequent structural component changes to perform the aforementioned refactoring-based adaptation and to shield *the specification* from those changes. In this way, we adapt existing specifications in the sense that we preserve their validity in the context of component evolution.

In Sect. 2 we sketch the refactoring-based adaptation by describing its main concepts used throughout the rest of the paper. Section 3 discusses the notion of protocol adapters in detail and continues with our main contribution – adaptation of existing adaptation specifications using refactoring information – followed by the discussion of relevant issues and empirical results in Sect. 4. We overview related work in Sect. 5 and conclude in Sect. 6.

2 Refactoring-Based Adaptation

Although refactoring preserves behavior, it changes syntactic component representation, on which other components may rely. Figure 2 shows the application of the `ExtractSubclass` refactoring to a component class modeling network nodes.[1] If an existing component *LAN* calling the method *broadcast* on an instance of *Node* is not available for update, it will fail to both recompile and link with the new *Node* version. As a consequence, any application using these two components will be broken by the upgrade of *Node*.

2.1 Comebacks

In [22] we formally define our refactoring-based adaptation, so that the adapters for the API refactorings could be constructed automatically and the soundness of

[1] This example was inspired by the LAN simulation lab used at the University of Antwerp for teaching refactoring [5]. For simplicity we omit several node methods (e.g., *send* and *receive*).

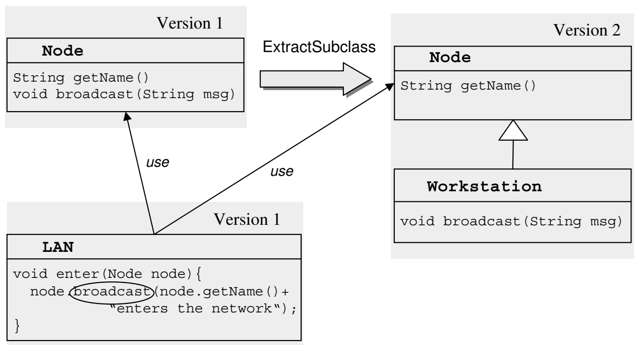

Fig. 2. `ExtractSubclass` refactoring. The method *broadcast* is moved from *Node* to its new subclass *Workstation* and cannot be located from the existing *LAN*.

the adaptation could be ensured. Effectively, we roll back the changes introduced by framework refactorings by executing their inverses. We cannot inverse directly on framework types, because we want new plugins to use the improved framework. Instead, we create adapters (one for each framework API type) and then inverse refactorings on adapters. We call these inverses *comebacks*. For our running example of `ExtractSubclass`, the compensating (object) adapter constructed by the comeback of the refactoring is shown in Fig. 3.

Technically, a comeback is realized in terms of refactoring operators executed on adapters. It is defined as a template solution and instantiated to an executable specification by reusing parameters of the corresponding refactoring. For some refactorings, the corresponding comebacks are simple and implemented by a single refactoring. For example, to the refactoring `RenameClass(name, newName)` corresponds a comeback consisting of a refactoring `RenameClass(newName, name)`, which renames

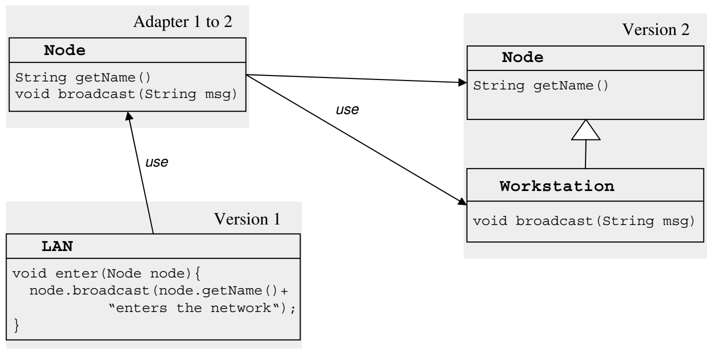

Fig. 3. Compensating adapter. The adapter represents the old *Node* and delegates to the appropriate component methods.

