Instant and Incremental QVT Transformation for Runtime Models

Hui Song, Gang Huang, Franck Chauvel, Wei Zhang, Yanchun Sun, Weizhong Shao, Hong Mei

Instant and Incremental QVT Transformation for Runtime Models

Hui Song, Gang Huang*, Franck Chauvel, Wei Zhang, Yanchun Sun, Weizhong Shao, Hong Mei

Key Lab of High Confidence Software Technologies (Ministry of Education)
School of Electronic Engineering & Computer Science, Peking University, China
{songhui06,franck.chauvel,zhangwei11,sunyc}@sei.pku.edu.cn
{hg,wzshao,meih}@pku.edu.cn

Abstract. As a dynamic representation of the running system, a runtime model provides a model-based interface to monitor and control the system. A key issue for runtime models is to maintain their causal connections with the running system. That means when the systems change, the models should change accordingly, and vice versa. However, for the abstract runtime models that are heterogeneous to their target systems, it is challenging to maintain such causal connections. This paper presents a model-transformation-based approach to maintaining causal connections for abstract runtime models. We define a new instant and incremental transformation semantics for the QVT-Relational language, according to the requirements of runtime models, and develop the transformation algorithm following this semantics. We implement this approach on the mediniQVT transformation engine, and apply it to provide the runtime model for an intelligent office system named SmartLab.

1 Introduction

Modern systems provide many kinds of data during runtime, such as their internal states and configurations, the status of their tasks, and even their physical environment. Runtime model is a promising approach towards the manipulation of such runtime system data [1], allowing developers to monitor and control the system in a model-based way. In this paper, we focus on the structural runtime models that can be regarded as dynamic *object diagrams* representing the snapshots of running systems. A key issue for such runtime models is to maintain their *causal connections* with the systems. That means when the systems change, the models should change accordingly and instantly, and vice versa.

Many research approaches provide structural runtime models for different systems [2–5]. These approaches focus on wrapping the low-level management capability of the target systems into model-based interfaces, and thus their runtime models directly reflect the system data. However, for a target system, only one such reflective runtime model is usually not enough. To meet the different

* corresponding author

requirements and concerns on system monitoring and control, we need to abstract the reflective model again in different concepts and organizations. Such abstract runtime models act as different views of the reflective runtime model. Due to the heterogeneity between the abstract model and the running system, maintaining their causal connection is difficult.

In this paper, we present a model-transformation-based approach to maintaining the causal connection for abstract runtime model, by propagating changes between this abstract model and the existing reflective model of the target system. The change propagation is guided by the relation between the two models, specified in the QVT-Relational language. The challenge here is twofold. First, the changes on the systems and the runtime models are usually small but frequent, and thus the traditional batching QVT transformation that transforms the whole model each time is not efficient. We need an instant (the transformation is triggered instantly after each change) and incremental (the execution is based on the change but not the whole model) transformation appraoch. Second, the relations between models and systems are usually bidirectional rather than bijective. That means for one system change, there may be multiple candidate abstract changes that all obey the relation, and vice versa. Therefore a clear and determinate semantics of the transformation need to be defined.

The contributions of this paper can be summarized as follows.

- We define an instant and incremental transformation semantics for QVT-Relational language, and formulate three properties, namely *consistency*, *stability* and *restorability*, reflecting the requirements of runtime models.
- We develop the transformation algorithm according to the semantics. we analyze the impact of the input change and only re-evaluate the influenced relations and model elements. The impact analysis is based on the QVT rule, the change type, and the trace of previous transformations.
- We implement an instant QVT transformation engine, on the basis of the mediniQVT. We apply this engine to provide the runtime models for an intelligent office system named SmartLab.

The rest of this paper is structured as follows. Section 2 explains the problem based on a running example. Section 3 and Section 4 present the semantics and algorithm of our transformation for runtime models. Section 5 evaluates the approach. Section 6 concludes the paper, with discussions and our future plans.

2 The Running Example

2.1 The SmartLab system

To improve the working condition, the Software Institute of Peking University sets up a smart office system in its office building. We installed sensors in the rooms to measure the physical environment such as temperature, brightness, etc. We also installed an RFID (Radio Frequency Identification) reader in each office or meeting room. Every member in the institute has a unique RFID tag, stuck

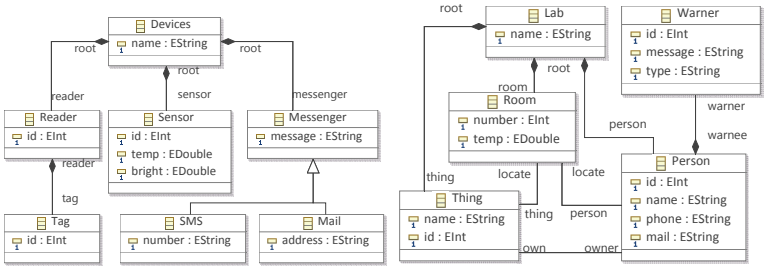

Fig. 1. The reflective and abstract meta-models for SmartLab

on his/her badge card. Some public assets and personal effects also have unique RFID tags bound with them. The tags termly transmit unique radio signals, which can be detected by the reader located in the same room.

