2010 Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems

Taxonomy-driven Adaptation of Multi-Layer Applications using Templates

Razvan Popescu*, Athanasios Staikopoulos*, Peng Liu*, Antonio Brogi† and Siobhán Clarke*
* *School of Computer Science and Statistics, Trinity College Dublin, First name.Last name@scss.tcd.ie*
†*Computer Science Department, University of Pisa, Last name@di.unipi.it*

***Abstract*—Current adaptation approaches mainly work in isolation and cannot be easily integrated to tackle complex adaptation scenarios. The few existing cross-layer adaptation techniques are somewhat inflexible because the adaptation process is predefined and static.**

In this paper we propose a methodology for the dynamic and flexible adaptation of multi-layer applications. We use events to trigger the process of matching adaptation templates, which expose adaptation logic as BPEL processes. The matching process employs taxonomies of adaptation mismatches to select adaptation templates based on the degree of match between events and adaptation mismatches. We provide support for cross-layer adaptation by allowing templates to be composed either directly, through invocations of WSDL operations or indirectly, through events.

***Keywords*-multi-layer applications; cross-layer adaptation; adaptation templates; matchmaking;**

I. Introduction

Complex applications are generally heterogeneous, loosely-coupled, long-lived and continuously running and have to cope with frequent changes to their requirements and environment. In order to address such changes, applications need to be inherently flexible and adaptive and supported by appropriate adaptation infrastructures. The adaptation process serves, for example, to ensure that the application is fault tolerant or compatible with new clients. The adaptation can be enforced either at design, or at run-time, and it can be triggered by the human designer or operator of the application, or by a monitoring process.

For instance, several various approaches have been defined for the adaptation of service-based applications. They generally tackle mismatches relating to signatures [1], behaviour [2], quality-of service [3], service-level agreement [4], or policies [5]. However, such techniques usually work in isolation, address single application layers and cannot be easily integrated to tackle complex adaptation scenarios [6]. Few adaptation approaches (e.g., [7], [8]) tackle the adaptation of multi-layer applications, yet even those are somewhat inflexible mainly because the adaptation logic is predefined and static.

In this paper we propose a methodology for the dynamic and flexible adaptation of multi-layer applications. Our adaptation approach comes as an attempt to enhance and semi-automate the adaptation of multi-layer applications primarily by combining:

- *Templates* (also known as patterns) that are already widely recognised as a powerful technique in software engineering in general and, more specifically, their use has already been successfully proposed for (behavioural) adaptation (e.g., [9]), and
- *Taxonomies of adaptation mismatches* (or issues) [10]), to semi-automate the discovery and selection of adaptation templates needed to fulfil complex (cross-layer) adaptation requirements.

In a nutshell, we use events to trigger the process of matching adaptation templates, which expose adaptation logic as BPEL processes [11] (see Figure 1). The matching process employs taxonomies of adaptation mismatches to select adaptation templates based on the degree of match (viz., *exact*, *plug-in*, *subsumes*, or *failed* [12]) between OWL-annotated [13] events and adaptation mismatches. When no adaptation templates corresponding to *exact* matches are found the matching process tries to find more general adaptation templates corresponding to *plug-in* matches. If no such templates are available then the matching process attempts to find more specialised templates corresponding to *subsumes* matches. Events, taxonomies and templates are bound to single application layers. In order to provide support for cross-layer adaptation we allow templates to be composed either directly, through invocations of WSDL operations [14] or indirectly, through events.

We illustrate in section III how our adaptation approach can be employed to solve a complex adaptation issue of a messaging client application from the social-networking domain. The application defines three layers: a service layer (viz., communication service), a behavioural layer (viz., high-level application behaviour as an orchestration of services) and an organisational layer (viz., application stakeholder roles and their objectives and relationships). The adaptation is triggered due to an *Invocation-fault mismatch* at the behavioural layer. The case study demonstrates how to flexibly discover (a set of) adaptation templates that solve the issue by replacing a faulty service, changing the stakeholder structure and modifying the application behaviour accordingly.

In summary, to the best of our knowledge our adaptation approach is the first one to offer all of the following features:

- Cross-layer adaptation,
- Taxonomies of adaptation mismatches,

978-0-7695-4232-4/10 $26.00 © 2010 IEEE
DOI 10.1109/SASO.2010.23

IEEE computer society

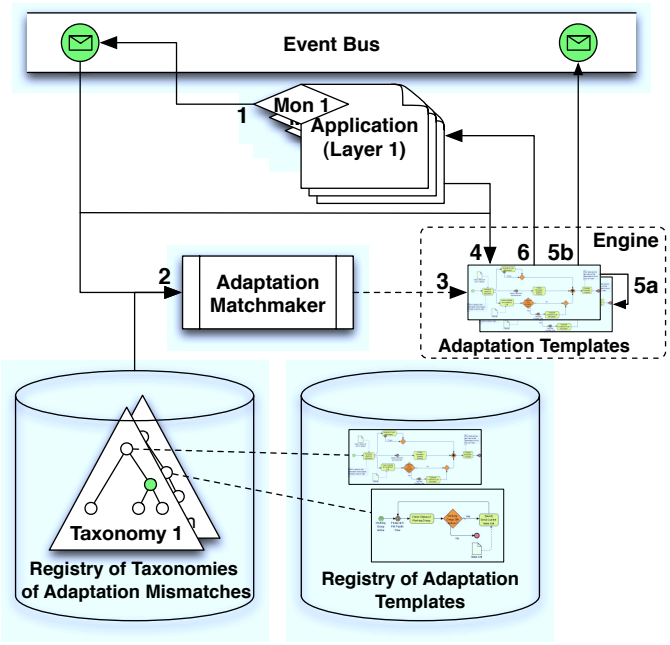

Figure 1. Conceptual architecture.

- Encapsulation of adaptation logic into templates associated to adaptation mismatches,
- Loosely-coupled event-based adaptation architecture,
- Composition of templates through direct invocations and events, and
- On-the-fly discovery of adaptation-templates through matching events and adaptation mismatches.

