Teaching Old Services New Tricks: Adding HATEOAS Support as an Afterthought

Olga Liskin, Leif Singer, Kurt Schneider
Leibniz Universität Hannover
Software Engineering Group
Welfengarten 1, D-30167 Hannover, Germany
+49 (0) 511 762 19667
{olga.liskin,leif.singer,kurt.schneider}@inf.uni-hannover.de

ABSTRACT

Hypermedia as the Engine of Application State, or *HATEOAS*, is one of the constraints of the REST architectural style. It requires service responses to link to the next valid application states. This frees clients from having to know about all the service's URLs and the details of its domain application protocol.

Few services support HATEOAS, though. In most cases, client programmers need to duplicate business logic and URL schemas already present in the service. These dependencies result in clients that are more likely to break when changes occur. But existing services cannot be easily updated to support HATEOAS: Clients could cease working correctly when a service is changed. Also, client developers might not have access to the service's source code, be it for technical or political reasons.

We discuss which information is needed to create a HATEOAS-compliant wrapper service for an existing service. We include a notation for modeling possible application states and transitions based on UML State Charts. We demonstrate the feasibility and advantages of our approach by comparing the clients for an existing service and its wrapped counterpart. Our approach enables client developers to wrap third-party services behind an HATEOAS-compliant layer. This moves the tight coupling away from potentially many clients to a single wrapper service that may easily be regenerated when the original service changes.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Service-oriented architecture (SOA) – *REST, HATEOAS.*

General Terms

Design, Reliability.

Keywords

Services, Hypermedia, HATEOAS, REST, Wrapper.

1. INTRODUCTION

Services encapsulate functionality behind an interface that ideally complies with open standards and is accessible over a network. The two most popular examples for service strategies are the *Web Services* standards (*WS-**) and the REST architectural style.

The former are more often found in enterprises that depend on comprehensive vendor support and have strict requirements concerning security, reliability, and similar aspects. They contain, for example, the WS-BPEL OASIS Standard [2] which permits the orchestration of services into executable business processes.

REST is neither a standard nor a technology, but an architectural style for building scalable networked applications. The style itself and especially the constraints required to implement it were first described by Fielding [4]. The most popular implementation of REST is the World Wide Web using the Hypertext Transfer Protocol.

Services are becoming more and more popular as means for decoupling systems from each other while at the same time making functionality and data available to all authorized applications on the network. While services may of course be called using general programming languages, specialized approaches exist. One of these is the aforementioned WS-BPEL, catering to enterprises. To combine publicly available services, several *mashup* tools are available, enabling even end-users to create new, albeit simple, applications from existing services. An example for such a tool is Yahoo! Pipes[1], which allows the user to connect services with operators and with other services.

Because services, akin to web pages, can easily be deployed on the Web and *Web 2.0* companies have more and more data available, publicly accessible services have risen in number in recent years. It is an ongoing discussion as to which degree these fulfill the REST constraints, but ProgrammableWeb[2] provides some rough statistics, showing that concerning public service APIs, the REST style is clearly dominant with 74% of all APIs listed on the site.

One attempt at bringing some order into the discussion of whether a given service may be considered *RESTful* – satisfying the REST

[1]http://pipes.yahoo.com

[2]See http://programmableweb.com/apis, *Protocol Usage by APIs*

constraints – is Leonard Richardson's maturity model[3]which Webber et al. described in [12]. The model describes several levels of RESTfulness:

- Level Zero: service invocations are merely remote procedure calls using HTTP POST as transport, often using XML to encode messages.
- Level One: uses multiple resources, but encodes method name and parameters in the URL, often using the GET HTTP method.
- Level Two: uses multiple resources and HTTP methods and status codes instead of an additional proprietary scheme.
- Level Three: a Level Two service, but embraces hypermedia to communicate the next possible state transitions to the service consumer.

Hypermedia as the Engine of Application State, abbreviated HATEOAS, is the REST constraint that is satisfied by Level Three services. Services realizing this constraint supply links in their responses, e.g. as XML tags weaved into the message contents as proposed by [8] or as HTTP headers as shown by [5].

A network-based application may be seen as a state machine [4, 12], which is in a certain state at a certain instant. It can transition to different states from there. Depending on the current application state, certain transitions are valid or invalid. The idea of using hypermedia within server responses is to tell a client which transitions lead to valid subsequent states. By, for example, selecting a hyperlink from a current representation and following it, the application transitions to a new related application state. Using this method, a service is able to expose its domain application protocol [12]. It leads the client through its application states so that the client may reach a business goal [10].