Using these devices, SmartLab monitors the status of the whole institute, and interact with the institute members via Email, short message service (SMS), etc. Here are two exemplar monitoring scenarios: 1) **Missing personal effects**. After meetings, people may leave their personal effects in the meeting room, such as mobile phones or keys. SmartLab warns the owners when this happens. 2) **Leaving the air-conditioners on**. People may exit a room without turning off the air-conditioner, wasting electricity. For such situations, SmartLab warns the persons in nearby rooms.

2.2 The Runtime Models for SmartLab

Based on our earlier work [5, 6], we provide a reflective runtime model for SmartLab, and an excerpt of its meta-model is shown in the left of Figure 1. The classes directly define the concepts specific to the devices, and the properties define the data that can be retrieved from them. However, this reflective runtime model is still not proper for the above scenarios, because it represents the data in the solution-space which has a gap between the problem-space concepts, such as *persons*, *things*, *rooms*, etc., and cannot carry the problem-specific information such as the ownership relation between persons and things. Therefore, we define an abstract runtime model as shown in the right part of Figure 1. Using this abstract runtime model, the first scenario can be implemented in a straightforward way: If the `locate` values of a `Thing` and its `owner` are not the same, then create a new `Warner`, and add it to the owner's `warner` list.

We need to maintain the *causal connection* between the abstract model and the system, e.g., if a new `tag` is detected by a `Reader`, then the `Person` (or `Thing`) should `locate` in the `Room`, and if a new `Warner` is created, a `Messenger` should be created. The causal connection is guided by the relation between the two models, specified as a QVT-Relational rule in Figure 2. The rule is constituted by a set of `relations`: `RR` defines that the root elements are mapped if they have the same name. `SR` defines that a `Sensor` maps to a `Room` with the same `number` and `temp` values, if their roots are mapped. `RTRP` defines that if there is a pair of

Fig. 2. Sample QVT relational transformation

```
1  transformation RFIDLab(sys:RFID,app:Lab){
2    key RFIDRoot{name}; key Sensor{id}; key Room{number};...
3    top relation RR{  name:String;
4      sys rs : Devices{name=name}; abs ra : Lab{name=name}; }
5    top relation SR{
6      id:Integer; temp:Real; rs:Devices; ra:Lab;
7      sys sensor:Sensor{id=id,temp=temp,root=rs};
8      abs room:Room{number=id,temp=temp,root=ra};
9      when{RR(rs,ra);} }
10   top relation RTRP{
11     rid:Integer; tid:Integer; rs:RFIDRoot; ra:LabRoot;
12     sys reader:Reader{id=rid,root=rs}; sys tag:Tag{id=tid,reader=reader};
13     abs room:Room{number=rid,root=ra};
14     abs person:Person{id=tid,root=ra,locate=room};
15     when{RR(rs,ra) and ra.person->collect(id)->includes(tid);}  }
16   top relation RTRT{ ... }
17   top relation SMSWarner{
18     phone:String; message:String;rs:Devices;ra:Lab;
19     sys sms:SMS{number=phone,message=message,root=rs};
20     abs person:Person{phone=phone,root=ra};
21     abs warner:Warner{message=message,warnee=person,type='phone'};
22     when{RR(rs,ra);} }
23   top relation MailWarner{ ... } }
```

Reader and Tag, and the Tag id is one of the Persons ids, then this Person is located in the Room. RTRT is similar. Finally, SMSWarner means that a Person and its Warner in type of "phone" map to an SMS. The relations illustrate the heterogeneity between the two models, e.g., both Sensors and Readers map to Rooms, and the containment association between Readers and Tags map to horizontal association from the Rooms to either Persons or Things. It is not straightforward to infer an abstract change from the system one, and vice versa.

2.3 Model Transformation for Runtime Models

Figure 3 summarizes our approach to supporting abstract runtime models. From the reflective runtime model from our previous work [5, 6], developers define the abstract meta-model according to the problem concepts, and the relation between it and the reflective one, using MOF and QVT-R[7], respectively. Here QVT-R is a natural choice, because it is originally designed for specifying the *relation* between models, rather than the transformation imperatives. Following the provided meta-models and the relation, our transformation engine propagates changes at runtime between the abstract model and the reflective one.

The engine requires a specific semantics and execution of QVT-R, because of the following features of runtime models. 1) The users of runtime models usually

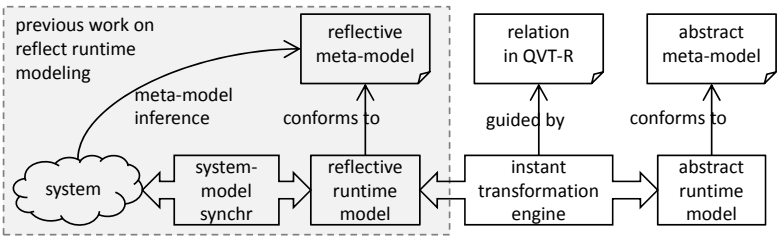

Fig. 3. Model transformation for runtime models

require to see the effects of changes and manipulations immediately, and thus we need *instant* transformation that is triggered by each change. Moreover, as the model scale is big and the changes are small but frequent, the transformation should be *incremental*, only considering the part of the model impacted by the change. 2) The users require a clear and determinate expectation about the causal connection between the model and the system. However, since the relations between are bidirectional [8], for a change on one model, there may be multiple candidate changes on the other model that all satisfy the relation. For example, considering `RTRP` in Figure 2, if a `Tag` escapes from a `Reader`, we can either delete a `Person` or just reset its `locate` value. Therefore, we need to formulate the semantics for this change-to-change QVT transformation, and this semantics should meet the common requirements of runtime system monitoring and manipulation. 3) The relation is between a model and its view, rather than two totally different models, and thus it will not be extremely complicated. Therefore, we can ignore some sophisticated syntax and usage of QVT-R.