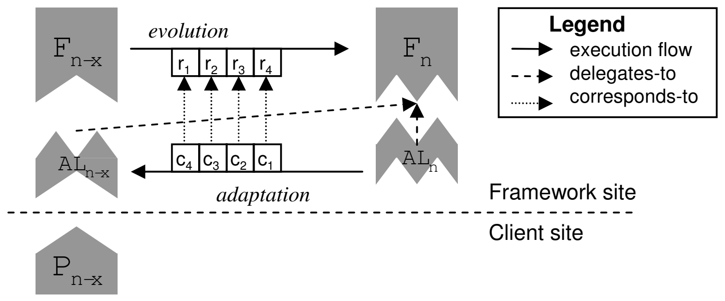

Fig. 4. Adaptation workflow. To a set of refactorings (r_1–r_4) between two framework versions correspond comebacks (c_4–c_1). Comebacks are executed on the adaptation layer AL_n backwards to the framework refactorings. The resulting adaptation layer AL_{n-x} delegates to the latest framework version, while adapting plugins of version P_{n-x}.

the adapter to the old name. For other refactorings, their comebacks consist of sequences of refactorings. For instance, the comeback of `PushDownMethod` is defined by the `DeleteMethod` and `AddMethod` refactoring operators, the sequential execution of which effectively moves (pushes up) the method between adapters. Moreover, complex comebacks may be defined by composing other, more primitives comebacks. This is the case for the comeback of `ExtractSubclass`, which is defined by combining the comebacks of `PushDownMethod` and `AddClass`.

For an ordered set of refactorings that occurred between two component versions, the execution of the corresponding comebacks in reverse order yields the adaptation layer. Figure 4 shows the workflow of refactoring-based adaptation in case the component being upgraded is a framework.[2] In a nutshell, we copy the latest API of the framework and inverse all refactorings on it, so that the old API (mimicked by the adaptation layer) is reconstructed fully automatically. First, we create the adaptation layer AL_n (the right part of the figure) that is a full copy of the latest framework API delegating to the latest framework version. Therefore for each API class of the latest framework version F_n we provide an adapter with exactly the same name and set of method signatures. An adapter delegates to its public class, which becomes the adapter's delegatee. Once the adapters are created, the actual adaptation is performed by executing comebacks backwards with respect to the recorded framework refactorings, where a comeback is derived using the description of the corresponding refactoring. When all comebacks for the refactorings recorded between the last and a previous framework version F_{n-x} are executed, the adaptation layer AL_{n-x} is syntactically identical to the API of F_{n-x}, while delegating to the newest framework version.

[2] Although we focus on object-oriented frameworks, the technique is similarly applicable to the upgrade of other types of object-oriented components, such as software libraries.

2.2 Tool Support and Limitations

We implemented our adaptation tool ComeBack! [2] using a Prolog logic programming engine. Currently we support Java and .NET components and can compensate for twelve common class and method refactorings. We provide a comeback library consisting of the corresponding comeback transformations specified as Prolog rules. Given the latest framework binaries, the information about the API types (type and method names, method signatures, inheritance relations) is parsed into a Prolog fact base. After examining the history of framework refactorings, the corresponding comebacks are loaded into the engine and executed on the fact base as described in the previous section. Once all comebacks have been executed, the fact base contains the information necessary for generating adapters (it describes the adapters) and is serialized to the adapter binaries. Thereby we extract and transform the information about the program and not the program itself; the adapter generation using this information is then the final step. We also combined ComeBack! with Eclipse to use the refactoring history of the latter.

Because comebacks are executed on adapters, for some refactorings comebacks cannot be defined due to particularities of the adapter structure. For instance, it is not possible to define a comeback for field refactorings (e.g., renaming and moving public fields), because adapters cannot adapt fields directly. Instead, we require API fields to be encapsulated. Moreover, for certain refactorings (e.g., adding parameters), developers are prompted for additional information (e.g., default parameter value) used to parameterize comebacks. In [23] we are making developers aware of such limitations and suggest solutions whenever possible.