If a client application is aware of HTTP methods, HTTP status codes and the media types used in a service, it will only need a single URL it can reach the service at [9]. All other operations may then be executed by state transitions the client performs when following links. This frees clients from having to know multiple URLs for the different resources a service provides.

More importantly, the client needs to know much less about the service's domain application protocol. When using a service that does not support HATEOAS, a client needs to calculate for itself whether to enable a specific action – for example, starting a *story card* in an agile development management system. The client would first need to determine whether the story is currently blocked by another user, whether the current user has sufficient privileges to do so, or whether the story has already been completed. These calculations of the client may be wrong and result in actions being made available that will fail when executed. Also, the correct calculations to be made might change over time, and a client that hasn't been updated yet would show impossible actions or fail to show some that are actually possible.

When communicating with an HATEOAS-enabled service, a client application only needs to look at the links provided in each service response. If the response for getting a story card does not include a link to start it, the client can trust the service to have determined this operation to not be possible.

Using a service that the client developer herself is able to change certainly is a special case – the general case being separated developers for clients and service, possibly even in different organizations. In particular, this should be the case when using public services. Still, a client developer would benefit from using the advantages of the HATEOAS constraint.

To satisfy this need, we provide a notation based on UML State Charts which is suitable for modeling the behavior of services. As a theoretical basis for these models, we define equivalence classes for application state. From such a model and some additional technical information, we are able to automatically create a wrapper service that mimics the original service in every way, only adding HATEOAS-conformant links into service responses.

A simplified example of such a model is shown in Figure 1. Based on this model the wrapper service can see that, for example, a story that is in progress can be finished or cancelled next.

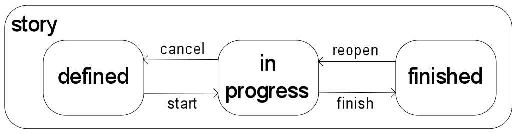

Figure 1: A state chart illustrating valid transitions for a story

The rest of this paper is organized as follows: the next section defines equivalence classes for application states, refining some of the terminology used in the REST community in the process. Section three describes the notation we developed from UML State Charts that is able to model these equivalence classes of services. Section four outlines our implementation that takes a service model and technical information as its input and provides a service wrapper from this. The next section compares the client written for an example non-HATEOAS service and the equivalent client needed for a service wrapped using our approach. The final section provides conclusions and an outlook on further research.

2. EQUIVALENCE CLASSES FOR APPLICATION STATES

To model which states and transitions exist in a networked application, we found it useful to introduce the notion of equivalence classes for application states. They are able to address variations in an application's handling of similar, but not equal resources.

We will stay with the above example of an application for managing story cards and tasks that each belong to a certain story. For certain stories, the available actions might be to block them or to finish them. For other stories, only unblocking them might be available.

Obviously there are at least two different kinds of stories that superficially differ from each other in the actions that may be applied to them. Internally, that difference may be a matter of arbitrarily complicated business rules.

To be able to model this circumstance, we introduce equivalence classes for application states later in this section.

2.1 Terminology

As the use of certain terms in literature and public discussions varies a lot or is unclear at times, we first discuss what an application is and what application states are. To differentiate, we

[3] Richardson presented the model at his Nov 20th 2008 QCon talk "Justice Will Take Us Millions Of Intricate Moves", see http://crummy.com/writing/speaking/2008-QCon/act3.html

also discuss the related terms of client state and resource state. Finally we introduce the term *domain application protocol*.

Application: In the context of the Web and services on the Web, an application is usually distributed between different physical components: a server and a client in the simplest case. For our purposes, we may ignore intermediary actors like proxies or gateways.

Fielding [4] and Umar [11] define such a network-based application as "a representation of the business-aware functionality of a system". Webber et al. [12] regard an application as "computerized behavior that achieves a goal".

According to these descriptions, an application includes all the requests and responses of all of the components that communicate in order to achieve a goal. Also, the focus is not on the network level, which is merely a communications medium. Instead, an application is about the more abstract business level.

Application State: In order to achieve a goal, the components must perform various actions that include requests, responses, and the processing of messages and data. *Application state* summarizes the state of the different components in one instant. According to Fielding [4], an application's state can be defined as "its pending requests, the topology of connected components (some of which may be filtering buffered data), the active requests on those connectors, the data flow of representations in response to those requests, and the processing of those representations as they are received by the user agent."