3 The Semantics

This section defines the semantics of our instant and incremental QVT transformation for runtime models, and formulates the properties that must be satisfied.

We first abstract the three inputs: The reflective meta-model S defines the set of all the states of the reflective model, and the abstract meta-model A defines the set for the abstract model. The literal meaning of the transformation is a relation $T \subseteq A \times S$. If $(a, s) \in T$, we say the two models are consistent.

The causal connection between a model and a system has two aspects [2], i.e., *introspection* that propagates the system changes in the abstract model, and *reconfiguration* that propagates the abstract changes back. We use Δ_S and Δ_A to denote all the possible changes on the two models, respectively. The causal connection following a transformation T are two functions on the models and their changes: $\texttt{Intro}_T : S \times A \times \Delta_S \to S \times A$; $\texttt{Recon}_T : S \times A \times \Delta_A \to S \times A$. To support the above functions, the instant transformation maintains two live models, and for each time of execution, it takes the change on one model as an input and output the change on the other model.

$$\overrightarrow{T}_{S\times A} : \Delta_S \to \Delta_A; \quad \overleftarrow{T}_{S\times A} : \Delta_A \to \Delta_S$$

Using this incremental transformation, we implement $\texttt{Intro}_T$ by calculating the abstract change δ_a from the system change δ_s and then merging δ_a into the original abstract model a: $\texttt{Intro}_T(s, a, \delta_s) = (s + \delta_s, a + \overrightarrow{T}_{s,a}(\delta_s))$. For $\texttt{Recon}_T$, we first calculate the system change δ_s, and merge it to s. Since merging changes to the running system does not always lead to the expected effect, we reflect the side-effects back to the abstract model: $\texttt{Recon}_T(s, a, \delta_a) = (s + \delta_s + \delta'_s, a + \delta'_a)$, here, $\delta_s = \overleftarrow{T}_{s,a}(\delta_a)$, δ'_s is the side-effects of δ_s, and $\delta'_a = \overrightarrow{T}_{s+\delta_s, a+\delta_a}(\delta'_s)$.

Considering the requirements of runtime models, and also referring to the properties of classical QVT transformations [9], we define the following three properties for our incremental transformation, in forms of the post-conditions on the result from any input $(s, a) \in T, \delta_s \in \Delta_S, \delta_a \in \Delta_A$.

Property 1. **Consistency**. First of all, after merging the input and resulted changes, the two models must be consistent.

$$(s + \delta_s, a + \overrightarrow{T}_{s,a}(\delta_s)) \in T;\ (s + \overleftarrow{T}_{s,a}(\delta_a), a + \delta_a) \in T$$

The first part of *consistency* ensures that after `Intro` or `Recon` the abstract model correctly represents the system state, and the second part ensures that the changes executed to the system conforms to the intention of abstract changes.

Property 2. **Stability**. If the input change on one model does not violate the relation, it should not cause any change on the other model.

$$(s + \delta_s, a) \in T \Rightarrow a + \overrightarrow{T}_{s,a}(\delta_s) = a;\ (s, a + \delta_a) \in T \Rightarrow s + \overleftarrow{T}_{s,a}(\delta_a) = s$$

For `Intro`, *stability* ensures that the irrelevant system changes (such as the change of brightness) and intermediate changes (such as detecting a new tag, but having not got its id) do not disturb the monitoring agents. For `Recon`, it not only ensures that the irrelevant abstract changes (such as changing the ownership relation between persons and things) do not influence the system, but also ensures the relevant abstract changes remain stable: The side-effect of valid system writing is usually just a complement to the original change, e.g., adding a `Mail` to the `Devices.messager` will cause the `Mail.root` set to the root element. Such complementary side-effects should not influence the original abstract change, so that the users can manipulate the model in a coherent way.

Property 3. **Restorability**. After a change δ_s and its propagation result δ_a lead the two models to $s+\delta_s$ and $a+\delta_a$, the opposite change δ_s^{-1} and its propagation result should restore both models back. The other direction is the same.

$$\overrightarrow{T}_{s,a}(\delta_s) = \delta_a \Rightarrow a + \delta_a + \overrightarrow{T}_{s+\delta_s, a+\delta_a}(\delta_s^{-1}) = a$$

$$\overleftarrow{T}_{s,a}(\delta_a) = \delta_s \Rightarrow s + \delta_s + \overleftarrow{T}_{s+\delta_s, a+\delta_a}(\delta_a^{-1}) = s$$

We require *restorability* based on the following reasons. First, it is a usual case that the users undo their last change on the runtime model, and their intention is to restore the system back. Second, the system changes usually happen in

