Incremental Change Propagation from UML Software Models to LQN Performance Models *

Taghreed Altamimi
Carleton University
1125 Colonel By Drive
Ottawa ON Canada
taghreedaltamimi @sce.carleton.ca

Dorina C. Petriu
Carleton University
1125 Colonel By Drive
Ottawa ON Canada
petriu@sce.carleton.ca

ABSTRACT

Model-Driven Engineering (MDE) [1] enables automatic generation of performance models from software design models by model transformations. The performance models thus obtained are used for performance analysis of software under development. In previous work, we have used a specialized model transformation language, Epsilon ETL, to generate Layered Queueing Network (LQN) performance models from UML software models annotated with the MARTE profile. When the UML model evolves during the development process, the traditional solution for keeping the performance model synchronized is to rerun the entire transformation each time the software model changes. Such a solution is expensive, especially in large-scale models. In this paper, we propose an incremental change propagation (ICP) approach to propagate changes from the UML+MARTE software model to the corresponding LQN model. The entire process starts by automatically generating an LQN model with the previously developed Epsilon ETL transformation. During the development process, when the UML model evolves, we detect the changes with the Eclipse EMF Compare tool, then incrementally propagate them to the LQN model to keep it synchronized. Note that Epsilon does not support incremental model transformation. The proposed ICP is implemented with the Epsilon Object Language (EOL) and it is evaluated by applying it to a set of case studies.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering • Software and its engineering → Software performance**

KEYWORDS

performance model; model transformation; incremental change propagation; synchronization; UML; MARTE; LQN; Epsilon.

*Produces the permission block, and copyright information
†The full version of the author's guide is available as acmart.pdf document
[1]It is a datatype.

WOODSTOCK'97, July 2016, El Paso, Texas USA

DOI: 10.1145/123_4

1 INTRODUCTION

In Model Driven Engineering (MDE) analysis models can be used to evaluate software Non-Functional Properties (NFP) such as performance, reliability, availability, safety, etc. These models can be automatically generated by model transformations from UML software models, which represent different views of the system. The software model evolves during software development in order to meet the functional requirements.

This continuous evolution creates inconsistencies between the software and analysis models. There is an urgent need to support the evolution of higher-level artifacts such as analysis and design models [19] [20]. Model evolution is important for improving the system quality [14] as it provides continuous feedback to the designers. According to [18] incremental approaches minimize the effort to change a small part of the model, as the effort is proportional to the size of the change.

In this paper, we propose an Incremental Change Propagation approach (ICP) to update the affected part of the analysis model when a set of changes is applied to the software model. We are motivated by the research question raised in [22] about how to incrementally propagate the changes between software and analysis models in the context of complex ecosystems containing heterogeneous interrelated modeling artifacts, such as models, metamodels, transformations, solvers and analysis results.

The ICP approach proposed in this paper is applied in a specific context: the source model *S* is a UML software model with MARTE performance annotations (see Fig.1), which is transformed for the first time into a target LQN performance model *P* and a trace model, by using a batch transformation previously developed by the authors in [1]. *P* is solved with an existing LQN solver, obtaining performance results that are fed back to the UML model via a performance analysis roundtrip. The batch transformation is able to generate an entire target model at once from an entire source model, but does not support incremental transformation. During the development process, the following chain of actions is repeated many times: different changes are applied to *S*, manually or automatically, producing a changed model *S'*, which is now out-of-synch with the performance model. We propose here an incremental change propagation approach, which automatically detects the set of changes between *S* and *S'* and propagates them to the target model *P'*, synchronizing it with *S'*. In order to automatically detect the changes between *S* and *S'*, we use an existing Eclipse tool, EMF Compare [2]. The differences detected by EMF Compare, the

LQN model *P* and the trace model become inputs to our ICP approach, which produces a synchronized version of the target model P'. The whole ICP approach is implemented in Epsilon Object Language (EOL) an imperative programming language for creating, querying and modifying EMF (Eclipse Modeling Framework) models [12].

The paper is organized as follows. Section 2 gives an overview of the related work in the research area. Section 3 presents the mapping between the source and target model. Section 4 describes the two phases of the ICP approach: a) change detection, and b) change propagation. Section 5 illustrates the detailed application of the ICP approach to two examples. Section 6 describes how the ICP implementation was validated and Section 7 concludes the paper.

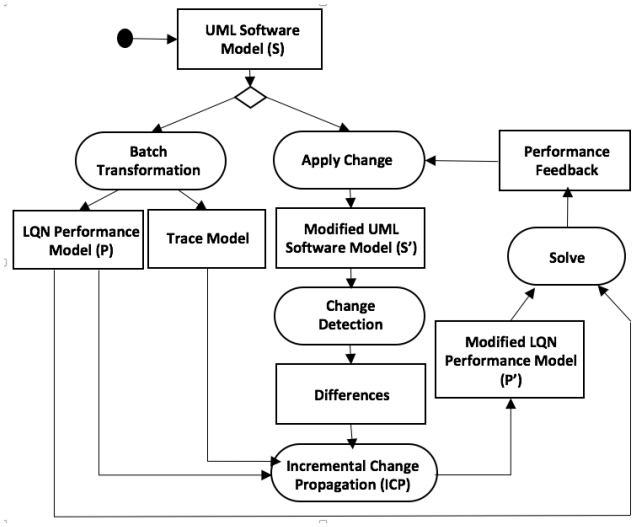


Figure 1. Overview of the proposed ICP approach

2 RELATED WORK

Incremental transformation is becoming a preferred alternative solution to the traditional solution (batch transformation) when the source model evolves during the software development process.

In the literature there are two main approaches for incremental transformation as noted in [10] [23]. The first approach is the *batch transformation*, which depends mainly on rerunning the whole transformation, even though not all parts of the source model have been evolved [8]. Rerunning the whole transformation does not maintain the transformation context, since it becomes unclear which part was changed and which was not. For instance, if we want to compare the performance results of the model before change with the ones after change, it is useful to know which elements have changed and which are the same. This information is clear in the incremental approach only. Also, merging the newly generated model with other related models (e.g., previous analysis results) depends heavily on the trace information generated by the transformation language [5][10]. The second more practical solution is *incremental transformation*. It focuses only on examining the elements of the target model that were affected by changes in the source model and propagating those changes from the source to target model, without consuming time in re-executing the whole transformation. Incremental approaches require less execution time, and thus are more practical and efficient for large-scale systems [20][23] because they avoid unnecessary overhead caused by rerunning the transformation [13] of the whole source model. Hence, incremental approaches are more economical.

A good example of incremental approach can be found in [9], where the authors propose a framework for incremental transformation and apply it for a transformation from UML to ESCM (a modeling language specialized in embedded systems) on the top of IBM Rational Rose. Two algorithms were developed in [9] for identifying the actions that need to be taken to change the target model according to the change in the source model. Our approach is similar in terms of checking the existence of the changed element in the source model, and of the corresponding element in target model. However, [9] does not take into consideration that the changed element in the source model should satisfy some conditions in order to generate a corresponding element in the target model. Our approach verifies such conditions (called guards) before creating a new element in the target model. Another difference is that the mapping between the source elements and the target elements is one to one in [9], while in our case its one to many.

Another approach for incremental transformation called Logic-based SLD Resolution is presented in [5], built in the context of Tefkat transformation language, which can only support declarative transformations. Also, only atomic changes are allowed (element insertion and deletion) [13]. Our approach is built in the context of Epsilon ETL, a hybrid transformation language. It supports both atomic (element insertion, deletion) and composite changes (updating and moving).

The approach proposed in [4] is similar to our approach, in the sense that it supports incremental model synchronization in a uni-directional transformation. It is applied for transforming class models to relational database models. It depends on the old traces to update or delete elements in the target model; for inserting, a knowledge base is used with information about the transformation pattern. Our approach is different, as it supports change propagation between source and target models with different metamodels. The difference in metamodels brings more complexity in interpreting the change in the source model and propagating it to the target model. Another difference is that our approach needs to check some guards in order to decide whether a source element change will lead or not to the creation of a target element. In addition, our approach checks if the propagated change affects internally other elements in the target and changes them, although their source elements did not necessarily change.

A forward and a backward change propagation between the source model and target model are proposed in an ATL transformation [24]. The forward method depends on re-executing the whole batch transformation, while the backward method does not support insertion in the target model.