Fielding also uses the notion of a steady-state of an application which is reached when the network-based application "has no pending requests and all of the responses to its current set of requests have been completely received […]" [4]. For our approach, only the steady-states of an application are relevant.

A first classification of these states can be derived from the representation of the resource the application was retrieving when it reached the steady-state. For example, *examine a story number 23* and *examine story number 17* are two different application states.

Furthermore, it is important to distinguish *application state* from other kinds of state that are also prevalent in RESTful architectures.

Client State: Client state only includes the state of the client component of a network-based system. Since it does not regard the application as a whole, the domain application protocol is also not within its focus.

Resource State: The main entities of abstraction in RESTful architectures are resources. These are objects with attributes that reside on the server and are communicated and manipulated using representations. The state of a resource is defined by the values of its attributes and connected resources [12]. This is deeply connected with application state, which can partly be determined by resource state.

Domain Application Protocol: As we have seen now, the notions of application and application state provide an abstract perspective on a networked system. They are focused on business goals and use cases related to these goals. The business logic, that is located on that level as well, is described by *domain application protocols*. Domain application protocols have been introduced by Webber [12]. They describe the valid interactions between clients and a server involved in a business process.

2.2 Equivalence Classes for Application States

From each state, an application may transition into a number of different subsequent states. The set of these subsequent states is determined by the current application state. For example, a blocked story card in the agile development management application cannot be finished, but needs to be unblocked first.

A model of an application can be created by compiling a list of all application states and the possible transitions between them. Such a model shows the operation of the application and how a series of transitions may lead to reaching a higher level goal. It especially helps in understanding the domain application protocol.

To create this model, the hardest part seems to be creating the list of possible application states. After all, an application is practically in a different state after each request that is fulfilled. For example, when a user creates 10 new story cards and after each creation returns to a list view of all story cards, the whole process entails 20 different states. Additionally, having created 10 new story cards means also having created at least 10 new possible application states: one for looking at each new story card. Obviously, the number of possible application states is indefinite and may change at any time.

We solve this problem by grouping application states by similarity, where our metric for similarity is an equivalence relation. This creates a set of groups that is finite and fixed. Therefore, a static model of the application can be created, describing the application's business rules.

Many states are not exactly the same but very similar. For example, the application states *examine story 23* and *examine story 17* are not exactly the same application states but have many things in common. They are both about examining the same type of resource, namely a story. Most importantly, the application permits the same actions to be applied to these stories. We will assign these two states to the same equivalence class because the application logic treats them the same.

The equivalence relation should divide application states into partitions for which the business logic allows the same outgoing transitions. At the same time, application states that permit different transitions should be assigned to different classes. Furthermore, we require the media types of the representations used in the different states to be the same.

We understand a media type as the structure of representations. This is similar to different mime types, but goes further: if both a story and a task are represented in XML without referencing XML Schema definitions, a machine might not be able to tell apart these representations. However, a client that understands the different media types will be able to tell them apart, anyway – in that case, we consider the media type to be implicitly agreed upon.

The equivalence relation that partitions all possible application states as described above is the following:

Two application states are considered equivalent iff the domain application protocol treats them the same, i.e., the same actions are applicable when the application is in one of those states. Also, the media types of the representations the application states use must be the same.

For example, *examine a list of stories* is one equivalence class of application states. A story list with five stories will be assigned to this class as well as a story list with seven elements. The application treats both elements the same: it presents a listing of all story elements, allows navigating to each element, and permits adding new elements to the list.

Examine a task is a different equivalence class. It does not display a list of the stories, but the detail information of one particular story. In our example, it is not possible to add a new story from this application state. However, the list of all the story's tasks and the list of all stories of the development project are reachable from this state.

The example shows that an equivalence class is mostly determined by the *class of the resource* (list of stories, story, list of tasks, or task) which has been exchanged for this application state.

Also, the *state of a resource* is important. Depending on the state of a resource, different outgoing transitions might be possible. For example, while the name of a story card might be irrelevant in this regard, the status attribute of a story records whether the story is *not started*, *in progress*, *blocked*, or *finished*. A story card that is blocked cannot be finished before it has been unblocked.

It is therefore necessary to distinguish between certain resource states when dividing application states into equivalence classes. For the given example, there might be two equivalence classes along the lines of *examine a defined story, examine a blocked story, examine a story that is in progress,* and *examine a finished story*.