Table 1. The modifications and their inverses

description	μ	μ^{-1}
set the value of $e.p$ from v to v'	$\mathsf{set}(e, p, v, v')$	$\mathsf{set}(e, p, v', v)$
add v to the set $e.p$	$\mathsf{insert}(e, p, v)$	$\mathsf{remove}(e, p, n)$
remove v from the set $e.p$	$\mathsf{remove}(e, p, v)$	$\mathsf{insert}(e, p, v)$
create e of class c, with $id = v$	$e \leftarrow \mathsf{new}(c, id, v)$	$\mathsf{delete}(e, id, v)$
delete the existing element e	$\mathsf{delete}(e, id, v)$	$e \leftarrow \mathsf{new}(c, id, v)$

couples, e.g., a person enters a room and then exits, a light is turned on and off again. Coupled changes restore the system state and this should be reflected on the abstract model. Third, for invalid system changes (such as trying to reset the temperature value of a sensor), the side-effect is their inverses, and when propagating them back, the original abstract changes should be clearly rolled back. Finally, *Restorability* and *stability* together allow the abstract model to carry the information that is irrelevant to the system. Since such information does not influence the relation, the transformation could change it any time without violating the relation. These two properties prevent it from changing this information arbitrarily.

4 The Instant and Incremental Transformation Algorithm

Our basic idea is to analyze the impact of the input changes to reduce the scope of execution. The impact analysis is based on the syntactical feature of QVT rules, the type of changes, and the trace recorded from previous executions.

A QVT-R transformation T is constituted by a set of *relations*. A relation has several *domains*, each with a class from the meta-models. The goal of QVT transformation is to *enforce* each of these primitive relations. For each relation, the engine tries to bind model elements to its domains, by *matching* the domain patterns. If no elements can be bound to a domain, the engine creates new elements or updates existing ones. The detailed (but informal) semantics of these `PatternMatching` and `CreateOrUpdate` operations can be found in the QVT standard [7]. For batching transformation, each time the engine checks and enforces all relations, and does pattern matching in the scope of all model elements. Our incremental transformation is also based on the enforcement of primitive relations, but we screen out the irrelevant relations and shrink the scope of model elements according to the input changes.

A *change* is a set of primitive modifications, following Alanen et al.'s definition [10]. Table 1 lists the five kinds of modifications we support. Each modification μ has an inverse μ^{-1}. When propagating a change, we deal with its modifications one by one, in the order of new, insert, set, remove, and delete[10].

A *trace* is a set of *relation instances*. An instance records a composition of model elements bound to the domains of the relation, and these elements satisfy the relation. We also record the change on the source model that causes this instance to be established, and the change on the target model calculated by the

Algorithm 1: The Instant and Incremental Transformation

```
1  function InstantTrans: (s, a, μ, tr) → (δ_a, tr')
2  δ_a←{}, tr'←tr
3  foreach r ∈ T : ∃d ∈ dom(r), μ.e.class = d.c do
4      if μ = set[e, p, v, v'] ∨ μ = insert[e, p, v] ∨ μ = remove[e, p, v] then
5          if p is mentioned by any patterns in r then
6              (δ''_a, tr'')←ReEvaluate(r, tr', s, a, μ)
7              tr'←tr''; δ_a←δ_a ∪ δ''_a
8      else if μ = e ← new[T, id, v] then
9          if e satisfies the pattern of d then
10             (δ''_a, tr'')←Construct(τ : {relation ↦ r, d ↦ e}, s, a, μ, tr', φ)
11             tr'←tr''; δ_a←δ_a ∪ δ''_a
12     else if μ ∈ delete then
13         foreach τ ∈ tr : rule(τ) = r ∧ μ.e = elem(τ, d) do
14             (δ''_a, tr'')←Destroy(τ, s, a, μ, tr')
15             tr'←tr''; δ_a←δ_a ∪ δ''_a
16 return (δ_a, tr')
```

enforcement. For each transformation, the trace can be accumulated from the previous executions on the changes that create the models from scratch, or can be created at once by a batching transformation.

4.1 The Algorithm

In the rest of this section, we present our algorithm to propagate changes and update the trace step by step. The following algorithm is in the direction from the reflective to the abstract model, and the other direction is the same.

The main algorithm *InstantTrans* takes as input the original models s and a, the modification μ on s, and the previous traces tr. It outputs the change δ_a and the new trace tr'. We initiate δ_a as empty, and tr' as the original tr (Line 2). In the main body, we first screen the relations, and only consider the ones whose domain classes include the class of μ. We handle the left relations according to the type of μ: For a set, insert or remove (Line 4), only if the modified property is mentioned in r, we *ReEvaluate* it. For a new, since there may be new compositions of model elements containing e that satisfies r, we *Construct* new relation instances, starting from a partial relation instant τ with e bound to the proper domain. For a delete, we *Destroy* all the existing instances of r that have been bound with the deleted element. After each iteration on a relation, we update the trace, and unite the resulted changes.