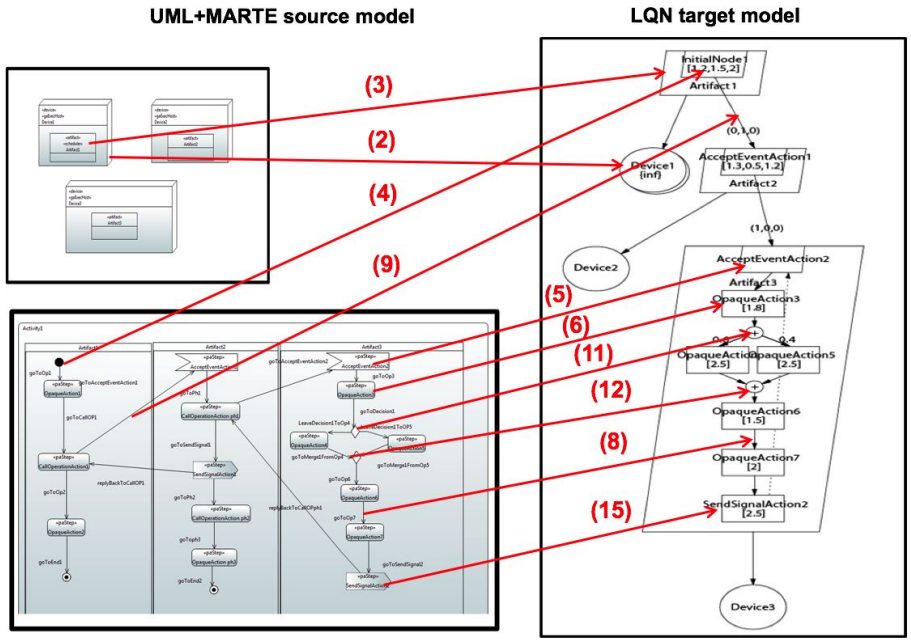


Figure 2. High-level view of mapping between the source and target model

The impact on the system performance model when applying a design pattern to a Service Oriented Architecture (SOA) design model is investigated in [15]. The Role Based Modeling Language (RBML) is used to define the design patterns. The changes produced by the pattern's application are propagated to the performance model. The similarity between [15] and our work is that both use the transformation between software models in UML+MARTE to LQN performance models. The difference is that in [15] only modifications due to design pattern application are propagated to the LQN model, while in our case any modification to the software model can be detected with EMF Compare and propagated to the LQN model.

3 UML+MARTE TO LQN MAPPING

As already mentioned, the context for our ICP is the transformation of a UML+MARTE source model *S*, composed of a set of model elements $E \epsilon S$, into an LQN performance model *P*, composed of a set of model elements $T \epsilon P$. A batch model transformation that transforms *S* into *P* contains a set of transformation rules, each with an optional guard $g(E)$. The transformation engine applies each rule $\mathcal{R}(E)$ to all source element instances of type *E*. If there is a defined guard $g(E)$ that evaluates to true for the *E* instance taken as parameter or the guard $g(E)$ is not defined, the rule will generate one or more target elements instances $\{T_1,...,Tn\}$ and initialize their properties:

if ((g(E).isDefined and g(E)=true) or g(E).isUndefined)
$\mathcal{R}(E) \rightarrow \{T_1,...,Tn\}$

Table 1. Mapping from UML+MARTE to LQN

UML Model Element *E*	Lqn Model Element *T*	Guard *g(E)*
1. Model	Lqnmodel	undefined
2. Device	Processor	undefined
3. Artifact	Task	undefined
4. InitialNode	Entry	isGraphPattern()
5. AcceptEventAction	Entry	isGraphPattern()
6. OpaqueAction	Activity	undefined
7. CallOperationAction	Activity	undefined
8. ControlFlow	Precedence	guardForControlFlow ToPrecedence()
9. ControlFlow	Synchcall	controlflowProcessing forSyncCall()
10. ControlFlow	Asynchcall	controlflowProcessing forASyncCall()
11. DecisionNode	Precedence	undefined
12. MergeNode	Precedence	undefined
13. JoinNode	Precedence	undefined
14. ForkNode	Precedence	undefined
15. SendSignalAction	Activity	guardForSendSignal Action2Activity()

Table 1 represents the mapping between the elements of the UML+MARTE source model and the LQN target model, including the guard functions. This mapping table is used in both the batch transformation from [1], as well as in the incremental change propagation presented in this paper.

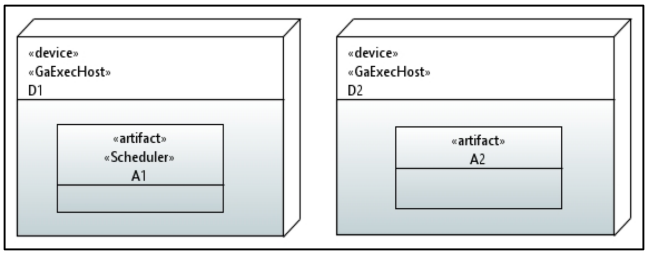
Fig.2 illustrates the high-level view of some of the mapping between the source and target model elements for a simple example, where the source model includes a deployment and a single activity diagram. (In general, a source model can have multiple activity diagrams for different scenarios). Note that Fig.2 is meant to give a bird's eye view of the relationships between model elements, without looking at all the textual details.

More detailed examples of source and target models and their relationships are given in Section 5. The thick red arrows represent the application of some of the mapping rules, numbered with the row number from Table 1. The elements in the UML are annotated with stereotypes from the MARTE profile (especially the performance analysis PAM subprofile) to bridge the gap between the UML and LQN performance models. For more clarification, a device in the deployment diagram has "*GaExecHost*" stereotype to show execution resources. The LQN model on the right of Fig.2 contains three processors (represented as ovals) which are generated from the UML devices from the deployment diagram. On each processor is deployed a software task (represented as a parallelogram) corresponding to the UML artifacts from the deployment diagram. (In general, more tasks can run on each processor). An artifact is annotated with "*Scheduler*" that represents a kind of *ResourceBroker*, which creates access to its brokered *ProcessingResource* or resources, following a certain scheduling policy. Each task has an entry that consists of an LQN activity graph generated from the partition of the UML activity diagram that models the entry behavior. (In general, a task may have multiple entries corresponding to different services offered by the task).

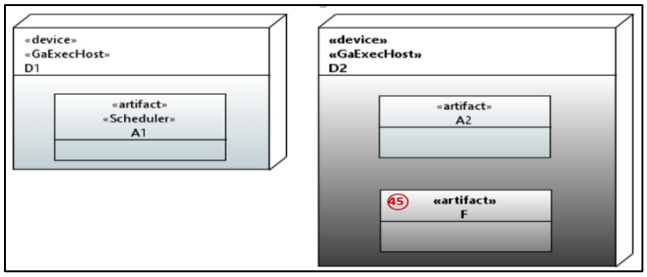
In the activity diagram *AcceptActionElement*, *Initial Node*, *OpaqueAction*, *CallOperation Action* and *SendSignalAction* are annotated with *<<PaStep>>* stereotype that is a type of *<<GaStep>>* and can inherits its properties; *<<PaStep>>* can be applied to UML actions or messages to indicate behavior steps. *OpaqueAction*, *CallOperationAction* and *SendSignalAction* are mapped to LQN activity element. *AcceptEventAction* and *InitialNode* are mapped to LQN entry element. In LQN there are two types of entries: a) phase-based entry composed of a sequence of one to three activities (called PH1PH2 type); and b) graph-based entry composed of a graph with branch/merge, split/join, etc. (GRAPH type). In Fig.2, the top two tasks have each a phase-based entry, while the bottom task has a graph-type entry. An entry can send a synchronous or an asynchronous call to the entry of another task. Such calls are generated from cross-border edges in the activity diagram (e.g., see the red arrow (9) from Fig.2). The rest of the elements in the activity diagram, such as *DecisionNode*, *MergeNode*, *JoinNode* and *ForkNode* are mapped to precedence.