3 Adapting Protocol Adaptation Specifications

We envisage the combination of the refactoring-based adaptation with other adaptation techniques. In this section we discuss, how several important problems associated with *protocol adaptation* are solved by its integration with the comeback approach. We start with a short introduction to and a running example of protocol adaptation, discuss then problems introduced by subsequent component refactorings, and stipulate a refactoring-based solution. Since our intention is to show problems and adopted solutions associated with the creation and maintenance of protocol adaptation specifications, and not the specifications themselves, we use a simplified example without going too much into the details of the specification formalism used.

3.1 Protocol Adaptation

Modifications of a component may affect its behavior, that is, the way it interoperates with other components (by exchange of method calls), leading to protocol mismatches. Common examples of such mismatches are [8]:

- Non-matching message ordering. Although components exchange the same kind of messages, their sequences are permuted.
- Surplus of messages. A component sends a message that is neither expected by the connected component nor necessary to fulfill the purpose of the interaction.

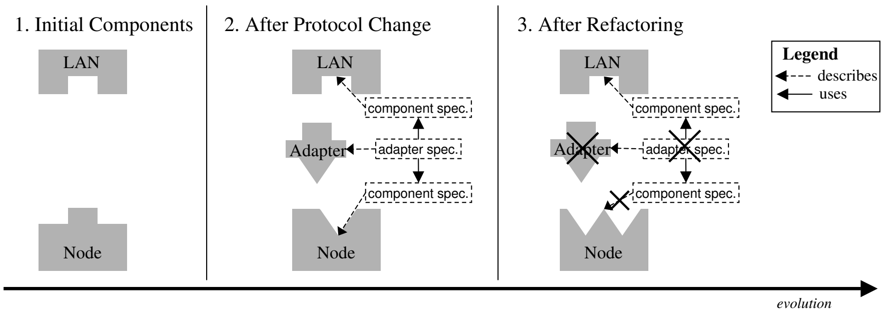

Fig. 5. Component Evolution. Initially the components cooperate as intended. The second release of *Node* requires adaptation specifications, which are then invalidated by a subsequent refactoring.

- Absence of messages. A component requires additional messages to fulfill the purpose of the interaction. The message content can be determined from outside.

Consider the two components of Fig. 5.1 (their APIs are not shown in the figure). The *LAN* component models a Token Ring local area network that accepts and removes nodes via the methods *enter(Node node)* and *exit(Node node)* and asks a node to create and forward a new packet using the method *broadcast(String msg)*. The *Node* component abstracts a separate network node and provides, among others, the methods to enter and exit the network (*enter(Node thisNode)* and *exit(Node thisNode)*). In addition, it has a method *broadcast(String msg)* that creates a packet from the given message, marks the packet with the ID of the creator node, forwards it to the successor node, and acknowledges (*broadcastAck()*) back to the network. Since both components are of the same release 1, their APIs and protocols match, so that there is no need for adaptation.

In the next release the developers generalized *Node* to increase its reusability for other types of networks. In particular, they introduced a two-phase protocol for entering the network: the node first asks for a permission to enter a network *enterReq(Node thisNode)*, passing in itself as the argument, and only if allowed actually enters the network. The same changes were applied to the protocol for leaving the network. In addition, there is no acknowledgement of the broadcast call sent back from *Node* anymore. Instead, the node issues a *ceased* message when it removes its previously created packet from the network.