To *ReEvaluate* a relation r, we first enumerate the instances of r that are bound with the modified element $\mu.e$. For each instance τ, we `check` the relation again and *Destroy* it if it fails now. The modification may cause new compositions of elements to satisfy the relation, and thus we seek and construct new binding

Algorithm 2: Re-Evaluate the QVT Rules

```
17 function ReEvaluate(r, tr, s, a, μ) → (δ_a, tr')
18 δ_a←{}; tr'←tr
19 foreach τ ∈ tr : rule(τ) = r ∧ ∃d_e ∈ dom(r) : μ.e = elem(τ, d_e) do
20     if ¬check(r, τ, s + μ, a) then
21         (δ''_a, tr'')←Destroy(τ, s + μ, a, tr')
22         tr'←tr''; δ_a←δ_a ∪ δ''_a
23     τ'←{relation ↦ r}
24     foreach d ∈ dom(r) : d = d_e ∨ μ.p is not mentioned by d do
25         τ'←τ' ∪ {d ↦ elem(τ, d)}
26     (δ''_a, tr'')←Construct(τ', s, a, μ, tr', φ)
27     tr'←tr''; δ_a←δ_a ∪ δ''_a
28 if no such τ is found then
29     if e satisfies the pattern of d then
30         (δ''_a, tr'')←Construct(τ : {relation ↦ r, d ↦ e}, s, a, μ, tr', φ)
31         tr'←tr''; δ_a←δ_a ∪ δ''_a
32 return (δ_a, tr')
```

compositions. Here we do not exhaustively enumerate all the possible compositions, but utilize the existing bindings as a reference. Note that the property $\mu.p$ is only mentioned by part of the domain patterns. Take `SMSWarner` in Figure 2 as an example, the property `SMS.message` is not mentioned by the `person` domain, since the pattern does not contain any direct or transitive reference to this property. If there is any new binding compositions emerging to satisfy r, then it must be because the modification makes an element satisfy the pattern that mentions this property. Therefore, we fix the elements bound to irrelevant domains, leaving the other domains as free, and then use this partial binding as a seed to construct new instances. If there is not any existing relation instance as reference, we construct relations just as if this element is newly created.

Construct is similar to the classical enforcement semantics of QVT, but has a partial relation instance as a seed, with some domains bound. The input also includes a δ_a that records the accumulated changes to bind these domains. If the seed τ is already complete, and satisfies the relation, this τ is a successful instance. We return δ_a as the final change, add the new instance τ into the trace, and record the source change $\{\mu\}$ and the target change δ_a under τ. If the input τ is not complete yet, we take one free domain d, perform `PatternMatching` on it to find all the elements that can satisfy the domain pattern, and store the elements in the set `cand` (for "candidate"). If no binding is found, we try to create new elements or update existing ones, and regard the result e as a candidate. Finally, we try to bind each element e in *cand*, and invoke *Construct* recursively to bind the rest of the free domains. Another thing to consider is the dependency between relations. Due to the establishment of this relation, some other relations that depends on it may be satisfied. Therefore, after constructing a new relation

Algorithm 3: Construct and Destroy Relation Instances

33 **function** *Construct*: $(\tau, s, a, \mu, tr, \delta_a) \to (\delta_a, tr')$
34 **if** *every* $d \in \texttt{dom}(r)$ *is bound in* $\tau \wedge \texttt{check}(\tau, s + \mu, a + \delta_a)$ **then**
35 $tr' \leftarrow tr \cup \{\tau\}$; $\tau.\texttt{rec}_s \leftarrow \{\mu\}$; $\tau.\texttt{rec}_a \leftarrow \delta_a$
36 **foreach** $r' \in T : when(r') = r$ **do**
37 $(\delta''_a, tr'') \leftarrow$ *Construct*$(r', tr', para, s, a, \mu, \delta_a)$
38 $\tau.\texttt{chd} \leftarrow \tau.\texttt{chd} \cup (tr'' - tr')$; $\delta_a \leftarrow \delta_a \cup \delta''_a$; $tr' \leftarrow tr''$
39 **return**(tr', δ_a)
40 **else if** $\exists d \in \texttt{dom}(r) : d \notin \texttt{dom}(\tau)$ **then**
41 $\texttt{cand} \leftarrow \texttt{PatternMatching}(\tau, d, s + \mu, a)$
42 **if** $\texttt{cand} = \phi \wedge$ *d is an enforce app domain* **then**
43 $(\delta'_a, e) \leftarrow \texttt{CreateOrUpdate}(\tau, s, a + \delta_a)$; $\texttt{cand} \leftarrow \texttt{cand} \cup \{e\}$
44 **foreach** $e \in \texttt{cand}$ **do** *Construct*$(r, tr, \tau \cup \{d \mapsto e\}, s, a, \delta_a \cup \delta'_a)$

45 **function** *Destroy*: $(\tau, s, a, \mu, tr) \to (\delta_a, tr')$
46 $tr' \leftarrow tr - \{\tau\}$; $\delta_a \leftarrow \tau.\texttt{rec}_a^{-1}$
47 **foreach** $\tau' \in \tau.\texttt{chd} : \neg\texttt{check}(r, \tau', s + \mu, a)$ **do**
48 $(\delta'_a, tr'') \leftarrow$ *Destroy*$(r, tr', \tau', s, a, \mu)$
49 $\delta \leftarrow \delta \cup \delta'_a$; $tr' \leftarrow tr''$
50 **return** (δ_a, tr')

instance, we find the relations depending on it, bind the mentioned elements in this relation to the new ones, and try to construct new instances from this partial seed. The constructed instances $(tr'' - tr')$ are recorded as the children of τ, so that when τ is not satisfied we can destroy them.