4 INCREMENTAL CHANGE PROPAGATION APPROACH

This section describes the proposed Incremental Change Propagation (ICP) from the modified UML+MARTE source model to the *Lqn* target model. Then we will illustrate in more

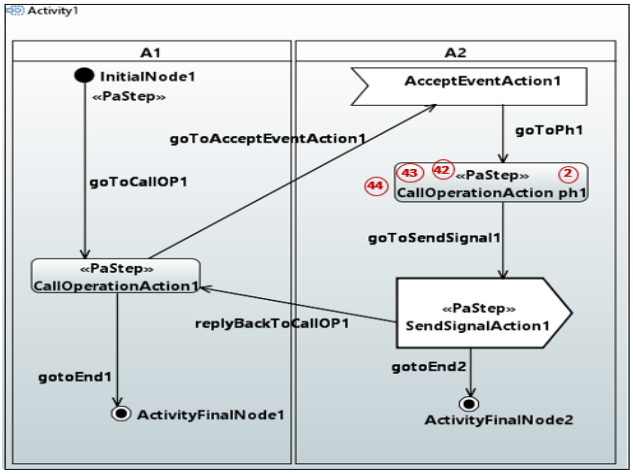


a. Example1: Original Deployment Diagram

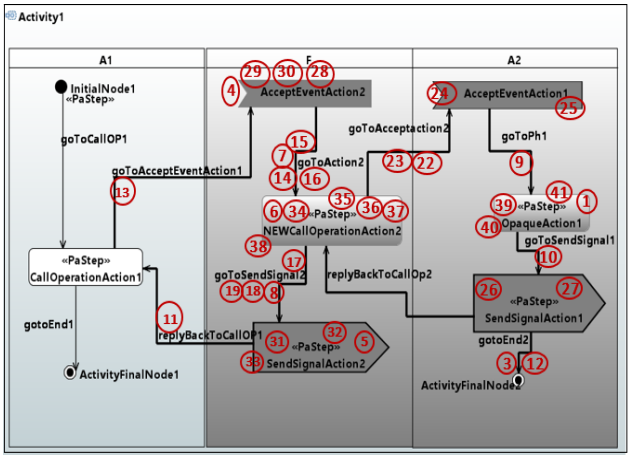


b. Example1: changed deployment diagram

Figure 3. Deployment Diagram for Example 1



a. Example1: original activity diagram



b. Example1: changed activity diagram

Figure 4. Activity diagram for Example1

details the ICP approach by applying it to two examples in section 4. The Epsilon Object Language (EOL) is used to implement the proposed ICP approach. The following files are used as ICP input: two versions of the source model (original and changed), *Lqn* target model corresponding to the original source, the trace model corresponding to the original transformation and the *Differences* file. The result of running the ICP is the synchronized *Lqn* model, with all the propagated changes.

4.1 Change Detection

The change detection compares directly two versions, *S* and *S'*, of the same UML+MARTE model. *S* the source model for the batch transformation from [1]. There are two approaches to detect the changes according to [7] [16] [17]: a) operator-based approach detecting the changes as a set of operations [6]; or b) direct comparison between two versions of the same model, which gives better results when comparing models containing elements with unique identifiers (such as UML) [3]. We use EMF Compare [2] to obtain the differences between *S* and *S*'. According to [11] using Static Identity-Based Matching approach (where every model element has a unique identifier) can be faster and user independent (i.e., no configuration is needed from the user).

The result of the comparison between S and S' is saved and queried as an XMI file called *Differences*. The root node of its schema is *Comparison*, a metaclass in the EMF Compare metamodel that has all comparison information such as matched resources, matched objects and detected differences [2]. The root can have many *match* children, and each *match* can have many submatches. A submatch has a *left* and *right* node representing the matched resources and zero to many differences. The differences are classified as RC, AC or RAC. RC (*ReferenceChange*) is detected when a reference value is changed (i.e., added, moved or deleted). AC (*AttributeChange*) is similar to RC, but it refers to an attribute rather than a reference. RAC (*ResourceAttachment Change*) is detected when one of the root of the matched resources changes. In a RC case, a difference has two children: a reference of the changed object or the attribute and its value. In an *AC* case, the difference has an attribute instead of a reference. Each difference has a kind that can be *ADD*, *CHANGE*, *DELETE* or *MOVE*. ADD includes two cases: a) adding a new element within the values of a multi-valued feature; or b) any change in a containment reference, even if that reference is mono-valued, represents a "new" element in the model. In the CHANGE case, the engine considers any modification to a mono-valued feature as CHANGE and excludes the containment references from this rule. DELETE follows the same logic as ADD, considering that a change to containment reference is deleting, even if that reference is a mono-value. A change is considered as a MOVE in two cases: a) moving an object from one container to another; and b) reordering the values of a multi-valued feature [2].

4.2 Notation

In this section, we introduce some notation used in the paper.

- *S* is the original source model, which consist of original elements *E* denoted by *S!E.*
- *S'* is the changed source model, after applying a set of changes to *S*. *S'* consists of elements *E'* denoted by *S'!E'*. The set of differences between *E* and *E'* is denoted by diff, where *E'=E+diff*.
- Every *E* in *S* has set of original properties p denoted by *E.p*. If *p* changes, then *p'* represents the changed property set *E'.p'*

EMF Compare matches pairs of corresponding elements that are different in *S* and *S'*. Possible matches *{E, null},{null, E'},{E, E'}* have the following meaning:

- *{E, null}: E* exists in *S* but *E*' does not exists in *S*', which means that *E* is an old element deleted from the model.
- *{null, E'}: E* does not exist in *S*, but *E'* exists in *S'*,which means that *E*' is a new element added to S'.
- *{E, E'}: E* exists in *S* and *E*' exists in *S'*, which means that the old element E ϵ S has been changed to *E'* ϵ *S'*.

Other notations used in the paper are:

- The result of comparing the two sides of a match results in a set of differences *diff*. Each difference has a *type* ϵ *{RC, AC, RAC}*, where: *RC* is *ReferenceChange, AC* is *AttributeChange* and *RAC* is *ResourceAttachmentChange*.
- The change actions applied to *S* can be of different kinds, where *kind* ϵ *{ADD, DELETE, CHANGE, MOVE}*.
- *DiffVal* is the value of a changed model element or property.
- *DiffRef* is a reference name if *diff.type = RC*.
- *DiffAttr* is an attribute name if *diff.type =AC*.

4.3 Change Propagation

The change propagation has six major steps applied repeatedly for every match:

Step 1: *Checking the existence of E in S and E' in S'*. As mentioned above, there are three cases denoted as *{E, null},{null, E'},{E, E'}*.

Examples of such cases are shown in detail in Section 5. For instance the activity partition *A2* is an old element in S (see Fig.4.a) that is changed in *S'* (see Fig. 4.b). On the other hand, the activity partition *F* does not exist in *S* (Fig. 4.a) but it was added to *S'* (Fig.4.b). The activity *CallOperationAction ph1* exists in *S,* but it is deleted from *S'.* Note that the changes in UML diagrams are shaded in darker grey.

Step 2: *Matching the UML elements with the same identifier in both UML model versions S, S' to get the name and type of E.* The matching is done by operation *getName (id: String)* that receives the identifier of *E* as a parameter and returns its name. Similarly, we get the type of *E* by executing operation *getType (id :String)*.

Step 3: *Getting all differences between E and E'*. Each match between E and E' can have one or more differences. Therefore, we need to iterate among the differences to get the *type*, *kind* and *value*. For example, after identifying that A2 is an original element in step1 and getting its *type* in step2, we get all its differences in step 3. A2 has three differences, each needing a specific action, as shown in the next section, Example 1, match A.

Step4: *Getting and checking the trace of E.* There are three cases related to the existence of *E* and *E'* in *S* and *S',* respectively:

If E and E' both exists, we get their differences and check if the target element(s) of *E* exists in *Lqn* by checking if it has a trace or not. If *E* has a trace (i.e., its target element exists in Lqn) we can update it according to the differences. If *E* does not have a trace, then we need to get its target element type from the mapping table (see Table 1). The creation of a new target element depends on whether there is a guard *g(E)* or not; if the guard exists and it is satisfied, then we can create the target element $T = \mathcal{R}(E)$ in Lqn. For example, in match C, Example2, *SendSignalAction1* is an old element of type *SendSignalAction*. It is changed in S' and it does not have a trace. Fom the mapping table we find that its type has a guard and should mapped to activity in Lqn. After verifying that the guard, the activity *SendSignalAction1* is created in *Lqn*.

