SCIENCE CHINA
Information Sciences

Model-based cross-layer monitoring and adaptation of multilayer systems

Hui SONG, Amit RAJ, Saeed HAJEBI, Aidan CLARKE and Siobhn CLARKE

Citation: SCIENCE CHINA Information Sciences **56**, 082101(15) (2013); doi: 10.1007/s11432-013-4915-5

View online: http://engine.scichina.com/doi/10.1007/s11432-013-4915-5

View Table of Contents: http://engine.scichina.com/publisher/scp/journal/SCIS/56/8

Published by the Science China Press

SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •
Special Focus

August 2013, Vol. 56 082101:1–082101:15
doi: 10.1007/s11432-013-4915-5

Model-based cross-layer monitoring and adaptation of multilayer systems

SONG Hui[1*], RAJ Amit[1], HAJEBI Saeed[1], CLARKE Aidan[2] & CLARKE Siobhán[1]

[1]*Lero: The Irish Software Engineering Research Centre, School of Computer Science and Statistics, Trinity College Dublin, Dublin 2, Ireland;*
[2]*IBM Software Ireland Laboratory, Dublin, Ireland*

Received May 4, 2013; accepted June 5, 2013

Abstract With the vision of "Internet as a computer", complex software-intensive systems running on the Internet, or the "Internetwares", can be also divided into multiple layers. Each layer has a different focus, implementation technique, and stakeholders. Monitoring and adaptation of such multilayer systems are challenging, because the mismatches and adaptations are interrelated across the layers. This interrelation makes it difficult to find out: 1) When a system change causes mismatches in one layer, how to identify all the cascaded mismatches on the other layers? 2) When an adaptation is performed at one layer, how to find out all the complementary adaptations required in other layers? This paper presents a model-based approach towards cross-layer monitoring and adaptation of multilayer systems. We provide standard meta-modelling languages for system experts to specify the concepts and constraints separately for each layer, as well as the relations among the concepts from different layers. Within each individual layer, we use run-time models to represent the system state specific to this layer, monitor the systems by evaluating model changes according to specified constraints, and support manual or semi-automated adaption by modifying the models. When a change happens in the run-time model for one layer, either caused by system changes or by the adaptation, we synchronize the models for other layers to identify cascaded mismatches and complementary adaptations across the layers. We illustrate the approach on a simulated crisis management system, and are using it on a number of ongoing projects.

Keywords multilayer systems, monitoring, dynamic adaptation, model driven engineering, bidirectional model transformation

Citation Song H, Raj A, Hajebi S, et al. Model-based cross-layer monitoring and adaptation of multilayer systems. Sci China Inf Sci, 2013, 56: 082101(15), doi: 10.1007/s11432-013-4915-5

1 Introduction

Internet is becoming a huge and unique platform to host a wide range of software-based systems. Due to the open and dynamic characteristics of Internet, software systems running on Internet also have a range of unique features, such as autonomous, cooperative, etc., and are also called *Internetwares* [1]. With the vision of *Internet as a computer*, the large scale and complex Internetwares can be also divided into multiple layers. For example, a typical service-based Internetware system is often considered to be constituted of the business layer, the service layer, and the infrastructure layer [2]. Multilayer systems

*Corresponding author (email: hui.song@scss.tcd.ie)

have a good separation of concerns: Different layers cater to different stakeholders, are implemented by different technologies, and involve different bases of knowledge.

This multilayer style does not only influence the development of these systems, but also shapes the adaptation of them at runtime: Ideally, the adaptation should also be isolated within each of the layers. On the one hand, the separation nature prevents one layer from getting the details from other layers, and makes the analysis and planning focused on individual layers. On the other hand, the existing monitoring or change management tools are usually based on particular technologies, and the system administrators are also based on particular technical domains. This makes the people more willing and experienced on handling the problems in their *own* layers.

However, despite the separation of concerns during development time, the run-time monitoring and adaptation on different layers are still interrelated with each other. In particular, a mismatch (a situation which is not in accordance with the desired one) happened in one layer may influence other layers, and an adaptation on one layer may require complementary adaptions on the other layers. This causes two questions: 1) When a mismatch is captured from one layer, how to find out the related mismatches from other layers before they would have actually showed their impact? 2) When an adaption is performed on one layer, how to identify all the complementary adaptions on other layers before executing them?

In order to deal with the overlap, many approaches on the monitoring and adaptation of multilayer systems [3–6] choose to regard the layers as a whole again, by defining the relations between the mismatches and adaptation solutions from different layers, and employ a centralized mechanism to handle them. However, these approaches violate a basic principle of multilayer systems, i.e., the separation of concerns between layers. In particular, following centralized approaches, the one who performs adaptations or defines adaptation templates has to consider mismatches and adaptations from all layers, as well as the complex relations between them. Therefore, he or she is required to have the expertise on all the different techniques, and to make decisions on behalf of different stakeholders. Moreover, the technology binding between layers can be flexible, and the mismatches and adaptations on different technologies are also different. This makes it impossible to enumerate all the possible adaptations and their relations in advance. In summary, a centralized approach towards cross layer adaptation is not a good way as the "software engineering of system adaption" [7].

In the development step of software systems, Model-Driven Engineering (MDE) [8] is one of the promising approaches to coping with the correlation between different development steps. The information about the system in different steps is captured by different models. Developers focusing on a particular step only work on the model of that step, and their effect on this particular model is automatically propagated to the other steps via model transformation. For example, in the Model-Driven Architecture approach, designers work on the platform-independent model (PIM) without caring about the platform details, and their design decisions will be propagated and embedded in the platform-specific model (PSM) via the model transformation from PIM to PSM.

In this paper, we present a model-based approach towards cross-layer system monitoring and adaptation in a decentralized manner. For a running system, its run-time status in each layer is captured by a run-time model [9], which is aligned to the concerns and techniques in that layer. Monitoring and adaptation are performed within a layer based on its model, and their effects on the other layers will be automatically propagated via transformation between models. At design time, the mismatch and solution specifications are defined on the layer-specific models, and it is not required to enumerate all the potential relations between mismatches and adaptation solutions from different layers in advance. At runtime, the adaptation agents work separately in their own layers. Using their own models, they can see the influence of mismatches or modifications happening on the other layers, propagate their adaptation results to the other layers to ask for complementary adaptations, and check the effect of their adaptation through the feedback from other layers. It is worth noting that this work is focused on the adaptation assistance, and thus we assume that in each layer, there is an external adaptation agent which plans the proper adaptation based on the mismatches. How these agents work is out of the scope of this paper.

The challenge of this decentralized approach is that the different layers of a system are changing and being modified simultaneously, and in each layer's model, the information particular to this layer and

the information influenced by other ones are mixed together. This requires more sophisticated model synchronization solution, rather than the simple, unidirectional, and once-for-all model transformation.

To meet these challenges, our contributions can be summarized as follows.

1) We propose a stack of modeling languages for multilayer systems and the cross layer adaptation on them, based on OMG's MOF meta-modeling standards.

2) We propose a run-time model- and OCL evaluation-based approach to assisting the monitoring and adaptation within individual layers.