We divide application states into fixed and finite partitions. These partitions are determined by the outgoing transitions of application states. These, in turn, are determined by resource class and resource state. We are now able to create a static model for the application. The following section presents our notation for doing so.

3. MODELING NETWORKED APPLICATIONS

In order to create a service that wraps an existing service and adds HATEOAS hyperlinks into responses, the wrapper needs information about the different application states and the valid transitions between those. It is possible to provide this information in a tabular way, but the table can quickly become very complex. Also, it does not link related/connected states graphically.

Therefore, we chose to create a graphical notation for application states and transitions. Another benefit of this is that the domain application protocol of the described service can be presented more clearly.

Fielding already described an application as a state machine which transitions between states with every different page that is requested by the client. Webber uses state machines to describe the implementation of business goals with hypermedia. Tilkov uses state charts to illustrate the states a resource may traverse – however, this is limited to a single class of resources [10].

A powerful basic notation for state charts is the UML State Charts [7] notation. Particularly additional elements like composite states and pseudo vertices are helpful for the required design. We therefore based our notation on UML State Charts.

The states in the state chart represent the possible application states, or, more precisely, the equivalence classes of application states. The transitions in the state chart represent possible transitions between application state equivalence classes.

At this point, the introduction of equivalence classes from the previous section becomes a useful tool. Let *examine blocked story* be an equivalence class of application states represented by a state element in a UML State Chart. Whenever the application is in a state that belongs to this equivalence class, the same types of transitions – like *unblock story* and *show tasks* – will be valid. The state transitions in the model that start at the *examine blocked story* state represent those two transitions that are valid for all *blocked stories*.

As mentioned before, we use UML State Charts as a basis for our notation. We now describe the adjustments in the semantics of the elements we had to make.

3.1 States

States will represent the equivalence classes of application states. The equivalence class of an application state is determined by the resource class and the resource state of the currently examined resource.

Therefore, some equivalence classes may belong to the same resource class. For example, *examine blocked story* and *examine finished story* are both associated with the resource class *story*. It is possible that all equivalence classes belonging to the same resource class have some transitions in common. For example, the list of a story's tasks should be accessible independent from the story's resource state.

We model this situation by grouping certain simple states into composite states. These composite states represent all states for a single *resource class*. They have multiple sub-states which represent the different possible *resource states* that are relevant for the application's behavior. An example for a story is shown in Figure 2. However, it is also possible that a resource class has no distinguishable sub-states. In this case, the state for the resource class itself is a simple state.

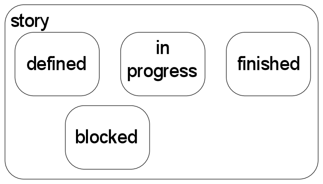

Figure 2: Multiple sub-states that are associated with stories are grouped into a composite state.

Whenever an application is in a composite state, it actually is in one of the sub-states. For formal reasons, we further require a distinguished *default state* as a sub-state of each composite state. The exact reasons for this will be explained in section 3.3.

3.2 Transitions between Simple States

All transitions having a simple state as their sources represent a possible transition of the application state from this particular class of states. We mark such transitions with labels that describe the purpose of the transition in the application's terms – e.g., *unblock story*.

The target of a transition is the expected application state after the transition has been followed by the client. If the target is a simple state, it describes the new application state completely. This contrasts with the behavior for composite states.

In network-based applications, it is not guaranteed that the expected application state will actually be reached. Instead, it is possible that the application changes into a different state, like an error-state. The service wrapper will not be using the targets of transitions so that it is sufficient to model the *expected* target states for the successful situation.

The wrapper itself is stateless and only needs to enable the client to reach one of the next possible application states by following hyperlinks. Due to its statelessness, the wrapper is unable to connect the current application state with the links it provided in a prior request.

3.3 Transitions between Composite States

Transitions between composite states have the same function as transitions between simple states. They indicate valid state transitions of the application. However, a composite state represents a resource class independently of its different sub-states. Accordingly, transitions between composite states are usually independent of resource states and are valid for the whole resource class. To be able to model control flow regardless, we will now give more concrete semantics for these kinds of transitions.

Transitions having a complex state as their *sources* denote that the according state transition can be initiated from *every* sub-state of the composite state. For example, it is possible to transition from a story to its tasks independently of the story's state – it does not matter whether the story is blocked or not. The model would then contain a single transition from the composite state *story* instead of one transition for each sub-state of *story*. This definition conforms to the usual semantics of UML State Charts.