This is an example of a protocol mismatch: although the components possess the functionality required for an interaction, they cannot interact properly and require adaptation. As the component API is limited to the syntactical representation of the public component types and methods, the adaptation of protocol mismatches requires additional specification of the components' behavior interface and of the valid mapping of component messages (Fig. 5.2). Listing 1 shows the behavior interfaces of *LAN* and *Node* specified using the (simplified) formalism of Yellin and Strom [24]. The interfaces (called *collaborations* in the listing) are described as a set of sent and received messages augmented by a finite state machine specification. The latter defines the legal sequences of messages that can be exchanged between a component and its mate (*init* stands for the initial state, - for sending and + for

```
 1  Collaboration Node {
 2    Receive Messages{
 3      mayEnter();
 4      maynotEnter(String:why);
 5      mayExit();
 6      maynotExit(String:why);
 7      broadcast(String message);
 8    };
 9    Send Messages{
10      enterReq(Node thisNode);
11      enter();
12      exitReq(Node thisNode);
13      exit();
14      ceased(Packet packet);
15    };
16    Protocol{
17      States{1(init),2,3,4,5};
18      1: -enterReq -> 2;
19      2: +maynotEnter -> 1;
20      2: +mayEnter -> 3;
21      3: +broadcast -> 3;
22      3: -ceased -> 3;
23      3: -exitReq -> 4
24      4: +mayExit -> 6;
25      5: +maynotExit -> 3;
26      6: -exit -> 1;
27    }
28  }
29  Collaboration LAN {
30    Receive Messages{
31      enter(Node node);
32      exit(Node node);
33      broadcastAck();
34    };
35    Send Messages{
36      broadcast(String message);
37    };
38    Protocol{
39      States{A(init),B};
40      A: +enter -> A;
41      A: +exit -> A;
42      A: -broadcast -> B;
43      B: +broadcastAck -> A;
44    }
45  }
```

List. 1: Component Behavior Specification

receiving messages). For instance, after sending the enter request, *Node* may only accept either *maynotEnter* (line 19) or *mayEnter* (line 20) messages and, in the latter case, can enter the network and broadcast packets (line 21). For clarity, the states of *Node* are marked by numbers (1–6) and those of *LAN* by letters (A, B).

For these two behavior interfaces List. 2 provides a mapping specification *Node-LAN* relating their states and messages. For each permitted combination of component states (numbers of *Node* and letters of *LAN* enclosed in angle brackets) it specifies the allowed transitions of both components and the valid memory state M of messages being exchanged. For instance, after receiving an enter request from a node (line 2), the adapter by default allows the node to enter the network (line 4) and, when getting the node's *enter* message, actually injects it into the LAN (line 6).

To keep track of the components' states, the adapter updates them correspondingly (again enclosed in angle brackets). To properly exchange the message data, certain method parameters need to be saved by the adapter. For example, *thisNode* sent by *Node.enterReq* may be sent to *LAN* only when the node actually enters the network (execution of *Node.enter()*). Meanwhile the adapter saves the parameter internally (shown as a call *write* to the implicitly generalized virtual memory). When the saved parameters are not needed anymore, they are deleted from the virtual memory by the adapter, as denoted by *invalidate*. Note also the handling of message absence (*broadcastAck* to *LAN*, *mayEnter* and *mayExit* to *Node*) and surplus (*ceased* from *Node*).

3.2 Refactorings Invalidate Specifications

Assume the specification is valid and the second release of the component was deployed successfully. However, in release 3 the developers realized that the *broadcast* method does not belong to every node of a network, but only to certain ones. In particular, workstation nodes may create and then broadcast packets, but print servers may not. To properly model this design requirement and avoid the abuse of the method, the developers applied the `ExtractSubclass` refactoring, effectively subclassing *Node* with a *Workstation* class and moving (pushing down) the method *broadcast* to the latter, as shown in Fig. 2 on page 4.

The refactoring will clearly invalidate the interface specification of *Node* (lines 7 and 21) and, hence, a part (lines 7–9) of the existing *NodeLAN* specification (Fig. 5.3). The situation aggravates with the growing number of releases and chaining of interdependent modifications. For instance, the developers of *Node* could later (1) rename (`RenameMethod` refactoring) the method *broadcast* to *originate* in order to reuse it also for multicasting, and (2) add a parameter to *maynotEnter* (`AddParameter` refactoring) to specify the reason of rejecting the entrance.