If *E'* exists in *S'* but *E* does not exist in *S*, this means that *E'* is a new element introduced in *S'*. Its corresponding target element $T' = \mathcal{R}(E')$ needs to be created in *Lqn,* following the same reasoning as above. Later we will discuss how can we add *T'* to its correct location by adding it to the right container. If *E* exists in *S* but *E'* doesn't exists in *S'*, then *S* was changed by deleting *E* from it. Consequently, we need to delete the target element $T = \mathcal{R}(E)$, corresponding to *E* from *Lqn* by following the trace.

Step 5*: Taking the corresponding action for each type of difference when matching E with E'.*

Case *kind* = *ADD, DELETE, or MOVE*:

a. Get the type for difference value by invoking *getType()*. From the mapping, we know if difference value is an element.
b. If difference value is an element, then check whether a corresponding target element exits in *Lqn,* to make sure that the element was not already created while checking other matches.
c. Create, and then add difference value to its container. The container can be either the target element corresponding to E' or we need to get it from the metamodel. Example 1 and Example 2 has different examples of kind ADD and will be discussed in more detail in section 4.
d. In case of DELETE, we follow the same previous steps then take an action to delete the target element corresponding to *E* from *Lqn*. See match A from example 1, difference 2.
e. In case of MOVE, we follow the same steps as above, then take the action to move $T' = \mathcal{R}(E')$, the target element corresponding to *E'* to its new container, which is *T'*.

Case *kind* = *CHANGE*: follow the same steps as for ADD.

Step 6: Update the trace model that was generated previously by the batch transformation by adding a new trace when creating a new element in the target model or deleting the old trace that corresponding to each deleted element in the target model.

5 EXAMPLES

In this section, we describe the proposed ICP approach in more details by presenting two model examples with different kinds of changes. Example 1 introduces a new artifact, as shown in Figs. 3 and 4, while Example 2 introduces a Graph Pattern partition change (see Figs. 10 and 11). We also discuss the actions needed to incrementally propagate each change to the target model.

5.1 Example1

Example1 is a simple source model represented in Figs. 3, and 4. We started by saving *S* as the original model, then applied different changes (highlighted in dark grey color) and saved it as *S'*. Figs. 3.a and 4.a represent the structural and behavioral views of Example1 old model, while Fig. 3.b and 4.b represents the changed model *S'*. The differences detected by the EMF Compare are marked on the diagram with red numbered circles.

The original model *S* was transformed to the LQN model shown in Fig.5 by applying the batch transformation presented in [1].

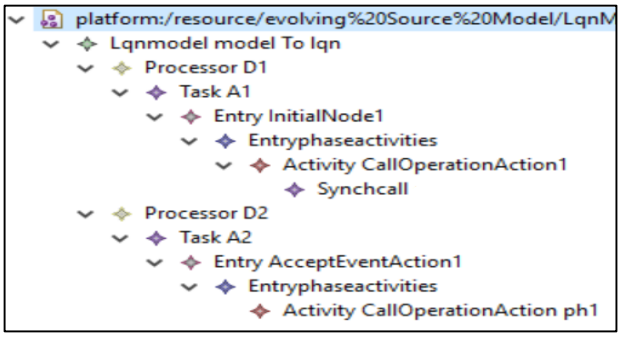


Figure 5. Original LQN for Example 1

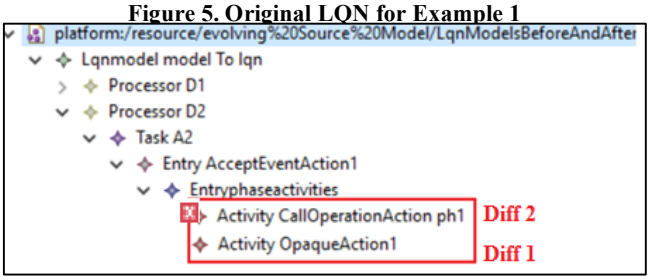


Figure 6. Example1: LQN after propagating diff 1 and 2

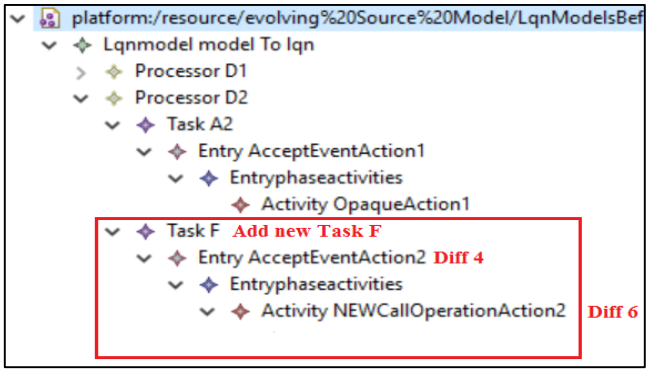


Figure 7. Example1: LQN after creating new Task F and propagating differences 4 and 6

The root element of type *LQNmodel* contains two processors *D1* and *D2*, and each processor contains a deployed task, *A1* and *A2* respectively. Each task has a phase-based entry, whose activities are contained in an element of type *Entryphaseactivities*. The entry of task *A1* contains an activity called *CallOperationAction1*, which sends a *Synchcall* to the entry of *A2* and waits for a reply. The entry of A2 contains an activity *CallOperationAction ph1*.

The entry ends by sending a reply back to the caller. While the reply is not explicitly modeled in LQN, it is implied by the *Synchcall* semantics. However, the UML activity diagram from Fig.4.a that models the behaviour of the entry of task *A2,* contains a *SendSignalAction1* that sends the reply back via a cross-border edge.

EMF Compare matches each element in *S* with each element in *S'* starting from the root. When the two elements are identical, EMF Compare does not detect any differences. We discuss only the matches that generate differences. They are in the same order as generated by EMF Compare.

A. Change activity partition *A2*: Differences in the form *{type, kind, DiffRef, DiffVal}* when matching *S!A2* with *S'!A2'*:

1. *{RC, ADD, node, OpaqueAction1}*
2. *{RC, DELETE, node, CallOperationAction ph1}*
3. *{RC, ADD, edge, goToEnd2}*.

Applying the previous steps from 1 to 4 on the changed activity partition *A2'*, where *A2* is an old element. As we discussed before, the existence of the element in UML does not mean that the element exists in *Lqn*. It depends if the element type has a guard or not and if that guard has been met or not. As a result, from applying step 4, we know that *A2* exits in *Lqn.* Following the transformation assumptions which states that the name attribute of each *ActivityPartition* should be identical to the name attribute of its corresponding *Artifact*, we can get the target type for A2 by invoking *getTargetType (ElementName)* operation which returns type *Task*. Then we can get the target name by executing *getTargetName (ElementName)* operation. After getting the task name, we are able to know the changes suffered by task A2. The next to step is to *update task A2* according the above differences (see step5). An update is a composite change that is performed as a set of operations on *A2*.

Action for Diff 1: we get the type of OpaqueAction1 with *getType* operation, which returns *OpaqueAction* type. Then *getTargetType* operation checks the mapping table and returns the activity type as *OpaqueAction* mapped to *activity* in Lqn.