The remaining of this paper is structured as follows. Section II introduces our adaptation methodology. Section III briefly presents a case study. In section IV we describe related work followed by some concluding remarks in section V.

II. Adaptation Methodology

In this section we introduce the key ingredients of our approach (events, taxonomies of adaptation mismatches and adaptation templates) followed by the description of the core steps of our adaptation methodology.

A. Events, Taxonomies and Templates

Events. We assume that the multi-layer application to be adapted is being monitored at one or more layers by dedicated monitoring units [5], [8], [15], [16], [17]. These units are in charge of issuing events that trigger a desired adaptation. Example events may be *Role-name mismatch* (at the organisational layer), or *Message-ordering mismatch* (at the behavioural layer), or *Invocation mismatch* (at the service layer). We argue for the use of OWL concepts to describe the events.

Taxonomies of adaptation mismatches. In order to provide a flexible adaptation framework we classify adaptation techniques based on taxonomies of adaptation mismatches that they can cope with. For each application layer one may define one or more such taxonomies. These taxonomies may be either generic (e.g., at the service layer in service-based applications), or they may contain domain information for particular application domains (e.g., at the service layer in real-time service-based applications). Similarly to events, we advocate for the use of OWL concepts to describe adaptation mismatches.

We argue for the definition of tree-based taxonomies and *is a* relationships between children and parent mismatches. Furthermore, we assume that the degree of match [12] between the concepts referred to by any two mismatches t_1 and t_2 in a taxonomy is one of the following: *exact* match (viz., if t_1 and t_2 are the same mismatch), *subsumes* match (viz., if t_1 is an ancestor of t_2, that is, t_2 is a descendent of t_1), *plug-in* match (viz., if t_1 is an descendent of t_2, that is, t_2 is an ancestor of t_1), or *failed* match (viz., t_1 is neither an ancestor of t_2 nor one of its descendants).

By modelling taxonomies in this way it is likely that adaptation techniques corresponding to higher-level nodes in the taxonomy (viz., *plug-in* matches) can successfully cope with adaptation mismatches at lower levels of the taxonomy. Higher taxonomy nodes refer to bigger adaptation issues that require more radical changes. For example, when a service client triggers an *Input subsumes mismatch* (1.1.1.2.1.2 in Figure 4), the adaptation process can successfully employ an adaptation template that semantically matches and then replaces a given service based on a client query. Given the wide applicability of this adaptation technique to solve issues at the service layer, one may assume that a template implementing this technique can be associated to e.g., an *Interface mismatch* (1.1 in Figure 4). Dually, *subsumes* matches reflect the fact that the matched adaptation technique may still (partially) solve the adaptation issue. For example, given a *Message-ordering mismatch* event (1.1.1.1 in Figure 3) an adaptation template that can solve *Sequential vs. Sequential mismatch*es (1.1.1.1.1 in Figure 3) could be employed to solve the mismatch when the involved (required and provided) protocols do not employ conditional or iteration operators.

We describe hereafter parts of generic taxonomies for three possible application layers: organisation, behaviour and service.

Figure 2 describes part of a possible taxonomy of adaptation mismatches for the organisational layer. This layer provides a formalisation of the application roles (stakeholders) and their objectives and relationships (dependencies) needed to support the achievement of the objectives. This taxonomy employs organisational concepts defined in [18]. For example, a *Stakeholder-role mismatch* (1 in Figure 2) may be due to a *Role-name mismatch* (1.1 in Figure 2) or *Objective mismatch* (1.2 in Figure 2). Furthermore, an *Objective mismatch* may be due to e.g., a *Predicate mismatch*

Organisational mismatch
1 Stakeholder-role mismatch
  1.1 Role-name mismatch
  1.2 Objective mismatch
    1.2.1 Objective-name mismatch
    1.2.2 Sub-objectives mismatch
    1.2.3 Predicate mismatch
      1.2.3.1 Missing-predicate mismatch
      1.2.3.2 Extra-predicate mismatch
      1.2.3.3 Predicate-ordering mismatch
2 Dependency mismatch
  2.1 Dependency-objective mismatch
  2.2 Dependency-roles mismatch

Figure 2. Example of a (partial) adaptation-mismatch taxonomy for the organisational layer.

(1.2.3 in Figure 2), caused by a *Missing-predicate mismatch* (1.2.3.1 in Figure 2) or by an *Extra-predicate mismatch* (1.2.3.2 in Figure 2) or by a *Predicate-ordering mismatch* (1.2.3.3 in Figure 2).

Figure 3 illustrates part of a possible taxonomy of adaptation mismatches for the behavioural layer of service-based applications. Part of this taxonomy is based on the behavioural mismatch patterns defined in [10], [9]. The taxonomy refers to mismatches that may occur when comparing a *required* behaviour specification with a *provided* one[1]. We have split (design-time) protocol mismatches based on whether the required and provided protocols are required to be *compatible* or *replaceable*. In the former case, the protocols have to complement each other – e.g., when one sends a message the other one has to receive it. In the latter case, the provided protocol has to include the required one, that is, the provided protocol has to behave as the required one with respect to the clients of the latter. For example, when checking protocol compatibility, if both protocols define the same set of message exchanges (viz., invoke and receive operations) yet the required protocol executes these activities in a sequence, while the provided protocol executes them in a loop, we then have a *Sequential vs. Iteration mismatch*[2] (1.1.1.1.3 in Figure 3). A *Split-invoke mismatch* (1.1.1.4.1 in Figure 3) occurs when the required protocol sends a message (i.e., one invoke operation) yet the provided protocol expects to receive the same information as part of several messages (i.e., several receive operations).

Finally, Figure 4 presents part of a possible taxonomy of adaptation mismatches for the service layer of a service-based application. The taxonomy refers to mismatches that may occur when comparing a *required* service specification with a *provided* one. An *Interface mismatch* (1.1 in Figure 4) can be classified into *Signature mismatch*

[1]In this context we refer to behaviour as containing protocol information (i.e., orchestration of messages) and information about partner links (i.e., roles and port types as defined by the BPEL specification). Unless otherwise specified, the mismatches refer to the required protocol.