In general, any syntactic change affecting components referred to in specifications will invalidate the latter. The reason is that by changing the syntactical representation, refactoring modifies the initial context on which the specifications depend.

```
 1  /*LEGEND M0={} M1={thisNode} M2={message}*/
 2  <1,A,M0>: +enterReq from Node -> <2,A,M1>, write(thisNode);
 3  <2,A,M1>: -mayEnter to Node -> <2,A,M1>;
 4  <2,A,M1>: +enter from Node -> <3,A,M1>;
 5  <3,A,M1>: -enter to LAN -> <3,A,M0>,
 6            node = read(thisNode), invalidate(thisNode);
 7  <3,A,M0>: +broadcast from LAN -> <3,A,M2>, write(message);
 8  <3,A,M2>: -broadcast to Node -> <3,B,M0>,
 9            message = read(message), invalidate(message);
10  <3,B,M0>: +broadcastAck to LAN -> <3,A,M0>;
11  <3,A,M0>: -ceased from Node -> <3,A,M0>;
12  <3,A,M0>: -exitReq from Node -> <4,A,M1>, write(thisNode);
13  <4,A,M1>: +mayExit to Node -> <4,A,M1>;
14  <4,A,M1>: -exit from Node -> <5,A,M1>;
15  <5,A,M1>: +exit to LAN -> <5,A,M0>,
16            node = read(thisNode), invalidate(thisNode);
```

List. 2: Mapping Specification

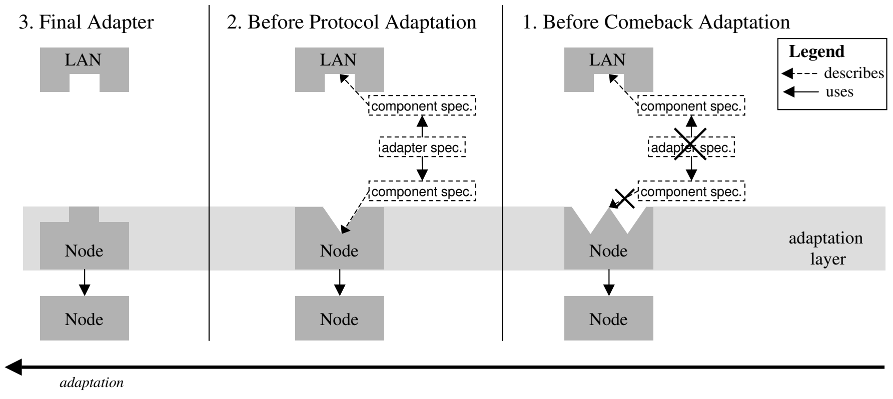

Fig. 6. Component Adaptation. The comeback execution reestablishes the context of adaptation specifications. They can then be executed directly on adapters to avoid performance overhead.

Either the specifications have to be updated to match the new context or the context itself has to be recovered. We perform the latter using comebacks.

3.3 Rescuing Specifications by Comebacks

As described in Sect. 2, the execution of comebacks in reverse order with regard to their corresponding refactorings yields an adaptation layer, which wraps new component functionality in terms of an old API syntax. Besides automatically bridging the signature mismatches, this execution leads to another important result – it reestablishes the context for previously written adaptation specifications. The intuition is that by effectively inverting refactorings on adapters, comebacks reconstruct the "right" syntactic names used in the specifications [22].