According to the LQN metamodel, task is not a direct container of activity, but it is a container of *task-activities* if entry type is *GRAPH.* On the other hand, *entry* is a container of *entry-phase-activities*, if entry type is *PH1PH2*.The entry *type* identifies the container type for the collection of activities modeling the entry behaviour . In order to add the new activity *OpaqueAction1*, we need two steps: first create the activity and next find the container for the new activity. In the first step, we have to check if type *OpaqueAction* has a guard that needs to be satisfied and if the activity *OpaqueAction1* exits or not in the *Lqn* (to double check whether *OpaqueAction1* was created when matching other differences). In this case *OpaqueAction* does not have a guard, there is no trace, and *kind* =ADD, so we can create a new LQN activity and initialize its name to *OpaueAction2.* This is done by invoking operation *addChildToTask* with parameter *E',* whose type is *task*, name is *A2* and *DiffVal* is *OpaqueAction1* of type *activity*. Inside the operation, we get the entry of *A2* and check its type by invoking *isGRAPHPattern* operation that returns *true* if entry is of type *GRAPH* and *false* if entry is of type *PH1PH2*. Some changes

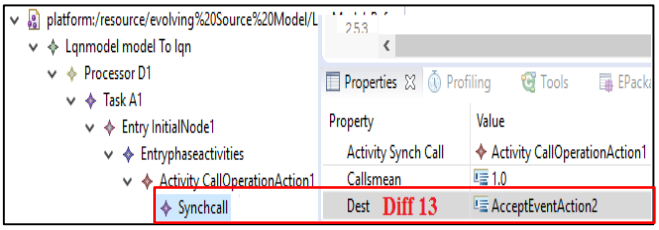


Figure 8. Example1: LQN after propagating difference 13

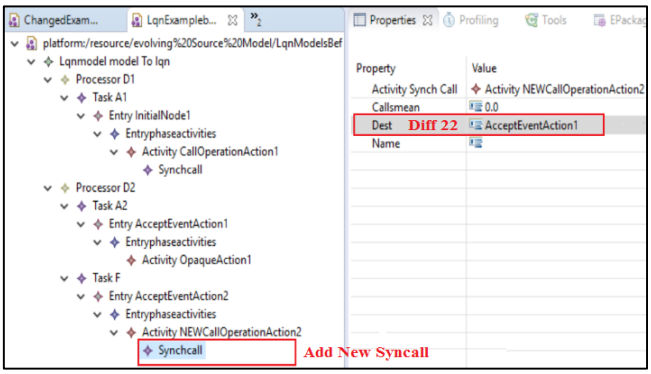


Figure 9. Example1: LQN after adding new Synchcall and propagating difference 22

to activity partition, for instance adding control nodes of type *decision* and *merge*, may affect indirectly the evaluation of *isGRAPHPattern* operation, which in turn may affect the entry type, even if the source element for that entry did not change. For that reason, invoke *isGRAPHPattern* inside *addChildToTask* operation and update the type of entry according to its returned value. Later, we will see such a case in Example 2. If the entry type is *PH1PH2*, then the container of the new activity is an *entry-phase-activities* block. We check if this block has already been created in *Lqn*. If yes, then it already has a collection of activities to which we can add the new activity. On the other hand, if *entry-phase-activities* is not defined, we need to create it, then add the new activity to it. Finally, we need to update the trace file by *addTrace* operation, in order to add a new trace for *OpaqueAction1* activity. In case of entry type *GRAPH,* we followed the same previous steps and add the new activity to *task-activities* block (if it already exist) or create a new *task-activities* block and add the new activity to it.

Action for Diff 2: *type=RC*, *kind=DELETE*: we deleted *DiffRef* node *CallOperationAction ph1*. From the above discussion, we know that *A2* is a task in *Lqn*, so we need to know the type of the deleted element from *S* by invoking *getType*, which returns *CallOperationAction*. The next step is to check if *CallOperationAction ph1* exists it in the *Lqn* model. If it does exist, we can delete it by *deleteActivity* operation. It takes *DiffVal* as a parameter and matches it with Lqn activity then deletes it. Finally, the trace is updated by deleting *CallOperationAction ph1* trace. Fig.6 shows the *Lqn* model after propagating the differences 1 and 2.

Action for Diff 3: adding a new reference edge to *A2* (the *DiffVal* is *gotoEnd2)*. Its action has the same steps as the previous ones,

except that type *ControlFlow* has three guards (see Table 1) so we need to check which guard is *true*. In this case, control flow *goToEnd2* does not satisfy any of the three guards, so no action is needed on the Lqn side.

B. Add new partition F: Differences in the form *{type, kind, DiffRef, DiffVal}* obtained by matching *S'!F'*:

4. *{RC, ADD, node, AcceptEventAction2}*
5. *{RC, ADD, node, SendSignalAction2}*
6. *{RC, ADD, node, NEWCallOperationAction2}*
7. *{RC, ADD, node, goToAction2}*
8. *{RC, ADD, node, goToSendSignal2}*

F' is a new *ActivityPartition* introduced in *S',* which did not exist before in *S*. We create the corresponding target element for F in *Lqn* as follows. First, we invoke *getType* (F') that returns *Activity Partition*. As mentioned in [1], there is an assumption that an *ActivityPartition* should have the same name as the *Artifact* whose behaviour is represented by the activities in that partition. Using this assumption, in the UML model the names of the artifacts from the deployment diagram appears as names of activity partitions in the activity diagram. We get the target type from the mapping table by *getTargetType* operation, which returns *task* in this case. Second, we need to identify the location in *Lqn* where the new task *F* should be added (i.e., the container of *F*). In order to do so, we identify the container of the UML artifact F with the *getNamespaceForArtifact* operation, which returns in this case device *D2* [21]. In the trace file, the target element D2 of type *Processor* corresponds to the source element *D2* of type *Device,* therefore the LQN processor *D2* is the container for task *F*. Finally, we add task *F* as a child to processor *D2* and update the trace by adding a new trace for task F. After adding F as a task to *Lqn*, we can take actions for its differences.

Action for Diff 4: Add *AcceptEventAction2* to task *F*. First invoke *getType* for *AcceptEventAction2* that returns *AcceptEventAction* type. From the mapping table, we get *entry* that is the target type for *AcceptEventAction* type. According to LQN metamodel, *task* is the container of *entry*. Next, we check if *AcceptEventAction* type has a guard, which is *isGRAPHPattern* that evaluates to true. A trace exists for *AcceptEventAction2* which returns *false*. Since *kind =ADD*, we create a new entry *AcceptEventAction2* in *Lqn* by invoking *addChildToTask* operation with parameter E', with type *task*, name *F*, *DiffVal* name *AcceptEventAction2* and its type is *entry*. The new entry is added to the containment reference from task to entry. The new entry is of type is PH1PH2 according to the operation isGRAPHPattern. Finally, we create a trace for *AcceptEventAction2*. Fig.7 shows the LQN model after adding F and propagating differences 4 and 6.

Action for Diff 5: Adding the target element corresponding to the source element *SendSignalAction2* of type *SendSignalAction* as an activity to task F follows the same steps as for adding *AcceptEventAction2* entry, except that *SendSignalAction* type is mapped to an LQN *Activity* and has a different guard named *guardForSendSignalAction2Activity*. The guard should be satisfied in order to create its corresponding target element. After executing *guardForSendSignalAction2Activity* in this case, it returns false, therefore no corresponding action is needed to change *Lqn*.

Action for Diff 6: To add *NEWCallOperationAction2* to task *F* we need to follow the same steps as for adding *OpaqueAction1* to task *A2* (see diff. 1). *NEWCallOperationAction* and *OpaqueAction1* types are similar, as both are mapped to type *Activity* in LQN and neither has a guard. As discussed before, we need to get the container for *NEWCallOperationAction* type and add the corresponding activity to the containment reference from *activity* to *entry-phase-activities* in the context of task F. Operation *addChildToTask* is taking a parameter type *task*, its name *F* and *DiffVal* name *NEWCallOperationAction2* and its type *Activity*. In the previous difference, we added a new entry *AcceptEventAction2* of *type PH1PH2,* then we had to get *entry-phase-activities* (if defined), otherwise we had to create a new collection of them. The last step is to add activity *NEWCallOperationAction2* to its container (*entry-phase-activities*) and update the trace file by creating a new trace for *NEWCallOperationAction2* activity. See Fig.7 for the modified LQN model.

Actions for Diff 7 and 8: are similar to Diff 3, as *goToAction2* and *goToSendSignal2* are both *ControlFlow* typed and none of them satisfies any of the control flow guards. Therefore, no action is needed to change *Lqn*.

C. ControlFlow Elements without target: Differences (9-12) result from matching the elements *(goToPh1, goToSend-Signal1, replyBackToCallOP1, gotoEnd2)* in both *S* and *S'*.

We discuss the above matches together as they are very similar. For all of them, both compared sides *E* and *E'* exists in *S* and *S'*. The *E type* returned by *getType* for all of them is *ControlFlow* and the operation checking the traces returns no trace for each one. This means that none of them has a target element in Lqn. As we mentioned before in Diff 3,7 and 8, *ControlFlow* type in the mapping has three target types *Synchcall*, *Asynchcall* and *Precedence*. Each alternative rule has a different guard. One of them needs to be satisfied in order to identify the target element type for each control flow. None of the above control flows satisfies any of the three guards, and then we do not need to take an action to change the *Lqn*.