Destroy deletes an existing relation instance whose bound elements no longer satisfy r. We delete this relation instance from the trace, and roll back the recorded change on the system side that has made this instance satisfy the relation. Since this relation is no longer satisfied, the relations depending on it cannot be satisfied any longer, and we delete them consequently.

4.2 Examples

We use a set of simplified examples to illustrate how the algorithm works. The original reflective and abstract models (s_0 and a_0, respectively) are shown in Figure 4, without the shaded part. Currently, these two models are consistent, and the trace is: $tr = \{\tau_1 : \langle \texttt{RR, sr, ar}\rangle, \tau_2 : \langle \texttt{SR, sn, rm}\rangle, \tau_3 : \langle \texttt{RTRP, rd, tg}_1\texttt{, rm, ps}\rangle\}$. For the sake of simplicity, we omit the names of domains. On these two models, we execute the following sample modifications.

For the first example, the sensor detects a change on the brightness, i.e., $\mu_1 : \texttt{set}[sn : \texttt{Sensor}, \texttt{bright}, 620.0, 150.0]$. Since only `SR` contains the class `Sensor`, whereas `bright` is not mentioned by it, the algorithm stops at Line 5.

For the second example, $\mu_2 : \texttt{set}[\texttt{sn} : \texttt{Sensor}, \texttt{temp}, 16, 15]$, the algorithm should propagate the new temperature to the abstract side. Following the al-

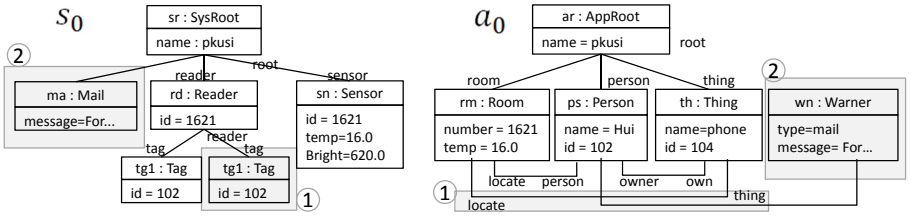

Fig. 4. Sample models for transformation

gorithm, we also find $r = \texttt{SR}$, and *ReEvaluate* this r. At Line 20, we find the relation instance τ_2, and since it does not satisfy the relation, we destroy it. After that, we instantiate a new τ_4 for r (Line 24), bind `sn` to its `sensor` domain, and invoke *Construct*. In this method, since there is one free domain `room`, we try to bind an element to it (Line 45). The `CreateOrUpdate` operation find `rm` and update its `temp` attribute, and thus finally $\delta_a = \mathsf{set}[\texttt{rm} : \texttt{Room}, \texttt{temp}, 16, 15]$.

For a complicated example, we consider the reader detects a new tag, i.e., $\delta_s = \{\mu_3 : \texttt{tg}_2 \leftarrow \mathsf{new}[\texttt{Tag}, \texttt{id}, 104], \mu_4 : \mathsf{set}[\texttt{tg}_3, \texttt{reader}, \bot, \texttt{rd}], \mu_5 : \mathsf{insert}[\texttt{rd}, \texttt{tag}, \texttt{tg}_3]\}$. This time, we expect the thing 104 locate in room 1621. We propagate these modifications one by one. For μ_3, we find two relations, `RTRP` and `RTRT`, but for the former, $\texttt{tg}_2$ does not satisfy its precondition, so we go on with the latter, and invoke *Construct* with $\tau_5 : \{\texttt{relation} \mapsto \texttt{RTPT}, \texttt{tag} \mapsto \texttt{tg}_2\}$ as a seed. In *Construct*, we cannot find any element to be bound to `room` (because $\texttt{tag}_2$.`reader` is not set yet), and stop the propagation on μ_3. When propagating μ_4, we *Construct* τ_5 again, and this time we bind `rd` to `reader`, `rm` to `room`, `th` to `thing`, and update `th.locate` to `rm`. So the final result is $\delta_a = \{\mathsf{set}[\texttt{th}, \texttt{locate}, \bot, \texttt{rm}]\}$ and a new $\tau_5 : \langle \texttt{RTRT,rd,tg}_2\texttt{,rm,th} \rangle$, marked as "1" in Figure 4. If his new tag escapes from `rd`, the abstract model should be rolled back to a_0. This change also contains three modifications, and for the effective one $\mu_7 : \mathsf{set}[\texttt{tg}_2, \texttt{reader}, \texttt{rd}, \bot]$, we destroy the relation instance τ_5 (Line 6 `->` Line 22). And in *Destroy*, we return the inverse of the recorded change under τ_5, i.e., $\{\mathsf{set}[\texttt{th}, \texttt{locate}, \texttt{rm}, \bot\}$.