Figure 6.1 shows the state of adaptation before executing the comeback of `ExtractSubclass`. In between the *LAN* and *Node* components there is the adapter component (also called *Node*), possibly created in a step-wise manner by previously executed comebacks. At this stage, the *NodeLAN* specification is not (yet) valid. Now the comeback of `ExtractSubclass` transforms the adapter and leads to the situation shown in Fig. 6.2. Because the comeback inverted its corresponding refactoring on the adapter, the latter has exactly the same syntactic form as the original component *Node* before `ExtractSubclass` in Fig. 5.2. At this stage *NodeLAN* becomes valid again and can be used to derive the corresponding adapter. Intuitively, the comebacks reconstruct backwards the public types to which the specification relates; the only difference is that now (some of) these types are adapters. As a consequence, the developer can specify durable adapters at the time of the actual change.

There is still an important difference between the two middle parts of the Figs. 5 and 6: while in Fig. 5.2 the adapter is generated between two initial components, in Fig. 6.2 it is generated between the LAN component and the previously derived adapter leading to stacking of adapters. Although not affecting the actual functionality, adapter stacking may considerably decrease the performance due to the additional layer of redirection. In systems where the quality of service requires a certain level of performance, the performance penalties may become unacceptable.

In our ComeBack! tool we address this performance issue by integrating the protocol adaptation machinery into the existing refactoring-based environment. Given protocol adaptation specifications, the validity of which is proved at the specification level, we translate them into a corresponding set of Prolog facts and rules. The execution of these rules at the time of protocol adaptation updates the (adapter) fact base created by previously executed comebacks and protocol adapters. Because the description of the protocol adapter is thus effectively embedded into the fact base, no separate adapter is required (Fig. 6.3).[3]

Technically, because we assume correctness of specifications, we do not have to implement their state machines. By construction, the client components behave according to the state automaton before the upgrade and a valid specification does not change that. In fact, the only state the generated adapters maintain at runtime is the pointer to their delegatees and a storage for temporarily saved messages to implement the *write* and *read* operations of List. 2. As a consequence, the task of integrating both adaptation approaches is reduced to generating custom adapter methods that (1) read and write message contents to a local backing-store, such as a hashtable (to save and retrieve data exchanged), (2) perform actual delegation, (3) send default messages (in case of message absence), or (4) do nothing (in case of message surplus).

As an alternative to the backward comeback execution that recovers the specification context, the specifications themselves could be updated to match the updated components. For each refactoring one could define a special operator to update affected specifications. A (forward) execution of these operators along the refactoring history would produce the correct specifications. Although this approach is appropriate for the formalisms that combine signature and protocol adaptation in the same specifications (e.g., [9, 16]), it has several drawbacks. First, one needs to specify signature adapters manually, which is avoided in the comeback approach. Second, the operators for specification update would be dependent on and need to be rewritten for any new formalism used. Third, and most important, the formalisms themselves would need to be extended, since none of them provides any means for reflecting such changes in the specifications (e.g., that some functionality of *Node* is to be found in *Workstation* in the new release).

4 Evaluation

We performed two case studies using small-to-medium size frameworks. Using API documentation and clients, for each framework we investigated changes between four of its major versions. Gradually, the plugins of the first major version were manually adapted to compile and run with the second, third and fourth framework versions. Whenever possible, such adaptation was performed by refactoring. In such a way we discovered exactly the backward-incompatible framework changes and, where possible, modeled them as refactorings. For all refactorings detected the corresponding

[3] One of the main requirements of our technology – the total order of changes in the component history – is realized using timestamps.

comebacks were specified. For the remaining changes we investigated whether they affected message exchange between clients and the framework.

In the first case study we investigated JHotDraw [4], a well-known Java GUI framework. We used its versions 5.2, 5.3, 5.4, and 6.0 as well as four sample clients delivered with the version 5.2. We discovered eight backward-incompatible changes between versions 5.2 and 5.3, three changes between 5.3 and 5.4 and one change between 5.4 and 6.0. In total 12 changes were discovered, 11 (92%) of them being refactorings. The exception was the change of a collection type from the built-in Java *Enumerator* to the JHotDraw user-defined *FigureEnumerator*. It could neither be modeled as a refactoring nor as a protocol mismatch.