D. ControlFlow generating Synchcall: Difference in the form *{type, kind, DiffRef, DiffVal}* obtained by matching *goToAcceptEventAction1* in *S* and *S'*.

13. *{RC, CHANGE, target, AcceptEventAction2}*

Action for Diff 13: Although this match is similar to the previous ones as both compared sides E (*goToAcceptEventAction1*) and E'(*goToAcceptEventAction1'*) exits in *S, S'* respectively, but the trace checking operation returns true. Therefore, it was easy to identify its target element *synchcall* from the trace file. Since we have all information we need, we can execute *updateAttribute (DiffVal)* operation that takes the *DiffVal* and control flow name as parameters, then updates *synchcall* and its attributes. Fig.8 shows the changed LQN after propagating difference 13.

E. New ControlFlow Elements without target. Differences (14-21) result from matching new elements (*goToAction2, goToSendSignal2, replyBackToCallOP2*) that exit in *S'* but do not exist in *S*. This case is similar to C Differences (9-12) where there is no need for any action to change the *Lqn*.

F. New ControlFlow generating Synchcall: Differences in the form *{type, kind, DiffRef, DiffVal}* obtained by matching *S'! goToAcceptAction2'*.

22. *{RC, CHANGE, target, AcceptEventAction1}*
23. *{RC, CHANGE, source, NewCallOperationAction2}*.

The control flow element *goToAcceptAction2* is a new element that exists in S' but not in S. Since its trace checking operation returns false, we need to get its type in order to take the necessary action to change the *Lqn*. Operation *getType* returns *ControlFlow*. Next, we get its target type from the mapping and its guards. Since the guard *controlflowProcessingforSyncCall* determines that the control flow parameter *goToacceptAction2* follows the synchronous call pattern, we have to take an action to create its target element and to add it to its correct container in *Lqn*. Operation *addSynchcall* takes control flow's source and target as parameters, creates a new *Synchcall* target element and assigns its property *dest*. The source name represents the activity name which is the container of *synchcall* according to the LQN metamodel, so we add the new element to its container and update the traces. See Fig.9 for the modified LQN.

The next step is to check the differences (22, 23). Action for diff 22 is similar to the action for diff 13. Diff 23 does not need an action, as *source* property is not mapped to any property in *Lqn*.

We do not discuss other matches because of the limited space and the fact that their differences do not need any action to propagate the change to *Lqn*. Some of them do not meet the guard conditions like *SendSignalAction2* and *SendSignalAction1*. Other matches have differences on properties that are not mapped to LQN, such as the *ControlFlow* properties *outgoing, incoming* and *inPartition*.

G. Container change: Differences in the form *{type, kind, DiffRef, DiffVal}* obtained by matching *S!D2* and *S'!D2'*:

45. *{RC, ADD, nestedClassifier, F}*.

This match has two sides *E'(D2')* and *E(D2)*. *D2* has changed when a new artifact *F* was added to its *nestedClassifier* reference. Note that processor is the container of task in LQN metamodel. *D2* is of type *Device* and is an old element that exists in both *S* and *S'*. Next, we have to check the existence of its corresponding target element in *Lqn*. The trace checking operation returns true. Since *D2* has a trace, we can get its target element and target type from the trace. *D2*'s target element is *processor* and its name is *D2*. Next, we check if *F* has a trace or not. The trace checking operation returns true, as F has been added during the matching of F, based on the previously mentioned assumption about using the same name for artifact and activity partition. Therefore, we do not need to change processor *D2* in *Lqn*. If *F* was not added before, we would get its target type (i.e., task) from the mapping and then invoke *addChild* operation to add task *F* to processor *D2*.

5.2 Example 2

Example 2 illustrates other differences (appearing in dark grey color) that lead to other Lqn changes not shown in Example 1. An

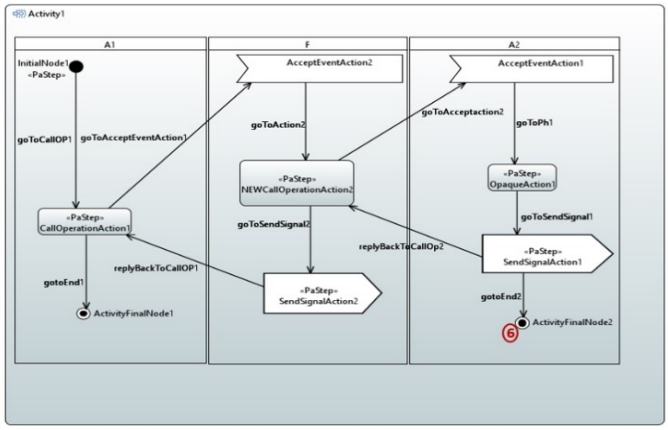


Figure 10. Example2: activity partitions without graph pattern

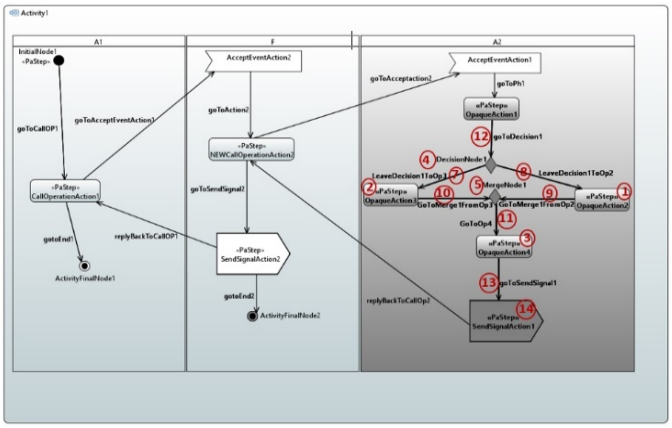


Figure 11. Changed Example2: activity partition A2 with graph pattern

interesting observation is that some elements in Lqn need to be checked even though their corresponding source element in UML has not been changed.

The deployment diagram for example 2 is the same as in Fig. 3.b. Fig.10 shows the original activity diagram of Example 2, where none of the three activity partitions has the activities organized in a Graph Pattern, and Fig.11 shows a changed activity diagram with activity partition *A2* as a Graph Pattern. Please note that the changes to partition *A2*, (marked as 12, 13 and 14) are changes to the properties of the old elements that already exist in *A2*.

A. Changes inside a partition: Differences in the form *{type, kind, DiffRef, DiffVal}* by matching *S !A2* and *S'!A2'*:

1. *{RC, ADD, node, OpaqueAction2}*
2. *{RC, ADD, node, OpaqueAction3}*
3. *{RC, ADD, node, OpaqueAction4}*
4. *{RC, ADD, node, DecisionNode1}*
5. *{RC, ADD, node, MergeNode1}*
6. *{RC, DELETE, node, ActivityFinalNode2}*
7. *{RC, ADD, edge, LeaveDecision1ToOp3}*
8. *{RC, ADD, edge, LeaveDecision1ToOp2}*
9. *{RC, ADD, edge, its DiffVal is GoToMerge1FromOp2.*
10. *{RC, ADD, edge, GoToMerge1FromOp3}*
11. *{RC, ADD, edge, GoToOp4}*
12. *{RC, ADD, edge, goToDecision1}*.