[2]The former refers to the required protocol, while the latter refers to the provided protocol.

Behavioural-Layer mismatch
1 Design-time mismatch
  1.1 Protocol mismatch
    1.1.1 Required protocol not *compatible* with provided protocol
      1.1.1.1 Message-ordering mismatch
        1.1.1.1.1 Sequential vs. Sequential mismatch
        1.1.1.1.2 Sequential vs. Conditional mismatch
        1.1.1.1.3 Sequential vs. Iteration mismatch
        1.1.1.1.4 Conditional vs. Sequential mismatch
        1.1.1.1.5 ... (similar to the above)
      1.1.1.2 Extra-message mismatch
        1.1.1.2.1 Extra-invoke mismatch
        1.1.1.2.2 Extra-receive mismatch
      1.1.1.3 Missing-message mismatch
        1.1.1.3.1 Missing-invoke mismatch
        1.1.1.3.2 Missing-receive mismatch
      1.1.1.4 Split-message mismatch
        1.1.1.4.1 Split-invoke mismatch
        1.1.1.4.2 Split-receive mismatch
      1.1.1.5 Join-message mismatch
        1.1.1.5.1 Join-invoke mismatch
        1.1.1.5.2 Join-receive mismatch
      1.1.1.6 Invocation-type mismatch
        1.1.1.6.1 Synchronous-invoke mismatch
        1.1.1.6.2 Asynchronous-invoke mismatch
    1.1.2 Required protocol not *replaceable* by provided protocol
      1.1.2.1... (similar to the above)
  1.2 Partner-Link mismatch
    1.2.1 Role mismatch
      1.2.1.1 Missing-role mismatch
      1.2.1.2 Extra-role mismatch
      1.2.1.3 Port-type mismatch
2 Run-time mismatch
  2.1 Invocation mismatch (process view)
    2.1.1 Invocation-timeout mismatch
    2.1.2 Invocation-fault mismatch
      2.1.2.1 Input-value mismatch
      2.1.2.2 Output-value mismatch

Figure 3. Example of a (partial) adaptation-mismatch taxonomy for the behavioural layer.

(viz., the required and provided interfaces have operations that differ either syntactically – different operation names, number, order, or type of input and output parameters, or semantically – use different OWL concepts for their inputs and outputs) and *Parameter-constraint mismatch* (viz., the required service interface imposes constraints – such as value range – on the input or output parameters of one of its operations and they are different from what the provided interface defines). Similarly, an *Operation data-type mismatch* (1.1.1.1.2 in Figure 4) can be classified into *Syntactic-input mismatch* (viz., a service operation has an input data type of unexpected/unknown type) and *Syntactic-output mismatch* (viz., a service operation has an output parameter of unexpected/unknown data type).

Adaptation templates. Templates define the high-level behaviour of adaptation processes. They are exposed as executable BPEL processes that may encapsulate (complex) adaptation techniques. Adaptation-template developers are in charge of associating the templates they develop to adaptation mismatches based on the types of mismatches they can cope with. For example, an adaptation template based on the algorithm defined in [2] can be used to solve *Message-ordering mismatch*es and hence it should be

Service-layer mismatch
1 Design-time mismatch
  1.1 Interface mismatch
    1.1.1 Signature mismatch
      1.1.1.1 Syntactic mismatch
        1.1.1.1.1 Operation-name mismatch
        1.1.1.1.2 Operation data-type mismatch
          1.1.1.1.2.1 Syntactic-input mismatch
          1.1.1.1.2.2 Syntactic-output mismatch
        1.1.1.1.3 Parameter-number mismatch
      1.1.1.2 Semantic mismatch
        1.1.1.2.1 Semantic-input mismatch
          1.1.1.2.1.1 Input plug-in mismatch
          1.1.1.2.1.2 Input subsumes mismatch
          1.1.1.2.1.3 Input failed mismatch
        1.1.1.2.2 Semantic-output mismatch
          1.1.1.2.2.1 Output plug-in mismatch
          1.1.1.2.2.2 Output subsumes mismatch
          1.1.1.2.2.3 Output failed mismatch
    1.1.2 Parameter-constraint mismatch
    1.1.3 Binding mismatch
      1.1.3.1 Binding-style mismatch
      1.1.3.2 Transport-protocol mismatch
2 Run-time mismatch
  2.1 Invocation mismatch (service view)
    2.1.1 ... (similar to Figure 3)

Figure 4. Example of a (partial) adaptation-mismatch taxonomy for the service layer.

Algorithm 1 Updating the dependency graph when adding adaptation template A associated to adaptation mismatch I

```
add A to the nodes of the graph
// look for direct dependencies emerging from A
for all template A' in the graph s.t. A' exposes an operation that A may invoke do
    add (A, A') to the edges of the graph
end for
// look for indirect dependencies emerging from A
for all event e that A may raise do
    for all mismatch J s.t. match(e, J) ≠ "failed" do
        for all template A' associated to J do
            add (A, A') to the edges of the graph
        end for
    end for
end for
// look for direct dependencies targeting A
for all template A' in the graph s.t. A' may invoke an operation that A exposes do
    add (A', A) to the edges of the graph
end for
// look for indirect dependencies targeting A
for all template A' in the graph do
    for all event e that A' may raise s.t. match(e, I) ≠ "failed" do
        add (A', A) to the edges of the graph
    end for
end for
```

associated to the respective adaptation mismatch (1.1.1.1 in Figure 3). Furthermore, an adaptation template based on the algorithm defined in [9] can be employed to solve e.g., *Extra-message mismatch*es (1.1.1.2 in Figure 3) or *Missing-message mismatch*es (1.1.1.3 in Figure 3).