3) We design the algorithms to integrate the monitoring and adaptation actions from individual layers, based on the change propagation between models, in order to detect cascaded mismatches and complementary adaptations across layers.

We illustrate the approach on a simulated three-layered, service-based Crisis Management System, as well as a double-layered smart office system.

The rest of the paper is organized as follows. We introduce the approach and a motivating example in Section 2. We present the design and runtime aspects in Sections 3 to 5, and evaluate the approach on the motivating example, as well as other ongoing projects in Section 6. Section 7 discusses the related approaches and Section 8 concludes the paper.

2 Approach overview

2.1 Motivating example

We take a crisis-management system (CMS) [6] as a sample multilayer Internetware system throughout this paper. When a flood incident is reported, an emergency centre performs rescue operations by organizing other departments to work together. Figure 1 illustrate the three layers of this CMS: 1) In the Business Process Management Layer (BL), the simplified workflow of the emergency centre is constituted by three activities, i.e., get location, launch rescue, and file the incident record. The rescue activity is delegated to medical service which sends ambulance to the indicated location. The army service with helicopter is a backup. 2) In the Service Layer (SL), the activities and processes are implemented or defined as services. One service could be registered to another one so that the latter could utilize the former's functions. 3) In the infrastructure layer (IL), the services are hosted by different server nodes.

The mismatches and adaptations usually happen in a particular layer, but may influence the other layers. For example, the crash of the Tomcat1 server (a mismatch in the IL), causing the GetGPS and MedicalService not available (SL), which eventually results in the brokerage of the workflow in the emergency centre (BL). For another example, if the governor from the business layer observes that the flood has damaged the roads, and thus the medical service's ambulance is of no use, then he/she adapts the business process to delegate the Rescue activity to the Army service (BL). This adaptation alone is not enough: We first need to register the ArmyService to the EmergencyCentre (SL). After that, since the Army service requires GSNLocation as input, which does not match the output of GetGPS service, an adapter (say GPStoGSN) is required between GetGPS and Rescue (BL) . It then requires the implementation of the corresponding service (SL), and the deployment of it to the node of Tomcat2 (IL). If this node is overloaded, the infrastructure administrator will have to migrate the service to the Tomcat1.

This case study resulted in a list of mismatches across several layers of a large-scale multi-layered system. It presents the evidence that mismatches in a layer are inter-related with the mismatches of other layers. In order to sufficiently adapt a system for an un-desirable environmental conditions, it is required to find out all the mismatches in all the layers and execute corresponding adaptations.

In this paper, we choose a decentralized way to the adaptation of such an system: Users perform or specify the adaptation in the separate layers, and our approach automatically derives the global adaptation from the separate ones.

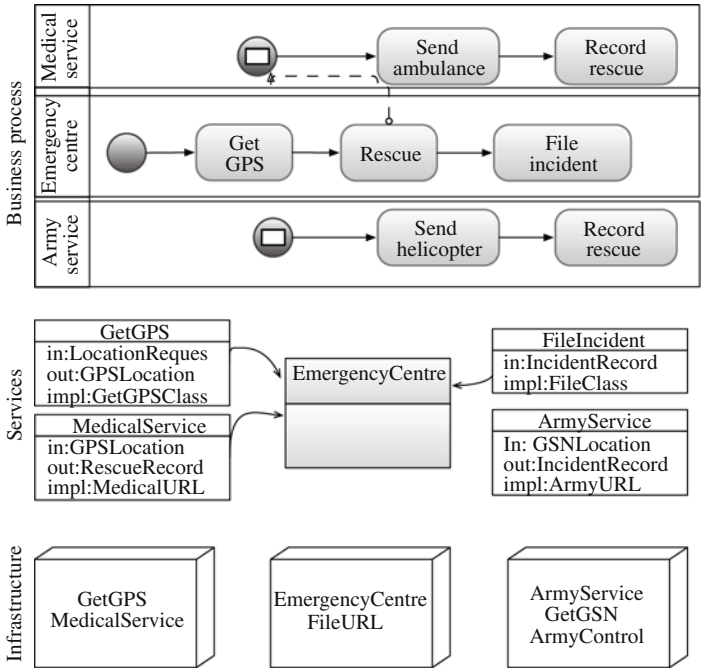

Figure 1 Part of the crisis management system (CMS).

2.2 Problem formalization

A running system's state is changing at runtime. We notate its state at a certain time point as $s \in S$, constituted by the system structure, status, configuration, and context, etc. The system changes, notated as $\Delta \subseteq S \times S$, happen at certain time points, and a change $\delta = (s, s')$ indicates that the system state changed from s to s'. Mismatch detection $\mathrm{dtct} : \Delta \to \wp(\mathfrak{Msm})$ evaluates the current system change δ and returns a subset of the mismatch names Msm describing the problem of the current system, notated as $\mathrm{dtct}(\delta) = \{\mathrm{msm}_1, \ldots, \mathrm{msm}_n\}$. From a set of mismatch and the current system state s, a system modification $\mathrm{mdf} : \wp(\mathrm{Msm}) \times S \to S$ leads the system to a new state s', either via automated planning or the operation of administrator. We name such state transfer forced by external inference as the system adaptation, notated by $\upsilon \in \Upsilon = S \times S$.

For a multilayer system with two layers m_i and m_j, the system state is described by the two states from the two layers, i.e., $S = S_i \times S_j$. The adaptation functions and their results are also divided into two parts. Ideally, if the system is perfectly isolated into two layers, the mismatch and modification are also isolated to two layers. That means one system change will only cause mismatch on a single layer, i.e., $\forall s = (s_i, s_j)[\neg(\mathrm{dtct}_i(s_i) \neq \emptyset \wedge \mathrm{dtct}_j(s_j) \neq \emptyset)]$. Similarly, modification only causes update on a single layer. However, due to the correlation between different layers, the adaptation is interrelated. We define three categories of cross-layer adaptation problems as follows.

1) *Mismatch spread*: $\exists \delta = ((s_i, s_i'), (s_j, s_j'))[\mathrm{dtct}(\delta) \cap \mathfrak{Msm}_j \neq \emptyset]$. That means monitoring on one layers should detect the mismatches related to other layers.

2) *Modification complement*: $(\exists s = (s_i, s_j), \mathrm{Msm} \subseteq \mathfrak{Msm}_i)[\mathrm{mdf}(s, \mathrm{Msm}) = (s_i', s_j') \to s_j \neq s_j']$. That means handling the mismatch on one layer may require updates on the other layers.

3) *Collaborative judgement*: $\exists \delta = (\delta_i, \delta_j)[\mathrm{dtct}_i(\delta_i) \neq \mathrm{dtct}(s) \cap \mathfrak{Msm}_i]$. That means mismatches detected within one layers are not the same as the mismatches in this layer detected by global monitoring. Similarly, $(\exists s = (s_i, s_j), \mathrm{Msm} \subseteq \mathfrak{Msm}_i)[\mathrm{mdf}(s) = (s_i', s_j') \wedge \mathrm{mdf}_i(\mathrm{Msm}, s_i)! = s_i']$. That means the result of modification within one layer is not the same as the result on this layer via global modification.