However, transitions that have complex states as their *targets* are a little more complex. In UML, a composite state needs a *start vertex* that defines which actual state is activated once control flow reaches a composite state. If there is no such start vertex in a composite state, the UML semantics dictate that control flow proceeds to an arbitrary sub-state. For our purposes, this behavior is not desirable.

When a client follows such a transition to a complex state, it cannot be determined beforehand which of the sub-states will actually be reached. Only after the client has requested the transition from the server and the server has responded may the actually reached state be found out. To model this behavior, we require that for each complex state, a start vertex exists that has a single transition to a *choice pseudostate* as defined in the UML. Such a state has multiple target states which all have a condition attached to their transitions. The path for which the condition evaluates to *true* will then be taken.

For our model, this means that when a transition reaches a composite state, the control flow proceeds to the start vertex and then to a choice pseudostate at which the resource's exact state is evaluated. Depending on the result of this evaluation, the application transitions to the according sub-state. This is where we need the *default state* mentioned before: by specification, an *else branch* is recommended for the choice pseudostate. This recommendation is especially reasonable for our use case, as the target state cannot be determined beforehand. Also, when creating the model, human error might have left some states missing. Therefore, for each composite state's choice pseudostate, we require an else branch that transitions to the composite state's default state.

Figure 3 shows an example that illustrates a composite state *story* and transitions that have the state as a target or a source.

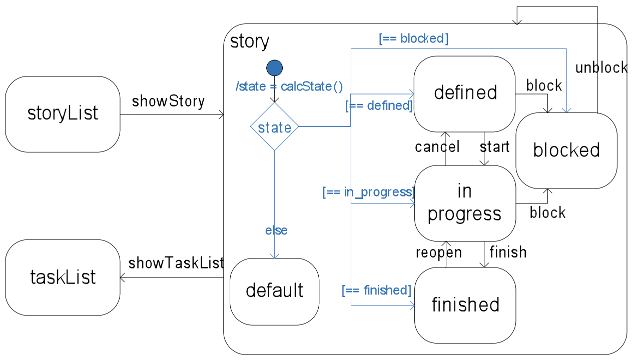

Figure 3: A composite state and its transitions

The outgoing transition *showTaskList* can be initiated from all sub-states of the *story* composite state. From the *storyList* state, it is possible to navigate to the individual stories using the *showStory* transition. When following that transition at runtime however, it is unclear which state a story currently is in. Only after the story has been looked up and examined, the application may transition to the actual sub-state.

3.4 A Simplified Notation for State Charts

The choice pseudostate is always used in the same manner. It connects the start vertex with the sub-states of a composite state. It is used to check whether a sub-state exists with name given by the variable *state* and transitions into it.

This construct is generic enough to be derived from the sub-states of a composite state alone. Therefore, we introduce an abbreviated notation of the construct which only contains sub-states but no start vertex, choice pseudostate or additional associations. We require these only implicitly. Using this convention, a model of an application becomes more readable and easier to create. Figure 4 shows the model from Figure 3 in abbreviated form.

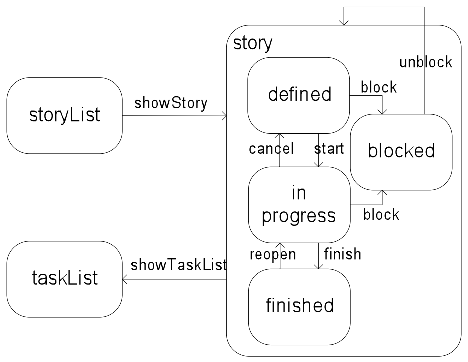

Figure 4: Abbreviated form of a composite state

By using a slightly customized version of UML State Charts to model networked applications, many tools are already available to create these kinds of models. As many software developers already use UML and corresponding modeling tools, only little additional knowledge is required for this process.

4. A GENERIC SERVICE WRAPPER

This section presents our implementation of a generic service wrapper. As its input, it takes the model of a networked application as introduced in the previous section, as well as some additional technical details. Using only this information, our implementation is able to wrap an existing service, adding an

HATEOAS layer in the process. The model of valid transitions for the equivalence classes of application state is used to derive which hyperlinks to inject into a given service response.

The wrapper service is placed between client and server and enriches the service responses with hyperlinks. These tell the client the next possible transitions after processing that particular response. Figure 5 shows the relations between the participating components and the way the wrapper works.