Cross-layer adaptation. Adaptation mismatches may require changes at various layers of a service-based application. For example, a *Stakeholder-role mismatch* event at the organisational application layer may require a the removal of an existing role and the addition of another role. This may also trigger changes at the behavioural and service layers. On the one hand, the behaviour may be adapted so as to take into account the new role and a new partner link. On the other hand, a new service may be needed to fulfil an organisational goal of the new role. Such complex adaptation scenarios that cross several application layers can be implemented by linking adaptation templates corresponding to layers where adaptation is needed. Templates may be linked either *directly* (step 5a in Figure 1), or *indirectly* (step 5b in Figure 1). In the former case, a BPEL adaptation template invokes the WSDL interface of another BPEL adaptation template. In the latter case, a BPEL adaptation template raises an event that will trigger the selection, deployment and execution of another adaptation template. This can simply be achieved using standard BPEL activities – *invoke* to generate events and *receive* or *pick* (*onMessage*) branches to receive events. A primary advantage of using this technique consists of the reusability of adaptation templates across BPEL engines.

On the one hand, direct linking of adaptation templates may be preferred when layers have very tight dependencies (e.g., when a behavioural adaptation template needs to make use of another adaptation template that (un)deploys a service) and the invoked adaptation templates are unlikely to change over time, or when linking simple adaptation templates into complex ones at the same layer. On the other hand, indirect linking of adaptation templates may be preferred when adaptation developers want to achieve full flexibility – e.g., when adaptation templates and adaptation-mismatch taxonomies are likely to change over time, or when triggering templates at different application layers.

Linking adaptation templates may lead to cyclical dependencies in the overall adaptation process. Such dependencies can be detected by searching for cycles in the directed graph of dependencies among adaptation templates. The nodes of the graph represent adaptation templates and directed edges represent dependencies. For example, when a template is added into a registry one may update the graph using algorithm 1. The algorithm first adds the new template into the graph. It then adds direct and indirect dependencies whose source is the new template. Finally, it adds direct and indirect dependencies that have the new template as target.

Having a flexible link between adaptation templates and adaptation mismatches organised into taxonomies brings the following benefits:

- *Flexible application logic*. Application logic may evolve over time. Developers may initially employ general adaptation templates to solve possible application issues. As they further develop their applications they may replace the adaptation logic with a new one just by simply replacing the event that triggers the required adaptation.
- *Flexible adaptation logic*. Similarly to applications, adaptation logic may evolve over time. For example,

when new versions of existing templates can cope with new adaptation issues, template developers can easily assign them to the new adaptation issues.

- *Efficiency*. Having a hierarchy of adaptation mismatches allows for more tailored adaptations to be performed (whenever available). For example, a *Transport-protocol mismatch* (1.1.3.2 in Figure 4) for a service-based application may be tackled more efficiently (viz., faster, with minimal intrusion and less undesired changes) through an adaptation template that just replaces the transport protocol (e.g., template associated to 1.1.3.2 in Figure 4), than through another adaptation template that looks for an alternative service and then replaces the entire service (e.g., template associated to 1.1 in Figure 4).
- *Increased application robustness.* Although one would ideally employ only tailored adaptation templates, robust applications may consider employing substitute adaptation templates (e.g., more general or specific adaptation templates) when no exact ones can be found.

B. Adaptation Steps

Figure 1 describes the conceptual architecture and the main steps of our approach for the adaptation of multi-layer applications.

Adaptation-triggering event. The adaptation process starts when a monitoring unit raises an adaptation event (step 1 in Figure 1). The event is to be defined using existing taxonomies of adaptation mismatches.

Matching adaptation templates. We employ a matching process to discover adaptation templates needed to cope with the raised issue (steps 2 and 3 in Figure 1). The process first matches the event to nodes in taxonomies representing classifications of adaptation mismatches. The matching process verifies the degree of match between the event and adaptation mismatches based on existing taxonomies at the layer at which the event was triggered.

Algorithm 2 presents the core of the matchmaking algorithm. The algorithm first tries to find an adaptation mismatch that *exact*ly matches the raised event. If such mismatch exists, the matching process nondeterministically selects one of the adaptation templates associated to this mismatch. If no adaptation templates are found for *exact* matches, the matching process inspects adaptation templates corresponding to *plug-in* matches (viz., ascendent mismatches). If no such templates are found, the process then looks for adaptation templates corresponding to *subsumes* matches (viz., descendent mismatches).

Performing the adaptation. In this paper we assume that the developer is in charge of configuring the template selected by the matching process during the previous step prior to its deployment. This is particularly important when the matching process returns either a *plug-in* or a *subsumes* match as well as when tackling adaptations due to multiple related events.

Algorithm 2 Matching event e to adaptation mismatches and corresponding adaptation templates

```
Let I be a mismatch s.t. match(e, I) = "exact"
A_I ← {(I, A) | A is a template associated to issue I}
if A_I ≠ ∅ then
    select (I, A) ∈ A_I
    return ((I, A), "exact")
end if
// if no "exact" template was found, then look for a "more general" template
J ← parent(I)
while J ≠ "root" do
    A_J ← {(J, A) | A is associated to J}
    if A_J ≠ ∅ then
        select (J, A) ∈ A_J
        return ((J, A), "plug − in")
    else
        J ← parent(J)
    end if
end while
// if no "more general" template was found then look for a "more specific" template
Js ← {J | I = parent(J)}
while Js ≠ ∅ do
    A_Js ← {(J, A) | A is associated to J ∈ Js}
    if A_Js ≠ ∅ then
        select (J, A) ∈ A_Js
        return ((J, A), "subsumes")
    else
        Js' ← {J | K = parent(J) and K ∈ Js}
        Js ← Js'
    end if
end while
// fail otherwise
return (nil, "failed")
```

As previously mentioned, we argue for adaptation templates to be defined and deployed as BPEL processes that perform the actual adaptation (steps 4, 5 and 6 in Figure 1). In order to produce the desired adaptation, templates input information about the application state and layer at which they operate (e.g., the BPEL process of a service-based application in case of an event triggered at the behavioural layer) and about the triggered event (e.g., a *Message-ordering mismatch* between a BPEL process of this application and one of its BPEL client processes). The adaptation is then performed through the execution of the deployed (composite) adaptation template(s). Steps 5a and 5b in the figure illustrate the possible cross-layer nature of adaptation templates, which may trigger adaptation at different application layers either directly through WSDL invocations, or through events.