Regarding these problems, a direct solution is to do adaptation from a global perspective. To do this, we should either specify the mismatch or solutions on the concepts from different layers, or specify them on separate layers, but in the same time explicitly provide the relations between them across the

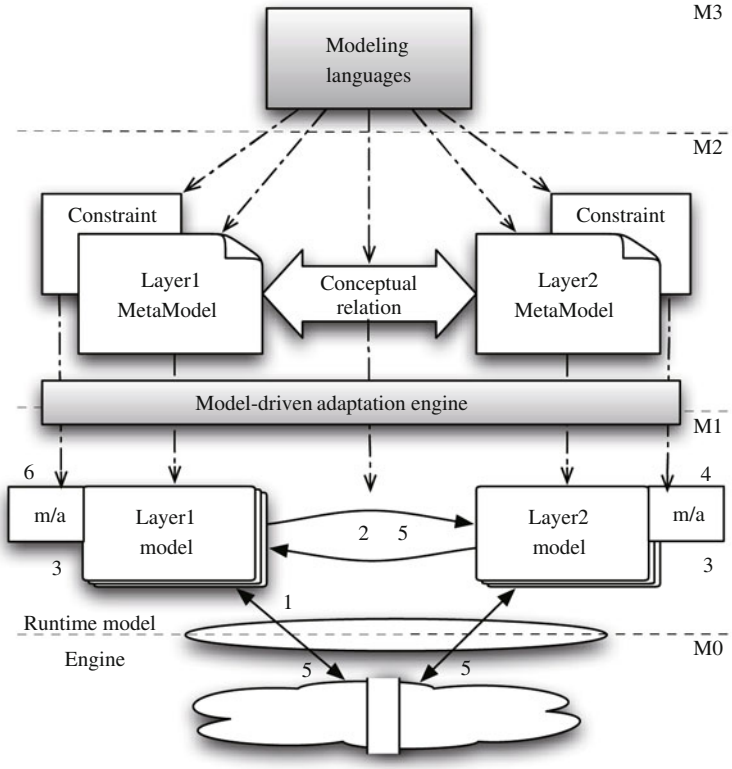

Figure 2 Approach architecture.

layers. However, as we have argued in Section 1, this requires strong expertise from the users who perform adaptation or provide the adaptation specifications. Alternatively, in this paper, we choose a decentralized way: Users perform or specify the adaptation in the separate layers, and our approach automatically derive the global adaptation from the separate ones. In other words, we allow different users to work on the function pairs of $(\text{dtct}_i, \text{mdf}_i)$ and $(\text{dtct}_j, \text{mdf}_j)$ separately, and automatically derive the correct global function pair $(\text{dtct}, \text{mdf})$.

2.3 The approach architecture

We provide an model-based approach to cross-layer system monitoring and adaptation. At design time, a stack of modeling languages is proposed for system experts to define the concepts of each layer and the relation between them across layers, and also to define the mismatches and possible solutions on individual layers. At runtime, a corresponding engine captures the system changes, identifies the mismatches on all the layers caused by the changes, and predicts the complementary adaptations on different layers when an administrator performs an adaptation on a particular layer. Figure 2 illustrates the approach architecture, according to the four-level meta-modeling architecture defined by OMG[1)].

In M3, or meta meta level, we provide the meta-modeling languages, which are used in the M2, or meta level, by system experts to define the system and its layers. Specifically, the system experts define the concepts in a layer as a meta-model, and define the mismatches and their solutions as constraints on the meta-model. Between the layers, the experts use the meta-model relations to define the relations between the concepts from different layers. These three specifications are defined using the OMG standard languages, MOF, OCL, and QVT-Relational, respectively[2)]. We describe how to use these languages, and their semantics on adaptation in Section 3.

1) http://www.omg.org/mof/.
2) http://www.omg.org/spec/index.htm.

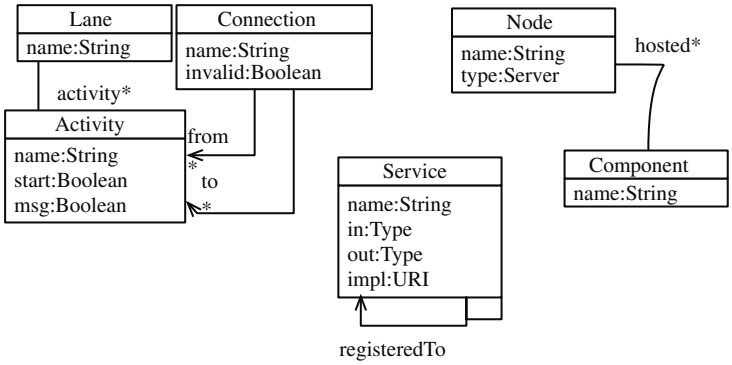

Figure 3 Simplified meta-models.

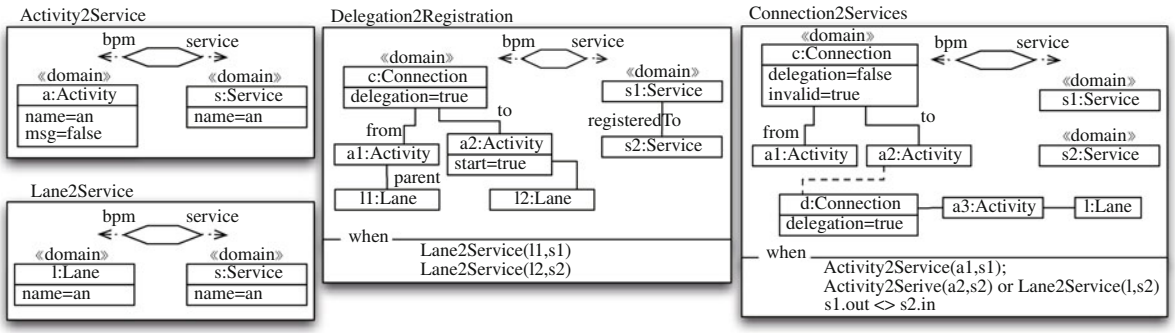

Figure 4 Sample QVT relations.

M1 and M0 shows how the approach works at runtime. A typical process is as follows: A system change on layer1 is captured by the runtime model (marked as step 1). The engine synchronizes the two models using bidirectional model transformation to propagate the changes (step 2), and then evaluate the constraints on both models and return the detected mismatches on each layer to the corresponding administrators (step 3). If an adaptation is applied on layer2 (step 4), the engine propagates the modification to layer1 (step 5), check mismatches and suggest complementary adaptations on layer1 (step 6). When all the mismatches are resolved, the final modifications on both runtime models are executed to the system. In Section 5, we present how the evaluation and bidirectional transformation approaches work, and how we use them together as an integrated adaptation process.