For the second case study we used SalesPoint [6], a framework developed and used at our department for teaching framework technologies. We used the framework versions 1.0, 2.0, 3.0 and 3.1 as well as two clients (student exercises) developed with the framework version 1.0. The first client reused mostly business logic of the framework. In total 48 changes were discovered, 47 (97.9%) of them being refactorings. The remaining one was the change of a default action (throwing an exception instead of doing nothing) and we compensated for it with the help of the corresponding protocol adapter.

Besides reusing the framework's business logic, the second SalesPoint client also made heavy use of the framework's GUI facilities, which changed considerably between the framework versions. Whereas in version 1.0 the GUI event handling was implemented solely by the framework, in version 2.0 (due to the switch to a new Java version) its implementation was based on the Java event handling. As a consequence, for the second client only about 70% (49 in total) of the changes could be considered refactorings. About 90% (16 in total) of the remaining backward-incompatible changes were protocol changes, in particular, of event handling in the GUI. Since we intended to explore the possibility of combining our refactoring-based adaptation with protocol adaptation, and not the full-fledged implementation of the latter, we implemented protocol adapters only for two selected protocol mismatches.

All in all, our results confirm the importance of refactorings for API evolution of software components previously pointed out by Dig and Johnson [13]. Moreover, in our case studies most of the remaining changes beyond refactorings could be modeled as protocol changes. However, the complexity of protocol adaptation in combination with refactoring-based adaptation may vary considerably depending on change particularities and therefore needs further investigation.

5 Related Work

As mentioned in the Introduction, most of the adaptation approaches [7, 9, 11, 16, 18, 21, 24] require additional adaptation specifications from developers. Recent research focused on how specific properties of certain modifications (e.g., behavior preservation of refactorings, deadlock-free protocol changes) can be proved statically and used to automate adaptation. For instance, the "catch-and-replay" approach [15] records the refactorings applied in an IDE as a log file. The file can be either delivered to the application developer, who "replays" refactorings on the application code (invasive adaptation), or used to generate binary adapters [22, 14]. These approaches, however, do not discuss adaptation of changes beyond refactorings.

In the same way, behavior adaptation is driven by the idea of capturing important behavioral component properties to reason about protocol compatibility, deadlock freedom, and synchronization of distributed components as well as to (semi-) automatically derive adapters and check for their correctness. Building on the seminal paper of [24], a number of approaches used, for example, label transition systems [10], message system charts augmented with temporal logic specifications [16], or process algebra [9] for specification of behavior and adapters. Although several approaches (most notably [9, 16]) address signature mismatches as well, their specifications are embedded into behavior specifications and suffer from the same maintenance problems we discussed in this paper.

6 Conclusion

Besides automatically adapting most of the component changes, our comeback approach fosters component maintenance in two ways. First, because refactoring-based adaptation only requires the new component API and the refactoring history, developers do not need to specify and later maintain specifications to compensate for structural changes. Second, for remaining changes of components' behavior developers are able to write in-time and durable protocol adaptation specifications, which are easier to specify and do not need to be maintained. Their automatic adaptation is implied by the particularities of our approach and comes "for free," as no additional means is required. We stipulate that the so-alleviated adaptation and maintenance not only reduce the costs and improve the quality of component upgrade but also relax the constraints on the permitted API changes allowing for appropriate component evolution.

Acknowledgement. We thank Mirko Seifert and our anonymous SERA'08 reviewers for their valuable comments.