III. Case Study

Social Messaging Application. The case study (see Figure 5) describes a "Social Network System" in which registered users (publishers) are allowed to send messages on specific topics to other users (subscribers) according to their registered preferences and observed mood or status. For example, Ann can send a message "meeting at pub tonight" under the topic "Entertainment" and Bob and Steven, subscribers on that topic, receive corresponding messages, the former to his computer via a Facebook message (status: at

home available) and the latter to his telephone via SMS (status: on road). In principle, the case study addresses the communication mediation among various parties, depending on their preferences and observed context. In this case, the communication mediation allows the dynamic reconfiguration of communication channels via adaptation, due to service problems such as the availability of services.

Application Layers. Our case study consists of three application layers: Organisation (OL), Behaviour (BL) and Service (SL). The layers are related and adaptations at one layer could be linked to another in any direction. The layers considered in this example are in accordance with the layers specified in the ALIVE project [18], aiming to promote high flexibility and availability of services across similar layers.

More specifically, the OL provides a formalisation of the application's requirements with *Roles* that software entities undertake, *Objectives* are goals that *Roles* have to fulfil and *Dependencies* mark the allowable interactions among *Roles*. The BL provides protocol details of how the *Roles* taken by Participants (Lanes) are orchestrated to fulfil certain *Objectives* by undertaking a sequence of intermediate activities. The actual orchestration takes the form of a business process such as BPMN [19] and corresponding BPEL that execute the defined behaviour. The SL presents the services that are available to our system with their providers.

In our case study, at OL we distinguish four roles; a Publisher, a SocialNetwork, a Subscriber and a FacebookProvider (see top part in Figure 5). The Publisher depends on the SocialNetwork to perform his/her main objective, that is, to send messages, the SocialNetwork depends on a FacebookProvider to send Facebook messages via a forward mechanism, and the Subscriber receives massages either directly from the SocialNetwork or the FacebookProvider via an appropriate medium and accordingly to his/her observed status. At BL we show how a publisher (e.g., Ann) initiates the process by sending a message under a specific topic (see middle part in Figure 5). Once the process receives the message an (internal) service will retrieve the topic subscribers and their preferences, it will then resolve the appropriate channels for each subscriber based on their preferences and current status and accordingly it will invoke a service to send a formatted message using the correct medium and preferences. At SL we outline the SocialNetwork and Facebook services used by our case study (see bottom part in Figure 5). The former is an internal service used by the process to resolve the subscribers on a topic and their preferences, as well as to send SMSMessages and VoiceMessages. The latter is an external service provided by the FacebookProvider and used by the process to forward Facebook messages to subscribers.

Adaptation Mismatch. We assume that during the execution of the case study (implemented as an executable BPEL process) the monitor identifies and generates an *Invocation-fault mismatch* event as the process cannot perform the

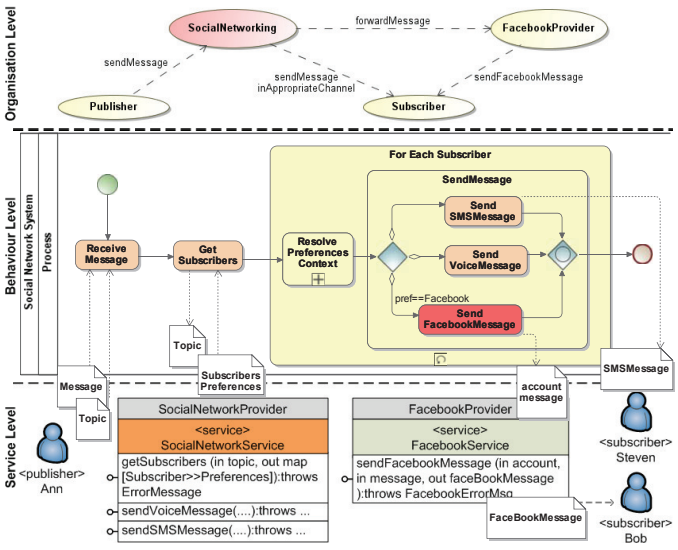

Figure 5. A "Social Network" case study across multi-layers.

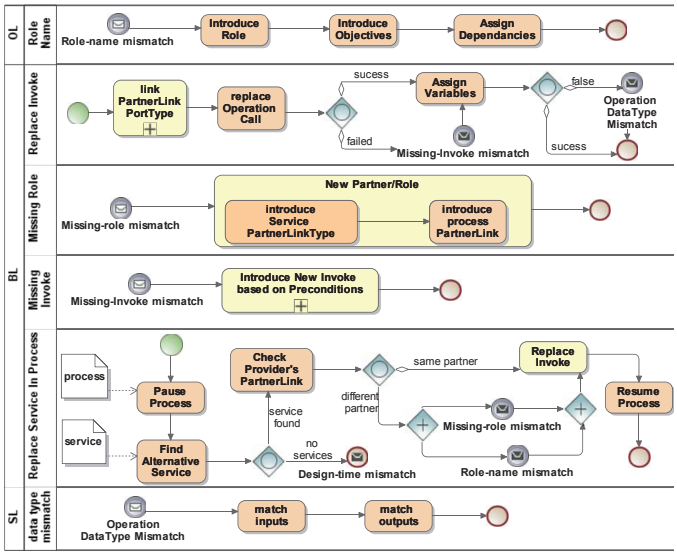

Figure 6. Examples of adaptation templates.

SendFacebookMessage activity via the invocation of the *FacebookService*. The mismatch may be due to service unavailability or because of changes to the service interface that break the communication protocol with the process. As a result, the matching process will match the event to an *Invocation-fault mismatch* mismatch, (2.1.2 in Figure 3, BL). For this scenario we assume that there are no adaptation templates associated with this mismatch. Thus, a plug-in match follows up that will match the event to a more general mismatch, an *Invocation mismatch*, (2.1 in Figure 3). In this case, the mismatch is linked to a *ReplaceServiceInProcess* BL template that will replace the previously failed invocation with an alternative one, as illustrated in Figure 6.