3 System modeling

• **Layer meta-models.** A meta-model defines the system concepts for a particular layer, the properties of each concept, and the association between the concepts inside the layer. The meta-model is specific to the technique and knowledge base in the layer. Figure 3 shows the simplified meta-models for our running example. Notice that a typical system with well-accepted layers does not require its meta-models to be defined from scratch, but using the meta-models according to the existing languages or APIs in the layers [10]. For example, the meta-model in the business layer is simplified from BPMN.

• **Relations.** For two layers' meta-models M_i and M_j, there exists a relation $R_{ij} \subseteq M_i \times M_j$. If the two layer models $m_i \in M$ and $m_j \in M_j$ satisfies $(m_i, m_j) \in R_{ij}$, we say the two models are consistent, simply notated as $R_{ij}(m_i, m_j)$. QVT-Relational, or simply QVT-R, is a declarative model transformation language designed on the basis of relation theory, and thus we use it as a language in our approach to specify the relation between the concepts from different layers. Figure 4 illustrates the QVT relations we defined between business layer and service layer. The two relations on the left-hand side explains that a service in SL is related to a lane, or a non-message, non-start activity with the same name in BL. The

```
1 constraint:
2   mismatch 'Registered service missed'
3   predicate context Service
4             pre: not self.registered.oclUndefined()
5             post: not self.oclUndefined()
6 constraint:
7   mismatch 'Server overloaded'
8   predicate context Node inv self.hosted->size()<4
9   fixing let alt=self.parent.node
10          ->select(e|e.type=self.type and e.hosted->size()<4)
11          ->getFirst().hosted->add(self.hosted->getLast())
12 constraint:
13  mismatch 'Server mismatch'
14  fixing context Node inv self.restart()
```

Figure 5 Sample constraints.

```
-Mismatch
 |- Business process layer mismatch
  |- ...
 |- Service layer mismatch
  |- Service mismatch
   |- Service missing mismatch
    |- Registered service missing
    |- Required service missing
   |- Service quality mismatch
    |- Service long response time
  |- Communication mismatch
 |- Infrastructure layer mismatch
  |- Server mismatch
   |- Server overloaded
   |- Server not response
   |- Server exceptions
  |- Platform mismatch
```

Figure 6 Sample mismatch taxonomy.

middle part defines the relations between delegations (a special type of connections) and the registrations between services: If there is a delegation c connecting two services a1 and a2, and their parents are l1 and l2, then there must exist two services s1 and s2 corresponding to the two lanes (according to the relation defined before), and s1 is registeredTo s2. The right-hand side part defines a criterion for a connection to exist: If there is a connection between a1 and a2, a1 maps to service s1, a2 (or the lane of its delegated service l) maps to s2, then s1 and s2 must be matched on their input and output.

• **Constraints.** The constraints on a meta-model defines the desired model instances under this meta-model. A mismatch appears when the system state does not satisfy the constraints. Following the OMG's meta-modeling standards, we utilize the OCL language for the specification of constraints. Figure 5 shows two sample constraints that we defined for service layer and infrastructure layer. The first constraint describes that a registered service cannot be missed after the change. The second constraint is state-based. Along with the constraint, we also defined an automated fixing logic, to find another node with the same type, and transfer one component to that node.

• **Mismatch taxonomy.** In order to organize the scattered constraints into a well-organized structure, we utilize a tree-based mismatch hierarchy, which we call a *mismatch taxonomy* [11]. The elements of a mismatch taxonomy are *general* or *specific* mismatches, and a child mismatch has a "is a" relation with its parent. A sample excerpt of the taxonomy for our CMS system is shown as Figure 6. The leaf nodes in the tree are the *specific* mismatches, such as the first two ones in Figure 5. These nodes have particular and detailed constraint specifications. Alternatively, the parents of these specific mismatches are the *general* ones, such as the last one in Figure 5. According to the taxonomy, the last two constraints in Figure 5 are a pair of child and parent, which means that "Server overloaded" *is a* special kind of "Server mismatch". For specific mismatches, we require developers to provide detailed specification of predicates and fixing logics (when applicable). For general mismatches, we do not allow developers to provide predicates, because general mismatch is only a concept. A general mismatch is detected only when the constraint of one of its descendants is violated. However, we allow developers to specify fixing logics on general mismatches, which serves as a default way to resolve all its descendant mismatches. For example, we give a simple fixing logic to "Server mismatches" so as to restart the node, which means that if any kinds of server mismatches is detected, and there is no fixing logic specified for this specific mismatch, we will try to restart the server first.

4 Monitoring and Adaptation inside a layer

4.1 Run-time models

For each layer, we maintain a model instance conforming to the defined meta-model. As a runtime model, there exists a *causal connection* between the model and the system [9]. That means a system change will cause the corresponding model to change, and a model modification will influence the system as well.

There exists many different techniques to maintain the causal connection, such as API wrapping [12,13] and event correlation [14].

In this approach, we utilize our previous framework named Sm@rt to construct run-time models for different layers and maintain their causal connection with the current system [13]. Using the Sm@rt framework, users need to specify "what data we can obtain and manipulate in a specific layer" and "how to manipulate them" by using the system API or the monitoring tools in the layer. Sm@rt provides domain specific modeling languages to assist these two specifications, and automatically generate the run-time engine which maintains the causal connection between the system and the model.

Run-time models support the monitoring and adaptation of individual layers in the following way. Based on the causal connections, when the system or the context evolves on a layer, we get two model states m and m' reflecting the system state before and after the change, and monitoring is to identify what this change from m to m' means on the current layer. From the current model m', an adaptation means a modification on the model, which leads to a new model state m°, and the causal connection will automatically update the system state to be consistent with this new model state. In this way, the system is changed according to the adaptation. We will detail the monitoring and adaptation processes in the following two sections, respectively.

4.2 Monitoring and adaptation

We use the constraints defined in Section 3 to do monitoring. The two model states m and m' before and after the change are used as the input to evaluate each constraint's predicate. For a state-based predicate SP, we evaluate whether $m' \models \mathrm{SP}$, and for a change-based predicate $\mathrm{CP} = (pre, post)$, we check whether $m \models pre \rightarrow m' \models post$. If the evaluation fails, we collect the mismatch description, and the model elements that break the constraint. The monitoring is directly performed as an OCL rule evaluation. If the evaluation result is true, we finish the current constraint and go ahead with the next one. Otherwise, we throw the mismatch description defined inside this constraint as the warning of a new mismatch detected. During monitoring, all the leaf nodes in the mismatch taxonomy will be evaluated, following the document order in the taxonomy tree. If a constraint is violated, we return the mismatch name, as well as the location in the taxonomy, so that users can trace the category of the detected mismatches.

After collecting the mismatches, we perform both automated and manual adaptations to fix them. If a mismatch has a fixing logic, we evaluate the statement of this fixing logic on the new model state m', and return the result $m^\circ = \mathrm{fix}(m')$ as the new adaptation result. This automated adaptation is also performed directly using the OCL engine: We extract the fixing statement as an OCL expression, and add the context part using the one defined in the predicate. The expression is fed into the engine with the current model state m', yielding a changed state m°. If there is no fixing logic defined in the constraint, we leave the mismatch for human administrators to handle. The administrator will be provided the current model state m', and is allowed to directly modify the model into a new state m°. In the practical situation, administrators may have to tolerate some unsatisfiable constraints to keep the system serving. To support this flexibility, we regard ignoring a constraint as a special kind of adaptation.