References

1. Borland JBuilder. `http://www.codegear.com/products/jbuilder`
2. ComeBack! homepage. `http://comeback.sf.net`
3. Eclipse Foundation. `http://www.eclipse.org`
4. JHotDraw framework. `http://www.jhotdraw.org`
5. LAN-simulation lab session. `http://www.lore.ua.ac.be`
6. SalesPoint homepage. `http://www-st.inf.tu-dresden.de/SalesPoint/v3.1`
7. Balaban, I., Tip, F., Fuhrer, R.: Refactoring support for class library migration. In: OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, pp. 265–279. ACM Press, New York, NY, USA (2005)
8. Becker, S., Brogi, A., Gorton, I., Overhage, S., Romanovsky, A., Tivoli, M.: Towards an engineering approach to component adaptation. In: R.H. Reussner, J.A. Stafford, C.A. Szyperski (eds.) Architecting Systems with Trustworthy Components, *Lecture Notes in Computer Science*, vol. 3938, pp. 193–215. Springer (2004)

9. Brogi, A., Canal, C., Pimentel, E.: Component adaptation through flexible subservicing. Science of Computer Programming **63**(1), 39–56 (2006)
10. Canal, C., Poizat, P., Salaün, G.: Synchronizing behavioural mismatch in software composition. In: R. Gorrieri, H. Wehrheim (eds.) FMOODS, *Lecture Notes in Computer Science*, vol. 4037, pp. 63–77. Springer (2006)
11. Chow, K., Notkin, D.: Semi-automatic update of applications in response to library changes. In: ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance, p. 359. IEEE Computer Society, Washington, DC, USA (1996)
12. Dig, D., Comertoglu, C., Marinov, D., Johnson, R.: Automated detection of refactorings in evolving components. In: ECOOP'06: European Conference on Object-Oriented Programming, *Lecture Notes in Computer Science*, vol. 4067, pp. 404–428. Springer (2006)
13. Dig, D., Johnson, R.: The role of refactorings in API evolution. In: ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), pp. 389–398. IEEE Computer Society, Washington, DC, USA (2005)
14. Dig, D., Negara, S., Mohindra, V., Johnson, R.: ReBA: Refactoring-aware binary adaptation of evolving libraries. In: ICSE'08: International Conference on Software Engineering. Leipzig, Germany (2008). To appear
15. Henkel, J., Diwan, A.: Catchup!: capturing and replaying refactorings to support API evolution. In: ICSE '05: Proceedings of the 27th International Conference on Software Engineering, pp. 274–283. ACM Press, New York, NY, USA (2005)
16. Inverardi, P., Tivoli, M.: Software architecture for correct components assembly. In: Formal Methods for Software Architectures, *Lecture Notes in Computer Science*, vol. 2804, pp. 92–121. Springer (2003)
17. Johnson, R., Foote, B.: Designing reusable classes. Journal of Object-Oriented Programming **1**(2), 22–35 (1988)
18. Keller, R., Hölzle, U.: Binary component adaptation. In: ECOOP'98: European Conference on Object-Oriented Programming, *Lecture Notes in Computer Science*, vol. 1445, pp. 307–329. Springer (1998)
19. Opdyke, W.F.: Refactoring object-oriented frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA (1992)
20. Reussner, R.: The use of parameterised contracts for architecting systems with software components. In: J. Bosch, W. Weck, C. Szyperski (eds.) WCOP'01: Proceedings of the Sixth International Workshop on Component-Oriented Programming (2001)
21. Roock, S., Havenstein, A.: Refactoring tags for automatic refactoring of framework dependent applications. In: XP'02: Proceedings of Extreme Programming Conference, pp. 182–185 (2002)
22. Şavga, I., Rudolf, M.: Refactoring-based adaptation for binary compatiblity in evolving frameworks. In: GPCE'07: Proceedings of the Sixth International Conference on Generative Programming and Component Engineering, pp. 175–184. ACM, Salzburg, Austria (2007)
23. Şavga, I., Rudolf, M., Lehmann, J.: Controlled adaptation-oriented evolution of object-oriented components. In: IASTED SE'08: Proceedings of the IASTED International Conference on Software Engineering. ACTA Press, Innsbruck, Austria (2008)
24. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM TOPLAS: ACM Transactions on Programming Languages and Systems **19**(2), 292–333 (1997)