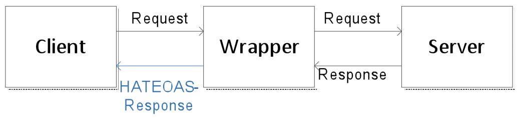

Figure 5: Insertion of the wrapper component

Instead of addressing its requests directly to the server, the client now sends its requests to the wrapper. The wrapper forwards the request to the server and receives the server response. It modifies the response and sends it to the client.

Currently, the wrapper only uses the server's response to find out which class of state the application will now transition into. While it is also possible to have the wrapper send additional requests to the server for finding out more about the response it's currently analyzing, we keep the scope of our solution more narrow for now.

The wrapper is stateless: it does not persist the application state the application has currently been in or just transitioned to. Instead, it analyzes each response anew as to which application state it will transition the application to. This is an important trait, as such networked applications might transition into new states without the wrapper witnessing it. For example, other clients still using the original service might change the resource state on the server.

For one request and response, the wrapper works according to the steps illustrated in Figure 6.

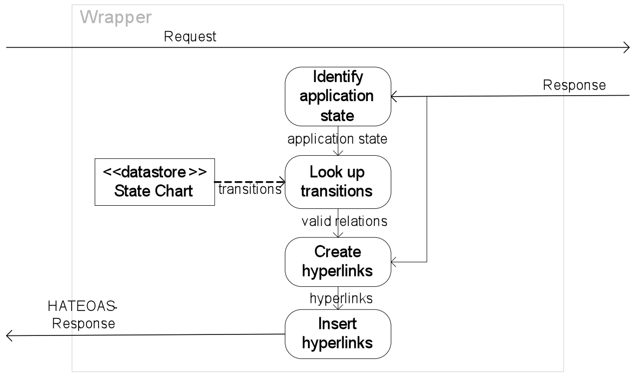

Figure 6: Activities of the wrapper process

The first step is to receive and forward the request. This step is not mentioned as an explicit activity, as it is purely technical. The other steps are conceptually more interesting and are described in more detail in the following paragraphs.

Identify application state

First of all, the wrapper needs to identify the current application state. It needs to determine the equivalence class of the application state in order to look up the transitions that are valid. For this, it needs to know the *resource class* and the *resource state* of the communicated resource.

In order to find out the application state, the wrapper can use information from the server response. The resource class as well as the resource state that belong to the current application state must be identifiable based on the representation of the communicated resource if they are relevant for the application logic. Otherwise it would be impossible for any client to handle the received representation correctly.

An example for a representation that takes the application into the state *examine blocked story* is shown in Listing 1.

```
<story>
  <id>5</id>
  <status>blocked</status>
  <description>…</description>
</story>
```

Listing 1: Exemplary representation of a story resource

The <story>-Tag identifies the *resource class* as *story*, and the <status>-Tag contains the necessary information about the *resource state*. For applications in which the representations do not contain such hints, further requests from the wrapper to the server might be needed.

In order to handle the server response, the wrapper must be able to extract information from the response message. Where exactly the required information may be found in the response must be provided as input to the wrapper. Our implementation is based on XPath2 [3] as a means to extracting information from documents, practically limiting our implementation to XML messages. Additionally to the application model, the wrapper takes XPath2 expressions as its input. These are used for retrieving a resource's class and for retrieving a resource's state. At runtime, the wrapper applies the expressions to the server response and receives the values for the current resource class and resource state. These define the equivalence class of the current application state.

Look up the possible transitions

The equivalence class determined in the previous step is represented by a state vertex in the application model. The next step for the wrapper is to look up the transitions that have the current application state equivalence class as their sources.

The transitions are identified by unique labels. The result of this step is a list that contains the labels of all transitions the application may consider next. The relation names are later used for the *rel* attribute within the hyperlinks created by the wrapper.

Create hyperlinks from transitions

From the discovered transition labels, concrete hyperlinks for following the transitions must now be generated.

For constructing hyperlinks, a schema with fixed and variable parts is required. This is provided by URI-templates that are associated with the transitions from the model.

Parts of some URIs must be filled with resource-specific information. The necessary specific information has to be extracted from the server response again. Like in earlier steps, we use XPath2 expressions that supply the required values when applied to the server response.

XPath expressions are able to return lists of values. It is therefore possible to create multiple hyperlinks for one type of transition. In our example, this is required when the application is in one of the list-related states, e.g. *taskList*. For each task of a story, a link to examining that task needs to be generated, with the only difference being the internal identifier of the task.