4.3 Mismatch ranking

When there are multiple mismatches detected, we rank them first before they are listed to the users. This ranking is not only a reference for human administrators to decide the priority for handling mismatches, but also help the automated planners decide the constraints to solve.

The ranking is based on the weights of specified constraints. During the specification of constraints, we allow developers to assign weights for some of the constraints, using an integer number between 0 and 100. Bigger number means the constraint is more important. If no weights are explicitly assigned, we give the constraint a default number of 50, which means that the constraint has a medium importance. After monitoring, all the detected mismatches (i.e., violated constraints) are sorted in descending order according to their weights. The weights are dynamically tunable, either manually or automatically. Manual weight tuning happens during the monitoring period. After being presented with the list of

mismatches, the administrators are allowed to change the list, and re-arrange the weights of involved constraints according to the new order provided by the administrator. Automated tuning happens when administrators ignore some constraints. If a constraint is ignored by the administrator, then we decrease its weight such that it is lighter than all the other constraints which are in the list but not ignored.

The basic usage of this mismatch ranking is to help administrators assess the monitoring result. When there are multiple mismatches detected, the administrators will see the mismatches from the most important ones. This is especially important when there are many mismatches. The ranking is also a key feature for us to implement the automated adaptation mechanism. When executing the fixing logics specified with multiple constraints, it is possible that different fixing logics have conflicting effects on the model. For example, one fixing logic indicates to change the value of an attribute, while another logic indicates to change the same attribute into a different value. In order to detect these conflicts and warn the administrator during automated fixing, we mark all the model elements that are modified during a session of adaptation as dirty, and if a subsequent execution of fixing logic intends to change a dirty element into a different value, we will raise a warning.

5 Cross-layer monitoring and adaptation

In this section, we first introduce the techniques we utilize to maintain the runtime models, synchronize them across layers, and do monitoring and adaptation within a layer. After that, we present the algorithm to integrate these techniques together to achieve a semi-automated cross layer adaptation approach.

5.1 Bidirectional model transformation

A QVT bidirectional transformation is constituted by two functions derived from the relation between two meta-models [15]: $\overrightarrow{R} : M \times N \to N$; $\overleftarrow{R} : M \times N \to M$. The first function $\overrightarrow{R}(m, n) = n'$ takes two model state m and n as input, and returns a new state n', satisfying $(m, n') \in R$. The second function $\overleftarrow{R}(m, n) = m'$ does the same thing but in the opposite direction. The two functions satisfy three properties, namely the correctness, harmlessness, and undoability [15]. Each transformation does not construct a target model from scratch, but uses the current target model state as a reference, change it to satisfy the relation, and keep the irrelevant part unchanged.

We use bidirectional transformation to propagate changes between layers. For two layers reflected by their models in state m_i and m_j respectively, if a change is captured by m'_i, then the transformation result $m'_j = \overrightarrow{R_{ij}}(m'_i, m_j)$ contains the influence of this change on the other layer. Similarly, if an adaptation modifies the target layer model from m'_j to m''_j, the result of the other transformation $m''_j = \overleftarrow{R_{ij}}(m'_i, m''_j)$ describes how this modification effects the original layer.

We use the sample relations in Figure 4 to show how bi-transformation works. Suppose an administrator modifies the business layer model in Figure 1, and redirects the delegation of Rescue to ArmyService. If we execute the transformation according to relation delegation2Registration, using the new business layer model and the current service layer model as inputs, we will find l1 and l2 as EmergencyCentre and ArmyService, and thus s1 and s2 will be the services with the same names, and we construct a new registeredTo between them. For another example, after this change, when executing the relation Connection2Service from SL to BL, we will find a pair of s1 and s2 as EmergencyCentre and ArmyService, and the corresponding a1 and l as the activity and lane with the same name. Since s1.out is not equal to s2.in, c.invalid will be set to true. After these two transformations, we automatically find out a new BL mismatch caused by the adaptation, with the help of SL information.

5.2 The integrated algorithm

We integrate the above techniques into the cross layer monitoring and adaptation algorithm. In a system with k layers, *monitoring* returns k mismatch sets, each of which contains the mismatches detected in a particular layer. On the contrast, *adaptation* is a process to eliminate these mismatches: We first try to resolve the mismatches according to their fixing logics, and then provide the rest of the mismatches to

the administrators, so that they can use the new mismatches as a reference to make manual adaptation decisions, either to modify the model or ignore the mismatch. After each adaptation, we synchronize the modified model state to the other layer models, evaluate the constraints, and update the mismatch sets.

Algorithm 1: Cross-layer monitoring and adaptation

Ref: $\mathcal{M} = \{M_i\}, \mathcal{C} = \{C_i\}, 1 \leqslant i \leqslant k$: The meta-models and constraints of the k layers.
$\mathcal{R} = \{R_{ij}\}, 1 \leqslant i \leqslant k-1, j = i+1$: The relations between neighboring layers.
In: $\{\delta_i = (m_i, m_i')\}$,: The changes on the k models
Out: $\{m_i^\circ\}$: The model states after the adaptation
Inter: $\{\text{Msm}_i\}$: The set of mismatches. Ign: The mismatches ignored by administrators
Monitoring:
1 queue $\leftarrow \{i | 1 \leqslant i \leqslant k\}$
2 **while** queue $\neq \{\}$ **do**
3 $i \leftarrow$queue
4 **foreach** $j \in \{i-1, i+1\} \cap \{1, \ldots, k\}$ **do** $m \leftarrow \overrightarrow{R_{ij}}(m_i', m_j')$; **if** $m \neq m_j'$ **then** $m_j' \leftarrow m$, queue$\leftarrow j$
5 **foreach** i **do** $\text{Msm}_i \leftarrow \text{Eva}[\![C_i]\!](m_i, m_i')$
Adaptation:
6 **while** (Unhd$\equiv \bigcup_i \text{Msm}_i - \text{Ign}) \neq \{\}$ **do**
7 **while** ($\exists$msm $\in$ Unhd)[msm.$fix \neq \emptyset$] **do**
8 msm$\equiv(i, c, e \subseteq m_i')$; $m_i^\circ \leftarrow \text{Fix}[\![c]\!](\text{msm}, e, m_i')$; Spread (i, m_i', m_i°)
9 $(m_i^\circ, \text{Ign}_i) \leftarrow$ManualAdaptation(); Ign$\leftarrow$Ign $\cup$ Ign$_i$; Spread (i, m_i', m_i°)
10 **Procedure** Spread (i, m_i', m_i°) **begin**
11 $\text{Msm}_i \leftarrow \text{Eva}[\![C_i]\!](m_i, m_i^\circ)$
12 **for** $j \in \{i-1, i+1\} \cap \{1, \ldots, k\}$ **do**
13 $m \leftarrow \overrightarrow{R_{ij}}(m_i', m_j^\circ)$; **if** $m \neq m_j^\circ$ **then** $m_j^\circ \leftarrow m$; $\text{Msm}_j \leftarrow \text{Eva}[\![C_j]\!](m_j, m_j^\circ)$

Algorithm 1 illustrates our monitoring and adaptation algorithms. Using the meta-level specifications as references, the input is a set of k changes δ_i captured on the layer runtime models, and the output is k sets of mismatches Msm_i (for monitoring) and the new model states m_i° (for adaptation) representing the modifications to the systems on different layers.