Adaptation. Within each layer we assume the availability of several adaptation templates, some of which are linked, and which are associated with different taxonomy mismatches (see Figure 6).

More specifically, within the BL, the *ReplaceServiceInProcess* template will initially pause the executing process and will continue with the *FindAlternativeService* activity that would replace the failed service via a matchmaking process (based on *exact*, *subsumes* and *plug-in* matches). In our example, the matched service is the FacebookThriftService, that is based on a new communication protocol (Thrift[3]). In addition, the new service is designed to send Facebook messages to groups instead of individual subscribers. So, there is a requirement of registering the interested individual to groups created per topic before invoking the new service. Next, if a service is not found, a BL *Design-time mismatch* event is generated, so an alternative and more radical action could be taken. Alternatively if a service is found, we check whether the provider of the service is an existing one, in which case we invoke the *Replace Invoke* template at BL, otherwise we generate both *Missing-role mismatch* and *Role-name mismatch* events (see Figure 6). In our case we assume that the service is exposed by a new provider, the FacebookProvider2.

Following, the *Missing-role mismatch* event is matched against a *Misssing-role mismatch* mismatch (1.2.1.1 in Figure 3) that will invoke the *Missing Role* template at BL. As a result, the template will introduce new partner-link types and partner links at service and process descriptions. In addition, the *Role-name mismatch* event is matched against a *Role-name mismatch* mismatch (1.1 in Figure 2) that will invoke the *Role Name* template at OL. The template will introduce a new role (corresponding to the service provider), new objectives (corresponding to new required invocation calls) and dependencies (links among roles). In our example, a new *Group* role will be introduced with dependencies from the Subscriber which needs to subscribe to the group and the new FacebookProvider in order to receive group FacebookMessages. Once the *Missing Role* and *Role Name* templates terminate they get synchronised and invoke the *Replace Invoke* template at BL.

The *Replace Invoke* template will replace the actual service invocation. It will start by linking the partner link and port type properties with the new ones, and it will continue with the replacement of the actual operation call. This activity can lead either to a *Missing-invoke mismatch* event, if was unsuccessful, or to the assignment of input and output variables, if successful. Similarly, the assignment of variables can generate an *Operation Data-type mismatch* event, if was unsuccessful. In our example, the *Missing-invoke mismatch* event will lead to the introduction of a new activity *Create Group for Topic* due to precondition

[3]http://incubator.apache.org/thrift/

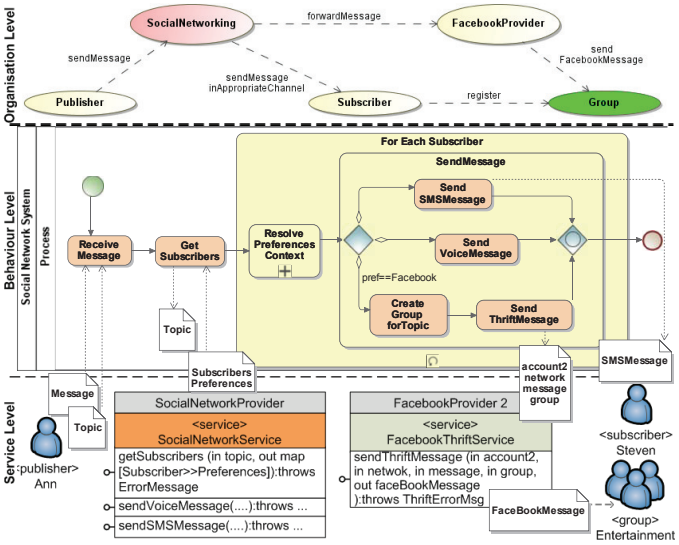

Figure 7. Adapting the case study across layers due to a service invocation failure.

requirements in which messages are send to groups, and the replacement of the previous *Send Facebook Message* activity with the *Send Thrift Message* activity based on domain matchmaking. In addition, the *Operation Data-type mismatch* event will lead to an adaptation template that will provide the appropriate inputs for the call. Once *Replace Invoke* finishes, the process is resumed and ready for the invocation of the new service. Figure 7 presents the result of the adaptation process across the layers.

In summary, the adaptation framework eases the job of the Social Network System developer (and of multi-layer application developers in general) mainly by supporting the decoupling of the cross-layer adaptation logic from the core application logic. An immediate advantage is that applications become more robust and easier to adapt. Furthermore, cross-layer adaptations can be orchestrated from simpler layer-specific adaptations, which can be updated in isolation. In real-life scenarios, applications, taxonomies of mismatches and adaptation templates evolve over time. On the one hand, application developers can focus just on the core application and suitably trigger adaptation templates that are needed to solve possible mismatches. New application versions may introduce new mismatches that can be solved, for example, by triggering additional or more general adaptation templates. The adaptation of new application versions that add new layers can be achieved by triggering adaptation templates that adapt the respective layers. Furthermore, developers may initially employ general adaptation templates to solve possible application mismatches. As they further develop their applications they may replace the generic adaptation logic with more specific ones by simply replacing the event that triggers the required adaptation. On the other hand, adaptation developers may update templates either

to better solve existing mismatches, or to solve new ones. These new mismatches may require the refinement of or the definition of new taxonomies of adaptation mismatches. The adaptation framework also supports the definition of more robust applications. Although one would ideally employ only tailored adaptation templates, the adaptation framework considers substitute adaptation templates (e.g., more general or specific adaptation templates) when no exact ones can be found.

IV. Related Work

In this section we focus on related adaptation approaches. We point the interested reader to [5], [8], [15], [16] for examples of monitoring of application layers.