Insert hyperlinks into server responses

Finally, the created hyperlinks must be inserted into the response message and the new response must be sent to the client. The hyperlinks consist of the URI that has been created in the previous step and of a relation attribute that contains the transition label from the application model. We use HTTP *Link* headers [6] for the hyperlinks. This allows us to keep the original response body, minimizing the risk for unwanted modifications. Also, we reduce our dependency on any specific message formats the server uses.

The process performed by the wrapper is now complete. The required input data can, for example, be provided by a developer or any person that has used or wants to use the service and has some knowledge about the service's API and the application logic. Very often, a big part of the information can also be obtained from the documentation.

The following section discusses why this approach is worthwhile for a developer and which advantages emerge by providing this more abstract service information to a wrapper compared with integrating this information directly into client applications.

5. COMPARISON

The main benefit of the wrapper is a simplification of the development of clients for services not supporting HATEOAS.

The problem with creating a client for a service that does not provide HATEOAS links is that the client needs to know about much of the service's business logic. It needs to know which requests are allowed from which application state and how these requests are constructed. In addition to an increased probability for errors during development, further problems occur if the service is modified. A simple change in URI structures might render a client unusable.

In order to demonstrate the benefits of our approach, we compare two simple client applications: one that uses hyperlinks from server responses and one that does not.

The clients are Java applications that use the story card service we used as an example before. They display the details of a story and offer buttons for changing its status. The buttons that cannot be used at a certain instance have to be disabled. Valid statuses for a story in this example are *defined*, *in progress*, *blocked*, and *finished*. Valid transitions between statuses are shown in Figure 7.

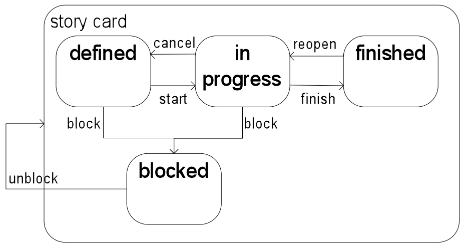

Figure 7: States and transitions of the exemplary service

A client that can use hyperlinks from the server response only needs to know the transition labels of the possible transitions. These labels are included in the hyperlinks' *rel* attributes. The presence of a link for a particular relation indicates a valid transition. The client just needs to follow the URI to perform that particular transition and does not need to construct the URI by itself. In our example, the client enables the respective buttons and sets its URI to the URI in the hyperlink. The code, as shown in Listing 2, is very generic.

```
private void updateControlPanel(ClientResponse response){
  disableAllButtons();
  List<String> links = response.getHeaders().get("Link");
  for(String link : links){
    if(link.contains("story.start")){
      String uri = extractUri(link);
      this.startButton.setUri(uri);
      this.startButton.setEnabled(true);
    }
    if(link.contains("story.finish")){
      String uri = extractUri(link);
      this.finishButton.setUri(uri);
      this.finishButton.setEnabled(true);
    }//...
  }
}
```

Listing 2: Client code that uses hyperlinks

From this point, one could go a step further and make the client even more generic. While in the example the client still knows which buttons – or transitions – are possible at all, it would also be promising to create a client that offers controls by using only the information in the link headers and no internal information. Such a client would create one button for each currently available link header. Then it could use the *rel* attribute of a link for a button's label. This would be very similar to how a web browser or a crawler as in the work of Alarcòn and Wilde [1] functions.

In contrast, a client that cannot use hyperlinks needs to identify the application state by itself. It needs to know the transitions that are currently valid according to the domain application protocol. Further, it has to construct all URIs by itself, using some internal knowledge about the patterns for the URIs. These issues are sources for errors during development and later on, should the service implementation change. Listing 3 demonstrates the example code for a client that has the same functionality as the client before. Error-prone areas are marked with bold letters.

```
private void updateControlPanel(ClientResponse response,
Story story){
  disableAllButtons();
  if(story.getStatus().equals(StoryStatus.Defined)){
    this.startButton.setEnabled(true);
    this.startButton.setUri("/stories/"+story.getId() +
      "/start");
    this.blockButton.setEnabled(true);
    this.blockButton.setUri("/stories/"+story.getId() +
      "/block")
  }
  if(story.getStatus().equals(StoryStatus.InProgress)){
    this.finishButton.setEnabled(true);
    this.finishButton.setUri("/stories/"+story.getId() +
    "/finish");
    this.blockButton.setEnabled(true);
    this.blockButton.setUri("/stories/"+story.getId() +
    "/block");
    this.cancelButton.setEnabled(true);
    this.cancelButton.setUri("/stories/"+story.getId() +
    "/cancel");
  }//...
}
```

Listing 3: Client code that does not use hyperlinks

This client is not as generic as the one before. It contains lots of hard-coded business logic, distributed among multiple locations. This makes the client more fragile as well as harder to develop and maintain. When we developed these very simple clients, that effect already became apparent: the most errors we made were located in exactly those areas we mention above.