Monitoring is implemented as a breadth-first search. We use a queue to store the layers that is not stable yet, and this queue is initialized with all the layers first. Until the queue becomes empty, we keep on executing a loop to spread the changes. In each iteration, we take one layer i out of the queue, synchronize the current state m_i' at layer to its two neighbors. If the transformation result m is not the same as the input m_j', then it means that the layer j is not stable yet, and we put it into the queue. Thanks to the *Harmlessness* property of bidirectional transformation, if a layer model already embeds the modifications from another layer, the transformation will keep the model state unchanged. In this way, the spread process will not fall into an endless loop, and we will finally reach a stable set of model states. After that, we evaluate the constraints on each model, and find the violated ones to fill the mismatch set.

Adaptation is implemented as a semi-automated loop, which does not end until the mismatch sets from all the layers are empty, or all the left mismatches are marked as ignored by administrators. Inside the main loop, we first try to resolve the mismatches that have fixing logic. For such a mismatch, we execute its fixing logic and get a new model state m_i°, and spread this new modification to the neighboring layers. Inside the spread procedure, we first re-evaluate the constraint, in order to delete the resolved mismatches and see if new ones are introduced . After that, we use bidirectional transformation to synchronize the modification result m_i° with the newest state of the neighboring model m_j°. If the result is different, then it means that the modification on layer i has influence on layer j. Thus we re-evaluate the constraints on j, and update Msm_j. In this way, we will remove the mismatches that are resolved by the modification on another layer, and also record the new mismatches on the remote modification.

When a set of mismatches are resolved automatically, we provide the remaining mismatches to the administrators. The invocation to ManualAdaptation on Line 9 will be blocked until any administrator on any layer performs a modification. The process will continue with the modified model state captured by m_i°, and the mismatches ignored by the administrator recorded in Ign_i. After that, we will do the

same spread approach as for the automated adaptation.

After an adaptation (automated or manual), the subsequent spread procedure presents exerts as follows. 1) If two mismatches from two layers i and j describe the same system fault, then the transformation of the adaptation result on i will no long cause the original mismatches on j, and thus the mismatches caused by the same source do not need to be resolved twice. 2) If the adaptation on one layer i requires the complementary modifications on other one j, the evaluation on the transformation result m_j^o will add new mismatches to the mismatch set Msm_j to indicate the required complement modification. If a new mismatch has a fix logic, the required modification will be automatically performed; otherwise, the mismatch will be a hint for further manual adaptation to complete the modification. 3) If an adaptation on one layer i is illegal because its complementary modification on another layer j (say Msm_j) cannot be resolved, then the administrator will have to ignore j, without any modification. In this situation, the backward transformation from j to i will roll the model state back on i, and throw the original mismatch again. This tells the administrator on layer i that his adaptation has failed. The *Undoability* property of bidirectional transformation [15] guarantees that such unsuccessful adaptation can be clearly rolled back.

6 Evaluation

6.1 The CMS case study

We implemented the approach on a simulated crisis management system. The simulation had a similar function and structure to that described by Popescu et al. [6]. For the sake of simplicity, we implemented it based on the Spring platform[3]. In the Business Layer, the processes were specified and executed based on Apache Camel[4]. The activities and lanes were mapped to the end points and routes in Camel, and the delegation was defined as the reference from an end point to another route. In the Service Layer, we implement the services as Java Beans, Servlets, or by the workflows defined by Camel (the services that map to lanes). The services were specified in the Spring configuration files. Finally, in the Infrastructure layer, the Beans and Servlets were running Tomcat servers.

We implemented the approach based on the Eclipse Modeling Framework (EMF[5]). We represented the information from each layer as an EMF model, and implemented a simple runtime model engine to maintain the causal connection: For the higher two layers, the engine translates XML configuration files to EMF model, and vice versa, and for the infrastructure layer, the engine retrieves and updates system state via server APIs and configuration files. Based on the EMF runtime model, we implemented the constraint evaluation and bidirectional model transformation using the Eclipse OCL engine and the mediniQVT[6] transformation engine, respectively.

We describe two typical scenarios as follows. The first scenario simulates how to get the mismatches on all the layers caused by the crash of a node in IL. We stopped the Tomcat node, and this change was captured by the IL runtime model with the disappearance of the first Node. The relation between IL and SL was defined that the components maps to the services with the same names, and thus the transformation from IL to SL resulted in the disappearing of GetGPS and MedicalService. The subsequent transformation from SL to BL, according to the relations defined in Figure 4, caused one activity and one lane to disappear. After the synchronization, the evaluation of the SL model according to the constraints as defined in Figure 5 yields two "*missing registered services*" mismatches. The evaluation on the BL model yields the *missing activities* and *missing delegation targets* mismatches. These mismatches are returned to different system administrators. The second scenario shows how the approach assists system administrators in adapting the system, as illustrated in Figure 7. Following the description in Subsection 2.1, the adaption started from a BL administrator who redirects Rescu to ArmyService (marked as 1 in the figure). The first transformation yields a new SL model with a new registration

3) http://www.springsource.org.
4) http://camel.apache.org.
5) http://www.eclipse.org/modeling/emf/.
6) http://projects.ikv.de/qvt.

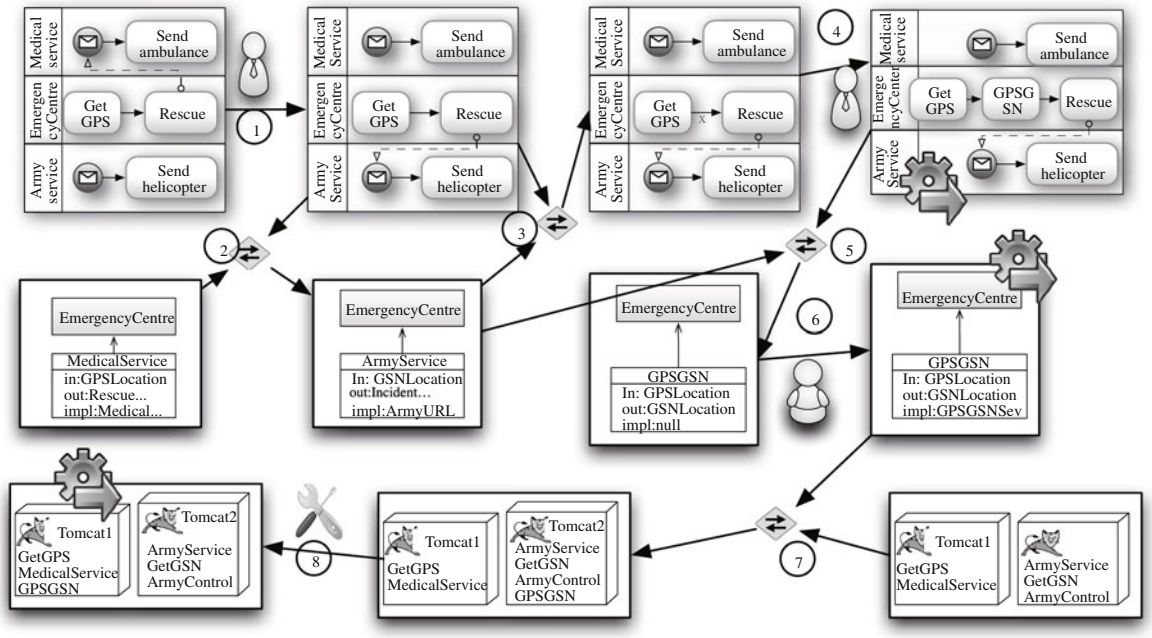

Figure 7 Adaptation scenario.