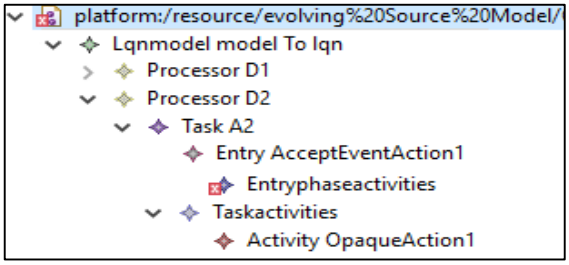


Figure 12. Intermediate step for adding Taskactivities

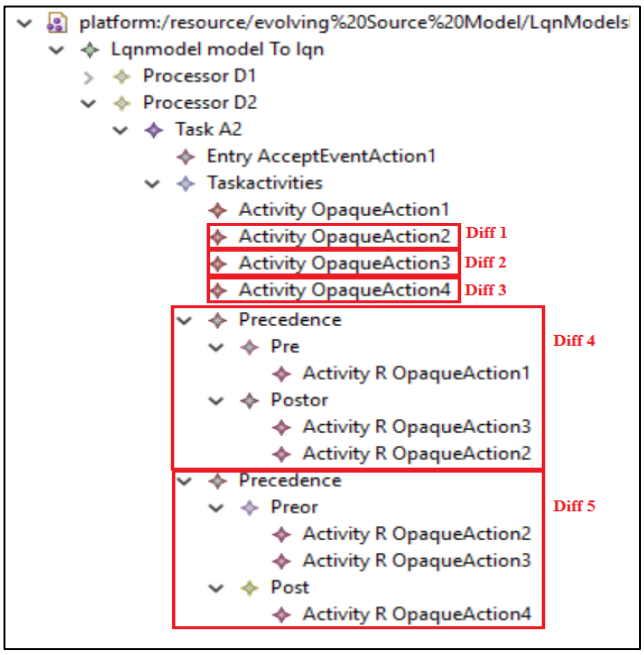


Figure 13. Example 2: Lqn after Propagating Differences 1 through 5

Action for Diffs 1, 2, 3 are similar to difference 1 in Example 1, as *A2* exists in both sides of the match and it has a trace, which means its corresponding target exists in *Lqn*. The target element for *OpaqueAction* type is LQN *Activity*; we need to create a target element for each source element *OpaqueAction2, OpaqueAction3* and *OpaqueAction4*. Task *A2* has entry *AcceptEventAction1*, of type *PH1PH2*, so it has an *entry-phase-activities* block as container for the collection of activities. As shown in Example1, diff 1, some changes, like adding new control nodes of type decision or merge can affect indirectly the type of entry. For that reason, operation *addChildToTask* invokes *isGRAPHPattern* operation in order to check the entry type. In this case, the type of entry should be updated to type GRAPH, and a *task-activities* block should be created instead of the existing *entry-phase-activities* block. Since *A2* has already a collection of activities, we need to create a new *task-activities* block and move that collection to it. According to the LQN metamodel, a *task-activities* block and entry-phase-activity block cannot exist at the same time, so we delete the *entry-phase-activities* block. Fig.12 represents an intermediate step for deleting *entry-phase-activities* and creating *task-activities*. As a last step, a new activity with the same name is created for each of the source elements *OpaqueAction2*, *OpaqueAction3* and *OpaqueAction4*. The new activities will be added to the already existing collection of activities.

Action for Diff 4: the difference here is adding a new node with *DiffVal* being *DecisionNode1*.To take an action for this difference, we get the type of the new node (that is *DecisionNode*) and invoke the trace checking operation that returns false, which means it does not exist in *Lqn*. In order to propagate the difference to *Lqn* we invoke *addChildToTask* operation, which in turn invokes *createPrePostorDecisionNode* operation. This creates the target elements for *DecisionNode1* in the context of *A2* task which is the container of *task-activities* block that in turn is the container of *Precedence,* the target type for *DecisionNode* type .

The mapping here between the source type, *DecisionNode*, and its target type, *Precedence*, is not one to one. Instead, each *DecisionNode* in *S* has three target elements in Lqn: *precedence*, *pre* and *postor* elements. The last two need to be created as children of their *precedence* container. Each *pre* and *post* elements contain elements of type *ActivityR*, which refer to the interconnected activities. For more details, see [1]. Finally, we update the trace.

Action for Diff 5: This difference is similar to diff 4, as *MergeNode1* needs to be added as a *precedence* in *Lqn*. However, the *precedence* has *post* and *preor* as children, so *addChildToTask* operation invokes *createPreorPostMergeNode* that takes the *mergeNode* and task *A2* as parameters and creates a *precedence* element and its children. Next, add the corresponding trace to update the trace file. Figure 14 represents *Lqn* after propagating diffs 1 through 5.

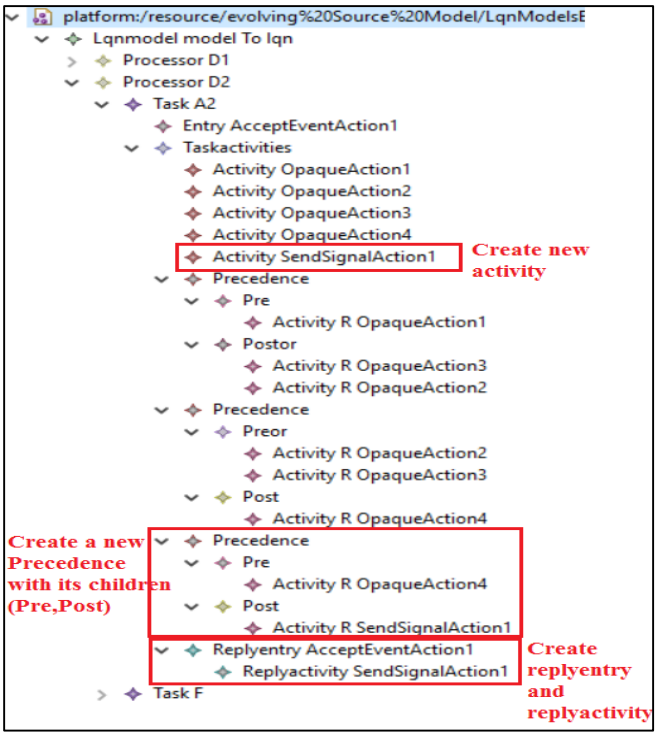


Figure 14. Example2: LQN after adding a new precedence and its children (Pre, Post), Replyentry and Replyactivity

Differences from 6 to 12, do not need any action for changing *Lqn*. *DiffVal* for diff. 6 is of type *Activity FinalNode* that is not mapped to any type in *Lqn*. Regarding the differences from 7 to 12, all of them are of type control flow and none satisfies any of the three possible guards.

A. ControlFlow generating precedence: Differences in the form *{type, kind, DiffRef, DiffVal}* obtained by matching *S!goToSendSignal1, S'! goToSendSignal1':*

13. {RC, ADD, source, OpaqueAction4}.

As discussed before, *goToSendSignal1* of type *ControlFlow* exists in *S* and has been changed in *S'*. However, it does not exist in *Lqn* as it has no trace. From the mapping, we know that control flow has three guards. After testing each of them, only *guardForControlFlowToPrecedence* returns true. Now we are able to create a *precedence* and its two children *pre* and *post* in *Lqn* as target elements. The action for propagating this difference is done by invoking *createPrePostControlFlow* that is invoked through *addChildToTask*.

After creating the precedence for *goToSendSignal1* control flow (see Fig.14), we can check the generated difference on its property *source*. Since *source* is a property not mapped mapped to any property of *precedence* in *Lqn* , there is no need for any action.

B. SendSignalAction generating activity: Differences *{type, kind, DiffRef, DiffVal}* obtained by matching S! SendSignalAction1 and S'! SendSignalAction1':

14. *{RC, ADD, outgoing, goToEnd2}*.

Although *SendSignalAction1* is an old element that exists in both *S* and *S'*, its target element does not exist in *Lqn* as there is no trace corresponding to it. Since *SendSignalAction* type has a guard, we have to evaluate it because changes occurred in the element itself or elsewhere in its activity partition might affect the evaluation of the guard. The respective guard takes the property *incoming* of *goToSendSignal1* as a parameter and checks two conditions: a) if the respective control flow is located inside the activity partition and b) its target element *SendSignalAction1* is not part of a GRAPH pattern. As both conditions are satisfied, we need to create a new LQN activity called *SendSignalAction1*, a *replyentry* and a *replyactivity* in *Lqn.* We also need to identify the location for the newly created elements. From the LQN metamodel we know that *Task* is the container of *task-activities* block or *entry-phase-activities* block and one of them can be the container of activity. In order to identify the location, we invoke the operation *getActivityPartitionForElement* which gets the activity partition for *SendSignalAction1*. Each *ActivityPartition* corresponds to an *Artifact* which is mapped to a *Task* in *Lqn*. Operation *addChildTask* checks the type of entry and its block, and then adds to them the new *activity*, *replyentry* and *replyactivity*. In this case, entry *AcceptEventAction1* is of type *GRAPH* and has a *task-activities* block to which we added the new activity *SendSignalAction1*, *replyentry* and *replyactivity* (as shown in Fig.14). Last, the trace file is updated.

We do not discuss other matches in order to avoid repetition, as they are similar to those already discussed. Some differences do not need any action to propagate the changes to *Lqn* because the guard condition is not met. Other matches have differences on properties that are not mapped to LQN.