The usage of a wrapper service as proposed creates advantages by enabling HATEOAS for services that do not support it by themselves. Another benefit is the centrality of the technical information and the application model given as input to the wrapper service. When these values need to be changed – e.g., because the service API has changed – these changes can be compensated quite fast in one place, as only essential information is stored. In many cases, clients may continue working as before – an exception being the addition or removal of transitions or resource classes. The benefits are even more apparent when multiple client implementations are using the wrapper.

6. CONCLUSIONS & OUTLOOK

We presented a novel approach for the introduction of HATEOAS links into services that do not support them. We provide a method and notation for capturing the domain application protocol of a given service. We showed how this model, along with some additional technical information, may be used to configure a generic wrapper service that injects HATEOAS links into a service's responses.

As we have discussed, this approach has several advantages for the development and maintenance of client applications. These become even more apparent when multiple implementations of clients exist. Apart from the developers of client applications, our approach might also be attractive to service providers wishing to add HATEOAS support to their services with comparatively little effort.

However, a client implementation still needs an understanding of the mime types and media types used by the service. Also, by using XPath2, we restrict our approach to services that use XML messages for communication. Approaches exist that convert JSON messages to XML – these seem worthwhile to explore, as more and more of the publicly available services switch their messages to JSON. To support these, our wrapper implementation would merely need to convert all JSON messages to XML messages before processing them. Alternatively, a path language for JSON could be used. However, as our approach requires some advanced features of XPath2 not present in XPath1, it might be possible that these languages are not powerful enough. We have not evaluated these options yet.

Apart from the potentially limited scope of our approach, its most serious drawback is the overhead it introduces into network communications. To some extent, this might be mitigated by placing the wrapper service into the clients' or the service's network, though. Other options exist – e.g., placing the wrapper directly into a client's code – but this would introduce tighter coupling again.

We do not yet have quantifiable data on the advantages of using our approach versus developing traditional clients or rewriting a service to use HATEOAS. Exploring these questions is one of our most important next steps.

7. REFERENCES

[1] Alarcon, R., and Wilde, E. Linking data from restful services. In *Third Workshop on Linked Data on the Web (LDOW2010)*, Raleigh, North Carolina, USA, April 2010.

[2] Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., et al. Web Services Business Process Execution Language Version 2.0. Tech. rep., OASIS, 4 2007.

[3] Berglund, A., Boag, S., Chamberlin, D., Fernández, M. F., Kay, M., Robie, J., and Siméon, J. XML Path Language (XPath) 2.0. online, last access: Feb 10 2011, 12 2010. http://www.w3.org/TR/xpath20/

[4] Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[5] Hadley, M., Pericas-Geertsen, S., and Sandoz, P. Exploring Hypermedia Support in Jersey. In *Proceedings of the First International Workshop on RESTful Design* (2010), ACM, pp. 10–14.

[6] Nottingham, M. Web Linking. Tech. rep., Internet Engineering Task Force (IETF), 10 2010.

[7] OMG. OMG Unified Modeling Language (OMG UML), Superstructure: Version 2.2. OMG Specification 2.2, Object Management Group, 2009.

[8] Parastatidis, S., Webber, J., Silveira, G., and Robinson, I. The Role of Hypermedia in Distributed System Development. In *Proceedings of the First International Workshop on RESTful Design* (2010), ACM, pp. 16–22.

[9] Richardson, L., and Ruby, S. *RESTful Web Services*. O'Reilly Media, Sebastopol, CA, USA, 5 2007.

[10] Tilkov, S. *REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien*, 9 ed. dpunkt Verlag, Heidelberg, Germany, 2009.

[11] Umar, A. *Object-oriented client/server Internet environments*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1997.

[12] Webber, J., Parastatidis, S., and Robinson, I. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, Sebastopol, CA, USA, 2010.