Finally, we show a bidirectional example, marked as "2" in Figure 4. SmartLab warns a person by creating a new `Warner` and adding it to the person's `warner` list. The last manipulation $\mu_8 : \mathsf{set}[\texttt{wn}, \texttt{warnee}, \bot, \texttt{ps}]$ leads to system changes in the following way. We find the relation `MailWarner` (Line 3), and invoke *ReEvaluate* (Line 6). Since this relation has no instances yet, we directly invoke *Construct* (Line 27), and it finally creates a new `Mail` in the system side and set its attributes. After successfully sending the message, the system destroys this new `Mail`. We finally invoke *Destroy*, and the returned abstract change is the inverse of the recorded modification μ_8, resetting `wn.warnee`.

5 Evaluation

Implementation. We implemented a prototype engine based on the mediniQVT. The relation instances are extended from the `QvtSemanticTasks`, which are orig-

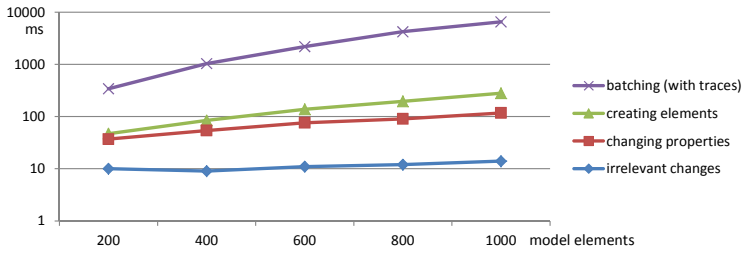

Fig. 5. Performance statistics

inally used by mediniQVT to store intermediate results during batching transformation. The checking, pattern matching, create-or-update operations in our algorithm are also reused and altered from mediniQVT. The syntactical analysis on QVT rules, such as determining the *mentioned* properties of each domain pattern, is implemented as queries and analysis on the QVT syntax tree.

Feasibility and Effectiveness. We applied this instant transformation engine to provide the runtime model for a medium-scale smart office system, the SmartLab. The reflective meta-model contains 27 classes and 69 properties, and the QVT rule contains 36 relations (471 lines in total). We encouraged all the members in our institute to propose and experiment monitoring scenarios based on the abstract model. Until now, there are totally 41 scenarios proposed within the capability of current SmartLab devices, e.g., turning off the lights when the room is empty, turning on the water boiler in advance before a scheduled meeting, warning nearby persons when a valuable public facility is moving, and so on. Our instant transformation supported all these scenarios: A dedicated group of students implemented all the scenarios as QVT operational scripts, and the execution of these scripts satisfies the expectation of both the scenario proposers and the script developers. To evaluate the approach on a wider scope of runtime models, we also applied it on some small-scaled systems to support different runtime models, such as C2 and Client/Server styled architecture for a JEE middleware named JOnAS and a mobile computing middleware named PLASTIC. We have tried these cases [11] using batching transformation. The reproduction of them still satisfies the requirements stated in the original papers.

Performance. The execution performance of our transformation engine is enough for SmartLab. In peak period there are more than 300 model elements, and for each change, the runtime model environment finishes the execution of monitoring rules between 0.1 to 1 second, including the time spent on device invocation, change collection, instant transformation and script execution. This performance is acceptable for our monitoring scenarios on SmartLab. For the other small-scaled cases, the execution time never exceed 0.1 second.

To evaluate the performance of transformation without the influence of other runtime costs, we made up five pairs of models conforming to the meta-models

in SmartLab, and executed the transformation on them. Figure 5 illustrates the experiment results. The horizontal axis lists the total number of model elements, and the vertical axis shows the time spent in millisecond (logarithmic scale). We performed four experiments on each subject. The first three were incremental transformations after the irrelevant changes, changing the properties, and creating new elements. As a contrast, we also executed the batching transformation directly using mediniQVT. All the experiments were executed on a PC with Intel Core 2 Duo 3GHz CPU and 2GB memory. From the curves, we have the following conclusions. 1) the improvement from batching transformation to incremental transformation is significant, and the time increases more gently as the model scale increases. 2) The execution time on irrelevant changes is stable around 10 milliseconds. That means the screening on the relations is independent to the model scales, and adds very little to the total cost. 3) The curve for changing properties is lower and gentler than creating elements. Since the only difference between them is that the former have more fixed domains, this shows that our effort to fix a part of the domain bindings is valuable.

We also performed stress tests to see the extreme change scale and frequency we support, upon the subject models with 1000 elements. For scale, we generate new models and calculate the changes from the original ones to them. When the change contains more than 220 modifications (in average), the time spent to transform these modifications becomes worse than transforming the whole model. For frequency, we continuously generate changes with single modifications, and use them to launch the transformation. The extreme interval between changes is 0.21s. For a smaller interval, there will be a queue of changes blocked.

6 Related Work

Runtime models are widely used on different systems to support self-repair [2], dynamic adaption [4], data manipulation [3], etc. As a direct reflection of the target system, these runtime models are maintained by imperatively mapping the model operations to the system management capabilities. In a previous work [11] we propose the initial idea of using model transformation to maintain the abstract runtime models that are not isomorphic to the low-level systems, but we use batching transformation in that work. Vogel et al. [12] use incremental transformation for runtime models. The difference is that they focus on integrating a general-purpose transformation engine into their runtime model environment, without revising the engine, whereas in this paper, we focus on the semantics and implementation of a new transformation specific to runtime models.