6 VALIDATION APPROACH

In order to validate the Incremental Change Propagation approach developed in Epsilon we used multiple test cases including two kinds of changes: a) atomic changes (add or delete model elements one by one) and b) composite changes (add, delete, update and move more elements at a time). Many of the tests for atomic changes are similar to the ADD and DELETE cases described in Section 5. For the composite changes, we used a set of six design patterns used for analyzing the performance effects of SOA design patterns in [25]: Façade, Service Decomposition, Service Callback, Redundant Implementation, Partial State Deferral, User Interface Mediator. Each design pattern was applied by hand to a UML model and then we used the version without and with pattern as *S* and *S'* models. For instance, the changes from Fig. 4.a to Fig 4.b are inspired from the Façade design pattern.

In order to facilitate the task of deciding whether the LQN models produced by our ICP approach are correct, we avoided the use of manual inspection as it is error prone. In fact, manual verification whether two models are identical represents a threat to the validity of the conclusion, especially when the compared models are large. In order to eliminate this threat, we used the following procedure after executing the proposed ICP approach:

- Re-execute the batch transformation taking the evolved UML models as source for updating the LQN model and the trace model. Using EMF Compare, we compare two LQN models for each evolved UML model: a) one generated by re-running the batch transformation and b) the other obtained via the ICP approach.
- A test passes when the two compared LQN models (a) and (b) are identical, which is indicated as "zero differences" by the EMF Compare tool. For all test cases used to validate the ICP implementation, we generated zero differences.

Please note that the updated LQN model is validated not only after completing the incremental change propagation, but also during the execution of ICP. This is done by taking advantage of the EOL run configuration, which checks every LQN model change against the LQN metamodel and displays error messages if any action to update the LQN model during the execution violates the metamodel constraints.

7 CONCLUSIONS

The contribution of this paper is the development of an automatic incremental change propagation (ICP) approach from UML+MARTE software models to LQN performance analysis model in the context of MDE of real-time distributed and embedded systems. The proposed approach supports atomic changes (add, delete) and composite changes (update, move). To the best of our knowledge, our approach is one of the few that can support move and update changes. Our ICP helps in supporting the integration of quantitative performance analysis in the early

phases of software development process, by automating the generation of performance models from the software models under development and keeping them synchronized as the models evolve. This is in agreement with [18], which states that evolution solutions should be integrated with solutions for model quality and model consistency, since the goal of model evolution is to improve the quality of the system.

A direction for future work is to apply the proposed ICP approach to propagate changes to other analysis models for other non-functional requirements (such as reliability, availability, safety) expressed in different formalisms (such as Petri nets variants, Markov Chains, fault trees). Since each analysis model has its own metamodel, we may need to extend our approach to take different actions according to the metamodel relationships. For now, our approach is able to take all the actions necessary to propagate changes from UML to LQN performance models.

ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through its Discovery Grant program.

REFERENCES

[1] Altamimi, T., Mana, Z. and Dorina, P. 2016. Performance analysis roundtrip : automatic generation of performance models and results feedback using cross-model trace links. *CASCON Proceedings Conference*. (2016).

[2] Eclipse EMF Compare: Developer Guide, 2009. */www.eclipse.org/emf/compare/documentation/latest/developer/developer-guide.html#The_Comparison_Model*. Accessed: 2017-05-23.

[3] Gruschko, G., Kolovos, D., Paige, R. 2007. Towards Synchronizing Models with Evolving Metamodels. *Workshop on Model-Driven Software Evolution (MODSE), 11th European Conference on Software Maintenance and Reengineering*. (2007).

[4] Hassani, F., Sedighiani, K., Aliee, F.S. and Angabini, A. 2011. From uni-directional model transformation to incremental model synchronization. *2011 5th Malaysian Conference in Software Engineering, MySEC 2011*. (2011), 88–94.

[5] Hearnden, D., Lawley, M. and Raymond, K. 2006. Incremental Model Transformation for the Evolution of Model-Driven Systems. *9th International Conference, MoDELS,Proceedings*. O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, eds. Springer, 321–335.

[6] Herrmannsdoerfer, M., Benz, S. and Juergens, E. 2009. COPE - automating coupled evolution of metamodels and models. Springer LNCS, Vol. 5653, (2009), 52–76.

[7] Herrmannsdoerfer, M., Vermolen, S.D. and Wachsmuth, G.H. 2010. An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. *Proceedings of the 3rd Int. Conf. on Software Language Engineering, SLE 2010*. Springer, LNCS 6563 (2010).

[8] Jimenez, A.M. 2005. *Change propagation in the MDA: A model merging approach*. University of Queensland.

[9] Johann, S. and Egyed, A. 2004. Instant and incremental transformation of models. *Proceedings - 19th International Conference on Automated Software Engineering, ASE 2004*. (2004), 362–365.

[10] Khalil, A. and Dingel, J. 2013. *Supporting the Evolution of UML Models in Model Driven Software Development : A Survey*. Technical Report 602, School of Computing, Queen's University, Ontario, Canada.

[11] Kolovos, D.S., Di Ruscio, D., Pierantonio, A. and Paige, R.F. 2009. Different models for model matching: An analysis of approaches to support model differencing. *Cvsm*. (2009), 1–6.

[12] Kolovos, D.S., Rose,L.,García-Domínguez, A., Paige, R.F., The Epsilon Book, www.eclipse.org/epsilon/doc/book/, last visitedAugust 2017..

[13] Kusel, A., Etzlstorfer, J., Kapsammer, E., Langer, P., Retschitzegger, W., Schoenboeck, J., Schwinger, W. and Wimmer, M. 2013. A survey on incremental model transformation approaches. *CEUR Workshop Proceedings*. 1090, (2013), 4–13.

[14] Levendovszky, T., Rumpe, B. and Sprinkle, J. 2011. Model Evolution and Management. *Model-Based Engineering of Embedded Real-Time Systems*. (2011), 241–270.

[15] Mani, N., Petriu, D.C. and Woodside, M. 2013. Propagation of incremental changes to performance model due to SOA design pattern application. *2013 4th ACM/SPEC International Conference on Performance Engineering, ICPE 2013*. (2013), 89–100.

[16] Méndez-Acuña, D. and Casallas, R. 2012. Effective detection of model changes. *2012 7th Colombian Computing Congress, CCC 2012 - Conference Proceedings*. (2012), 0–4.

[17] Mengerink, J.G.M., Schiffelers, R.R.H., Serebrenik, A. and Van Den Brand, M.G.J. 2016. DSL/Model Co-Evolution in Industrial EMF-Based MDSE Ecosystems. *ME@ MODELS* (2016), 2–7.

[18] Mens, T., Blanc, X. and Mens, K. 2007. Model-driven software evolution : An alternative research agenda. *The 6th BElgian-NEtherlands software eVOLution workshop (BENEVOL 2007)* (2007), 1–7.

[19] Mens, T., Michel, W. and Stephane, D. 2005. Challenges in Software Evolution. *8thj International Workshop on Principles of Software Evolution (IWPSE'05)*, 13–22.

[20] Mens, T., Serebrenik, A. and Cleve, A. 2014. *Evolving Software Systems*. Springer-Verlag Berlin Heidelberg.

[21] Object Management Group 2015. Unified Modeling Language (UML). *Version 2.5,formal-15-03-01*.

[22] Petriu, D.C. 2015. Challenges in integrating the analysis of multiple non-functional properties in model-driven software engineering. *Proc. of ACM/SPEC Workshop on Challenges in Performance Methods for Software Development*, *WOSP-C 2015,* 41–46.

[23] Ráth I., Bergmann G., Ökrös A., V.D. 2008. Live Model Transformations Driven by Incremental Pattern Matching. *ICMT '08 Proceedings of the 1st international conference on Theory and Practice of Model Transformations Pages 107 - 121* (Berlin, Heidelberg, 2008), 107–121.

[24] Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M. and Mei, H. 2007. Towards automatic model synchronization from model transformations. *Proceedings of the 22 IEEEACM international conference on Automated software engineering ASE 07*. (2007), 164–173.

[25] Mani, N., 2015. Studying the Performance Impact of SOA Design Patterns via Coupled Model Transformations. PhD Thesis, Carleton University, Dept. of Systems and Computer Engineering.