A substantial amount of research has been done on adaptation of service-based applications, either at the service layer, or at the behaviour layer. Erradi et al. [5] develop a middleware to improve the dependability of service-based applications by introducing adaptation frameworks that support dynamic Web service selection and composition. Dedicated framework services continuously monitor interactions with the participating services to verify that the configured monitoring policies are being satisfied. Similarly to our approach, whenever an undesired condition is detected, the monitoring service generates a violation event with a fault type to trigger adaptation. However, their adaptation approach is mainly targeted at the service-interface layer. Heuvel et al. [20] propose a configurable adapter architecture to implement self-adaptive Web services. The key construct in this architecture is the generic protocol adapter that defines a mapping between businesses or conversation protocols between service providers and consumers. At runtime, the service manager can choose or compose existing mappings to adapt interacting services. Although this architecture is extensible, it is not clear how the self-adaptation of Web services can be triggered and woven to the running business process instance. Furthermore, the generic adapter is limited to solve Web service protocol mismatch. Canal et al. [21] define a model-based adaptation approach focusing on mismatches appearing at the behaviour level. Taking as input the behavioural interfaces of the components to be adapted and an adaptation contract that describes the constraints for the interacting components, the approach generates adapter protocols using either synchronous products or Petri net encodings. However, the adaptation contracts in their work must be manually designed, which is not trivial for complex service protocols. To solve this, Martin et al. [22] define an approach for the automatic generation of adaptation contracts which can be used to overcome signature and behavioural mismatches.

Various recent approaches propose the use of Aspect-Oriented Programming (AOP) to implement the required adaptation. Kongdenfha et al. [9] propose a framework using AOP for service adaptation due to interface and protocol mismatches. In the proposal, for each type of mismatch between an external service specification and implementation, a template containing a set of <pointcut, advice> pairs to describe the service adaptation approach is provided. However, the approach requires developers to manually define the mismatch before performing the adaptation. Charfi et al. [23] define a plug-in architecture for self-adaptive Web service compositions by modularising self-adaptation features in aspect-based plug-ins. These plug-ins are implemented as two types: monitoring aspects and adaptation aspects. The monitoring aspects collect information and decide whether to activate the adaptation aspects. Both monitoring and adaptation aspects can be hot-deployed to BPEL engines that support the aspect-oriented workflow language AO4BPEL. In their approach, each monitoring aspect is associated to certain adaptation plans. Karastoyanova et al. [24] illustrate how the AOP paradigm can be mapped and applied in the BPEL language to enable the adaptation of running orchestrations. In the infrastructure, a BPEL engine publishes business-process status notification events to a broker component which (de)attaches WS-Policy attachments similar to aspect weaving. However, the authors do not discuss how to generate appropriate WS-Policy attachments (aspects) for the desired service adaptation.

With respect to the above adaptation approaches, our proposal features a cross-layer adaptation framework that is not bound to (single) predefined layers. Our focus is on the dynamic and flexible discovery of composite adaptation templates that solve cross-layer adaptation dependencies.

Recent adaptation approaches tackle the adaptation of multi-layer applications. Gjørven et al. [7] propose a technology-agnostic adaptation middleware that can be used to integrate and exploit adaptation techniques and mechanisms from both application and service layers. The proposed middleware focuses on providing a framework that can integrate various adaptation techniques in different layers in order to control them in one place. Kazhamiakin et al. [8] propose a cross-layer framework and underlying conceptual model to address the monitoring and adaptation problem in service based application. The authors provide a set of requirements for cross-layer monitoring and adaptation frameworks of service-based applications and illustrate a uniform conceptual model underlying such frameworks.

With respect to the above cross-layer adaptation approaches, our proposal enhances the flexibility of the adaptation process. Our flexibility comes from combining taxonomies of adaptation mismatches and an event-based mechanism to drive the discovery of composite adaptation templates.

In summary, note however that existing adaptation techniques, such as the ones presented above, can be plugged into our adaptation framework as adaptation templates. For example, one may define an adaptation template based on the algorithm described in [5] to solve *Syntactic-input*

*mismatch*es and *Syntactic-output mismatch*es at the service layer, and another one that use the work defined in [9] to solve *Extra-message mismatch*es at the behavioural layer.

V. Concluding Remarks

In this paper we have outlined a methodology for the dynamic and flexible adaptation of multi-layer applications, whose main ingredients are events, taxonomies of adaptation mismatches and adaptation templates.

In a nutshell, our methodology employs a loosely-coupled event-based architecture in which events trigger the process of matching templates, which encapsulate adaptation logic into BPEL processes that perform the required adaptation. The templates may be easily reused and linked so as to tackle cross-layer adaptation issues. The matching process employs taxonomies of adaptation mismatches to dynamically select adaptation templates based on the degree of match between the OWL concepts associated to events and adaptation mismatches. The use of taxonomies of adaptation mismatches allows for a flexible matching of more general or more specific adaptation templates that may solve a given mismatch. Such templates correspond to *plug-in* (more general) and *subsumes* (more specific) mismatch matches. We provide support for cross-layer adaptation by allowing templates to be composed either directly, through invocations of WSDL operations, or indirectly, through events. Developers are in charge of ensuring that (selected) matching template(s) will actually perform the desired adaptation. If the developer has access to the behaviour of the templates then they may be customised prior to their execution – for instance when triggered templates cannot adapt all affected application layers, or when they attempt to adapt layers that the application does not define. In the former case, the developer can change the behaviour of templates so as to trigger or directly invoke other needed templates. In the latter case the developer can inhibit part of the templates' behaviour.

In this paper, we described how a mismatch event can trigger a systematic cross-layer adaptation process. It is worth noting that in certain situations the execution of complex multi-layer applications may generate more than one mismatch at (almost) the same time. While multiple mismatches can be addressed one at a time by our methodology, a meta-level coordinated selection of the sequence of templates to be applied may in some cases yield globally better adaptations. This is precisely one of the directions for our future work.