link, and since this model was changed, the adaptation engine went on to spread it, and the backward transformation from SL to BL changed a connection to *invalid* (step 3, as shown in Section 5), because the two services are not compatible. This new mismatch led the BL administrator to add an adapter between GetGPS and Rescue (4), and the transformation adds a new service in SL model, and automatically generate its input and output types (step 5), but leave impl as empty. The mismatch of "*service not implemented*" calls for SL maintainers to implement the service (6). The default transformation (7) deploy the new service to Tomcat1, but the fixing logic of the constraints shown in Figure 5 automatically migrates the component to the other server for balance (8). Finally, no transformation would cause new changes on the models, and we execute the final models of the three layers back to the system.

The scenarios reveal the following features of our approach. 1) *Separation of concerns.* At design time, the mismatches are specified on the concepts within a particular layer, and no explicit links need to be defined between mismatches from different layers. At runtime, administrators handle mismatches and do adaptations on their own layers. 2) *Automation.* The spread of mismatches and adaptations are automatically performed by the engine. We also support the automated adaptation to resolve some mismatches, provided that the fixing logics are defined. 3) *Productivity.* The inputs required by this approach are high-level meta-models, constraints, and relations, in standard modeling languages. Users do not need write any low-level code. The specifications are reusable between systems with similar layers.

6.2 The smart office case study

We did another case study on smart office systems. The system comprises a set of sensors installed in different corners of the building to capture the physical environment of the office rooms. We also attach RFID tags to the office members' entrance cards, as well as to the key public assets and personal properties. A set of RFID readers are deployed in different rooms to locate the members and staffs by sensing the tags in their scopes. The key function of the system is to automatically detect the mismatches among the members, assets and environment, e.g., a member forgets his personal belonging in a meeting room, or forgets to turn off the heating system, etc.

We divided this system into two layers: A device layer contains all the information we can get from the physical world, but it is not easy to be used for detecting the mismatches as we described before, because of the concept gap between these technical concepts, and the ones we mentioned in the mismatch

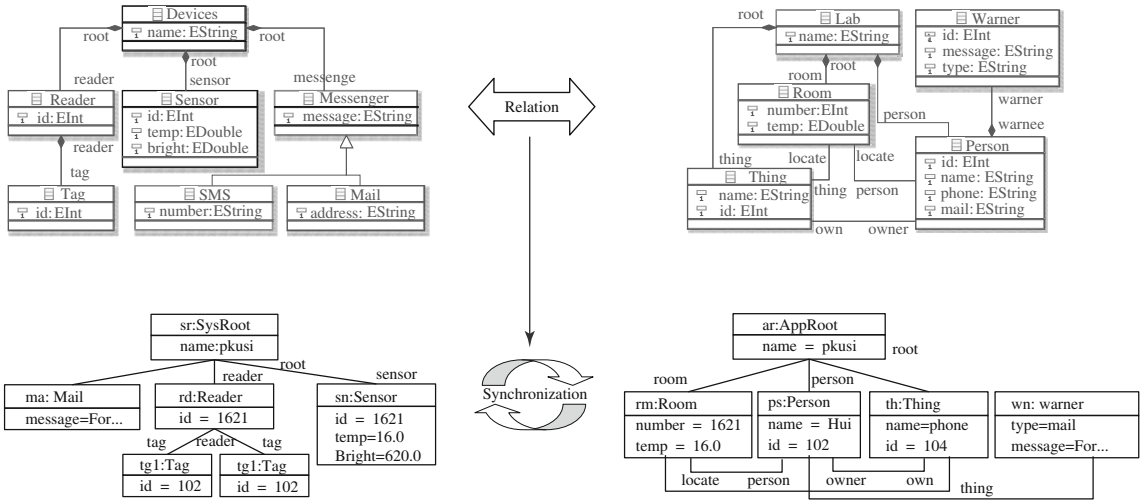

Figure 8 Smart office models and meta-models.

```
1  transformation RFIDLab(dev:RFID,app:Lab){
2    key RFIDRoot{name}; key Sensor{id};
3    top relation RR{name:String; dev r: Devices{name=name}; abs ra : Lab{name=name};}
4    top relation SR{id:Integer; temp:Real; rs: Devices; ra:Lab;
5      dev sensor:Sensor{id=id,temp=temp,root=rs}; abs room:Room{number=id,temp=temp,root=ra};
6      when{RR(rs,ra);}}
7    top relation RTRP{rid:Integer; tid:Integer; rs:RFIDRoot; ra:LabRoot;
8      dev reader:Reader{id=rid,root=rs}; dev tag:Tag{id=tid,reader=reader};
9      abs room:Room{number=rid,root=ra}; abs person:Person{id=tid,root=ra, locate=room};
10     when{RR(rs,ra) and ra.person–>collect(id)–>includes(tid);}}}
```

Figure 9 Sample QVT relational transformation.

description before, i.e., persons, rooms, etc. We define an abstract layer of this smart office system, which is directly comprised of the concepts appearing in the mismatch description.

Figure 8 illustrate the model-based structure on the monitoring of this smart office system. The left-hand side of the figure shows the meta-model and model of the system layers. The meta-model defines the concepts such as sensors, RFID readers, tags, etc. The run-time model of this layer is an instance of this meta-model, with each model element corresponding to a real system element, and the states of them describes the current system state of the corresponding device. The right-hand side of the figure shows the meta-model and a sample run-time model of the abstract layer. Figure 9 describes the relation we defined between the models of device and abstract layers. For example, the relation SR in line 4 means each sensor maps to a room, provided that it has the same id. The temperature of the rooms equals the reading from the sensors. The following relation RTRP defines that if a reader finds a tag, and if the tag belongs to a person, then the corresponding person is located in the room where the reader is installed.

With the help of a bidirectional transformation between the two models, we ensure that the abstract model is synchronized with the device layer model, and in turn synchronised with the current system states. In this way, we support the monitoring of the whole smart office system from the abstract layer. For example, the monitoring scenario we described above can be specified as the OCL constraints shown in Figure 10, which describes that if a Thing is not located in the same room as its owner, we detect a mismatch "lost personal stuff". And the fixing logic is to send the person a message about this loss.