A declarative transformation rule may allow multiple execution effects. The solution is to give unambiguous semantics for transformation languages according to specific usage. Foster et al. formulate three basic properties for the "view-update" transformation between tree-based data [13]. Xiong et al. design and implement their ATL-based model synchronization according to four pre-defined properties [14]. Stevens [9] discusses the semantics of the batching bidirectional QVT transformation. Our properties of instant transformation root in Steven-

s's work, but are defined on model changes. Diskin et al. [15] formally discuss the semantics and requirements of generic delta-based bidirectional transformation, but in this paper, we employ a more lightweight and easy-to-implement semantics, specific to the requirement of runtime models.

Johann and Egyed [16] implement instant and incremental model transformation approach based on the impact analysis of model changes, but the model relation they support is only the simple mapping between elements. On the basis of incremental pattern evaluation [17], researchers also implement incremental transformation following the trigger-action rules [18] and ATL rules [19]. However, such imperative rules are not natural for specifying the relation between runtime models and systems. Giese and Wagner systematically discuss the definition and requirement of instant and bidirectional transformation, and implement it based on their TGG transformation engine [20]. However, TGG is still heavy-weight for specifying model relations. To the best of our knowledge, there is no work of instant and incremental transformation on QVT-R.

7 Conclusion

This paper presents a model-transformation-based approach to maintaining causal connections between the running systems and their abstract runtime models. We define a new incremental transformation semantics for the QVT-Relational language according to the usage in runtime models, and develop the instant transformation algorithm based on the impact analysis of changes. We implement the approach based on the mediniQVT, and apply it in a pragmatic smart office system named SmartLab.

As an initial attempt, the current target of this approach is not a general-purpose incremental QVT transformation, but the one customized for runtime models. The performance of this approach is not good for too big and too frequent changes. However, these two cases are not common in runtime models. We also have some restrictions on the usage of MOF and QVT. For MOF, we require every class to have a key attribute, and require all the multiple properties to be unordered. For QVT, we require 1) every element mentioned by a relation is explicitly declared as a `domain`, 2) all the relations are defined as `top` ones, and 3) only `when` clauses are used to compose relations. According to our experience, with these restricts, it is still enough to specify the relations in runtime models.

Our main future plan is to evaluate the feasibility and effectiveness of this approach on other transformation contexts rather than merely runtime models, improve the semantics and algorithms, and evaluate the possibility towards wide-scope or even general-purpose instant and increment transformation on QVT-R.

ACKNOWLEDGMENT. This work is sponsored by the National Basic Research Program of China (973) under Grant No. 2009CB320703; the National Natural Science Foundation of China under Grant No. 60873060, 60933003, 60821003; the EU FP7 under Grant No. 231167; the Program for New Century

Excellent Talents in University; the National S&T Major Project under Grant No. 2009ZX01043-002-002.

References

1. Blair, G., Bencomo, N., France, R.: Models@ run.time. Computer **42**(10) (2009) 22 –27
2. Sicard, S., Boyer, F., De Palma, N.: Using components for architecture-based management: the self-repair case. In: ICSE. (2008) 101–110
3. MoDisco Project `http://www.eclipse.org/gmt/modisco/`
4. Morin, B., Barais, O., Nain, G., Jézéquel, J.M.: Taming dynamically adaptive systems using models and aspects. In: ICSE. (2009) 122–132
5. Song, H., Xiong, Y., Chauvel, F., Huang, G., Hu, Z., Mei, H.: Generating synchronization engines between running systems and their model-based views. In: MODELS Workshops. (2009) 140–154
6. Song, H., Huang, G., Xiong, Y., Chauvel, F., Sun, Y., Mei, H.: Inferring metamodels for runtime system data from the clients of management APIs. In: MODELS. (2010) 168–182
7. OMG: MOF/QVT model query, view, transformation: `http://www.omg.org/spec/QVT/`
8. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: A cross-discipline perspective. In: ICMT. (2009) 260–283
9. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. In: MoDELS. (2007) 1–15
10. Alanen, M., Porres, I.: Difference and union of models. In: UML. (2003) 2–17
11. Song, H., Huang, G., Chauvel, F., Xiong, Y., Hu, Z., Sun, Y., Mei, H.: Supporting runtime software architecture: A bidirectional-transformation-based approach. Journal of Systems and Software **84**(5) (2011) 711–723
12. Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B.: Incremental Model Synchronization for Efficient Run-Time Monitoring. In: MODELS Workshops. (2009) 124–139
13. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**(3) (2007) 17
14. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: ASE. (2007) 164–173
15. Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F.: From state- to delta-based bidirectional model transformations: The symmetric case. In: MODELS. (2011) accepted
16. Johann, S., Egyed, A.: Instant and incremental transformation of models. In: ASE. (2004) 362–365
17. Cabot, J., Teniente, E.: Incremental evaluation of ocl constraints. In: Advanced Information Systems Engineering, Springer (2006) 81–95
18. Ráth, I., Bergmann, G., Ökrös, A., Varró, D.: Live model transformations driven by incremental pattern matching. (2008) 107–121
19. Jouault, F., Tisi, M.: Towards incremental execution of ATL transformations. In: ICMT. (2010) 123–137
20. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and Systems Modeling **8**(1) (2009) 21–43