We also intend to formalise and enhance the matching process with a template selection algorithm based on rankings of templates. The ranking should measure the suitability of a sequence of adaptation templates by taking into account the number of adapted application layers, the number of raised events and applied templates, as well as the type of matching (*exact*, *plug-in*, *subsumes*) with which templates were selected.

Last but not least, we plan to deploy the conceptual architecture defined in this paper as an adaptation framework that allows third-party developers to easily define taxonomies of adaptation mismatches and integrate adaptation logic into templates.

Acknowledgement. This work has been supported by the EU-funded ALIVE project [18] and by Science Foundation Ireland grant 03/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

References

[1] M. Dumas, M. Spork, and K. Wang, "Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation," in *BPM*, ser. LNCS, S. Dustdar, J. L. Fiadeiro, and A. P. Sheth, Eds., vol. 4102, 2006, pp. 65–80.

[2] A. Brogi and R. Popescu, "Automated Generation of BPEL Adapters." in *ICSOC'06*, ser. LNCS, A. Dan and W. Lamersdorf, Eds., vol. 4294, 2006, pp. 27–39.

[3] J. Harney and P. Doshi, "Speeding up Adaptation of Web Service Compositions Using Expiration Times," in *WWW '07*. ACM, 2007, pp. 1023–1032.

[4] N. C. Narendra, K. Ponnalagu, J. Krishnamurthy, and R. Ramkumar, "Run-Time Adaptation of Non-functional Properties of Composite Web Services Using Aspect-Oriented Programming," in *ICSOC'07*, ser. LNCS, B. J. Krämer, K.-J. Lin, and P. Narasimhan, Eds., vol. 4749, 2007, pp. 546–557.

[5] A. Erradi, P. Maheshwari, and V. Tosic, "Policy-driven middleware for self-adaptation of web services compositions," in *Proceeding of 7th International Middleware Conference*, vol. 4290 of LNCS, Nov 2006, pp. 62–80.

[6] S-Cube, "PO-JRA-1.2.1: State of the Art Report, Gap Analysis of Knowledge on Principles, Techniques and Methodologies for Monitoring and Adaptation of SBAs," 2008, http://www.s-cube-network.eu.

[7] E. Gjørven, R. Rouvoy, and F. Eliassen, "Cross-layer self-adaptation of service-oriented architectures," in *MW4SOC'08*. ACM, 2008, pp. 37–42.

[8] R. Kazhamiakin, M. Pistore, and A. Zengin, "Cross-layer adaptation and monitoring of service-based applications," in *Proceedings of the 2nd Workshop on Monitoring, Adaptation and Beyond (MONA+), LNCS*, 2009.

[9] W. Kongdenfha, H. Motahari-Nezhad, B. Benatallah, F. Casati, and R. Saint-Paul, "Mismatch patterns and adaptation aspects: A foundation for rapid development of web service adapters," *IEEE TSC*, vol. 2, no. 2, pp. 94–107, April-June 2009.

[10] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli, "Towards an engineering approach to component adaptation," in *Architecting Systems with Trustworthy Components*, ser. LNCS, R. H. Reussner, J. A. Stafford, and C. A. Szyperski, Eds., vol. 3938, 2004, pp. 193–215.

[11] A. Alves et. al. (Eds), "Web Services Business Process Execution Language Version 2.0," April 2007, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

[12] M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara, "Semantic matching of web services capabilities," in *ISWC'02*, ser. LNCS, I. Horrocks and J. A. Hendler, Eds., vol. 2342, 2002, pp. 333–347.

[13] D. McGuiness and F. van Harmelen (Eds), "OWL Web Ontology Language Overview," Web guide, February 2004, http://www.w3.org/TR/owl-features.

[14] E. Christensen et. al. (Eds), "Web Services Description Language (WSDL) 1.1," March 2001, http://www.w3.org/TR/wsdl.

[15] O. Ezenwoye and S. M. Sadjadi, "Trap/bpel: A framework for dynamic adaptation of composite services," in *In Proceedings of the International Conference on Web Information Systems and Technologies (WEBIST 2007)*, 2007.

[16] O. Moser, F. Rosenberg, and S. Dustdar, "Non-intrusive monitoring and service adaptation for ws-bpel," in *WWW '08: Proceeding of the 17th international conference on World Wide Web*. New York, NY, USA: ACM, 2008, pp. 815–824.

[17] R. Popescu, A. Staikopoulos, and S. Clarke, "An extensible monitoring and adaptation framework," in *Proceedings of the 2nd Workshop on Monitoring, Adaptation and Beyond (MONA+), LNCS*, 2009, to appear. http://www.scss.tcd.ie/~rpopescu/MONA+.pdf.

[18] ALIVE, "Coordination, Organisation and Model Driven Approaches for Dynamic, Flexible, Robust Software and Services Engineering," 2010, European Commission 7th Framework Programme, ICT, http://www.ist-alive.eu/. [Online]. Available: http://www.ist-alive.eu/

[19] OMG, "Business Process Model and Notation (BPMN) version 2," OMG, Aug 2009, http://www.omg.org/spec/BPMN/2.0/.

[20] van den Heuvel, Willem-Jan, H. Weigand, and M. Hiel, "Configurable adapters: the substrate of self-adaptive web services," in *ICEC'07*, 2007, pp. 127–134.

[21] C. Canal, P. Poizat, and G. Salaun, "Model-based adaptation of behavioral mismatching components," *IEEE TSE*, vol. 34, no. 4, pp. 546–563, July-Aug 2008.

[22] J. A. Martin and E. Pimentel, "Automatic generation of adaptation contracts." *ENTCS*, vol. 229, no. 2, pp. 115–131, 2009. [Online]. Available: http://dblp.uni-trier.de/db/journals/entcs/entcs229.html#MartinP09a

[23] A. Charfi, T. Dinkelaker, and M. Mezini, "A plug-in architecture for self-adaptive web service compositions," in *ICWS'09*, Jul 2009, pp. 35–42.

[24] D. Karastoyanova and F. Leymann, "Bpel'n'aspects: Adapting service orchestration logic," in *ICWS'09*, Jul 2009, pp. 222–229.