7 Related work

Existing approaches on cross-layer adaptation often follow a centralized way. Guinea et al.'s framework [5] allows different techniques to monitor different layers, but employs a centralized adaptation agent to

```
1 constraint:
2   mismatch 'lost␣personal␣stuff'
3   predicate context Thing inv: self.locate <> self.owner.locate
4   fixing self.owner.warner += object Warner{message:='Forget␣something?',
5     type:=(if self.name='phone' then 'mail' else 'sms' endif)}
```

Figure 10 Sample constraint of smart office scenario.

collect the events and analyze the violation of key performance indicators. Popescu et al. [6] execute adaptations following a set of predefined templates, and in these templates, users have to explicitly define for each adaptation solution, what mismatches would be raised on other layers. In contrast, we adopt a decentralized approach, where mismatches and adaptations are defined and performed separately on different layers, and we automatically calculate the dependency using the relation between layer concepts. From this perspective, our approach is related to ECMAF [16], which uses a dependency model between the components to enable the detection of a component that contains the root cause of a mismatch. However, by using bidirectional transformation, we achieve the spread of mismatches and adaptations across complicated relations, rather than the simple traceability between components.

Model driven engineering techniques, especially runtime models, are widely used in dynamic adaptation systems. Morin et al. [17] describe a typical architecture for these approaches, i.e., to capture the system information as runtime models, analyze and reconfigure the models, and finally execute the changes back. This paper extends the typical ideas to multi-layer systems, using multiple runtime models for different layers, and introduces bidirectional transformation to associate the runtime models. Baresi et al. [18] also present a model-driven approach to the management of multilayer service-based systems. But their concern is how to generate the monitoring engines from the models defined in business and service layers. Such a top-down approach requires the lowest layer to contain all the information from other layers, and scarifies the flexibility of the approach.

Our approach is a novel usage of bidirectional model transformation. Unlike the classical usage of bi-transformation at design time [15], we utilize the transformation together with constraint evaluation and multi-user model changes to form a runtime monitoring and adaptation process.

8 Conclusion

This paper presents a model-based approach to the cross-layer system monitoring and adaptation. We provide the meta-modeling languages for system experts to specify the layers, the relations between them, as well as the constraints on each layers, and implement the engine to assist monitoring and adaptation based on the specifications. We evaluated it on a simulated service-based crisis management system.

The approach is by far an initial attempt. We can identify the cascaded mismatches and complementary adaptations only if all the information related to them can be described by the pre-defined runtime models, model relations and constraints. As a future plan, we will evaluate the approach on typical but more complicated target systems, and investigate the extension of modeling and relation specification languages to cover all mismatches and adaptations.

Currently, we simply employ an existing QVT engine, the mediniQVT, to realize the bidirectional transformation based cross-layer monitoring and adaptation. Another future plan is to evaluate the usage of bidirectional transformation on more complicated cross-layer adaptation scenarios, summarize the required properties from bi-transformation to support correct mismatch and adaptation spread, and extend the existing engines to satisfy these properties. The current algorithm is straightforward, and may perform unnecessary transformations and evaluations for particular scenarios. To improve the performance of the approach, we will optimize the process, and investigate the usage of incremental bi-transformations. At this stage, the approach relies on the system experts to ensure the effectiveness and consistency of the meta-models, constraints, and relations. We will consider the static verification of these meta-level specifications as an assistant to designers.

Acknowledgements

This work was supported in part by Science Foundation Ireland grant 10/CE/I1855 to Lero—the Irish Software Engineering Research Centre (www.lero.ie).

References

1 Yang F, Lv J, Mei H. Technical framework for Internetware: an architecture centric approach. Sci China Ser F-Inf Sci, 2008, 51, 6: 610–622
2 Kazhamiakin R, Pistore M, Zengin A. Cross-layer adaptation and monitoring of service-based applications. In: ServiceWave Workshops. Stockholm: Springer, 2010. 325–334
3 Yuan W, Nahrstedt B, Adve S, et al. Grace-1: cross-layer adaptation for multimedia quality and battery energy. IEEE Trans Softw Eng, 2006, 5: 799–815
4 Zengin A, Kazhamiakin R, Pistore M. Clam: cross-layer management of adaptation decisions for service-based applications. In: Internetional Conference on Web Services. Washington DC: IEEE, 2011. 698–699
5 Guinea S, Kecskemeti G, Marconi A, et al. Multi-layered monitoring and adaptation. In: Kappel G, Maamar Z, Motahari-Nezhad H R, eds. Service-Oriented Computing. Berlin/Heidelberg: Springer, 2011. 359–373
6 Popescu R, Staikopoulos A, Brogi A, et al. A formalized, taxonomy-driven approach to cross-layer application adaptation. ACM Trans Auton Adapt Syst, 2013, 7: 7–24
7 Cheng B, de R Lemos, Giese H, et al. Software engineering for self-adaptive systems: a research roadmap. In: Cheng B H C, Lemos R, Inverardi P, et al., eds. Software Engineering for Self-Adaptive Systems. Dagstuhl: Springer, 2009. 1–26
8 France R, Rumpe B. Model-driven development of complex software: a research roadmap. In: Future of Software Engineering, Minneapolis, 2007. 37–54
9 Blair G, Bencomo N, France R. Models@run.time. Computer, 2009, 42: 22–27
10 Song H, Huang G, Xiong Y, et al. Inferring meta-models for runtime system data from the clients of management APIs. In: Models Driven Software Engineering, Lauguage and Systems, Oslo, 2010. 168–182
11 Popescu R, Staikopoulos A, Liu P, et al. Taxonomy-driven adaptation of multi-layer applications using templates. In: International Conference on Self-Adaptive and Self-Organizing Systems, Budapest, 2010. 213–222
12 Sicard S, Boyer F, de Palma R. Using components for architecture-based management: the self-repair case. In: International Conference on Software Engineering, Leipzig, 2008. 101–110
13 Song H, Xiong Y, Chauvel F, et al. Generating synchronization engines between running systems and their model-based views. In: Models in Software Engineering, Denver, 2009. 140–154
14 Schmerl B, Aldrich J, Garlan D, et al. Discovering architectures from running systems. IEEE Trans Softw Eng, 2006, 32: 454–466
15 Stevens P. Bidirectional model transformations in QVT: semantic issues and open questions. In: Model Driven Software Engineering, Languages and Systems, Nashville, 2007. 1–15
16 Zeginis C, Konsolaki K, Kritikos K, et al. Ecmaf: an event-based cross-layer service monitoring and adaptation framework. In: Liu C F, Ludwig H, Toumani F, et al., eds. Service Oriented Computing. Berlin/Heidelberg: Springer-Verlag, 2012. 147–161
17 Morin B, Barais O, Jézéquel J, et al. Models@run.time to support dynamic adaptation. Computer, 2009, 42: 44–51
18 Baresi L, Caporuscio M, Ghezzi C, et al. Model-driven management of services. In: IEEE European Conference on Web Services, Lugano, 2010. 147–154