Cross-Model Traceability for Coupled Transformation of Software and Performance Models

Nariman Mani, Dorina C. Petriu, Murray Woodside
Department of Systems and Computer Engineering, Carleton University
Ottawa, Ontario, Canada
{nmani | petriu | cmw}@sce.carleton.ca

Abstract— **In Model Driven Engineering, the relationship between a source and target model can be maintained, when the source model undergoes changes, by a coupled transformation, whereby changes applied to the source model are incrementally propagated to the target model. Cross-model traceability links are key to applying the correct changes to the target model. The coupled transformation considered in this paper propagates changes to a Layered Queueing Network (LQN) performance model (originally derived from a UML design model of a SOA system) as an effect of applying design patterns to the SOA model. A special problem arises because of differences in the level of abstraction between UML and LQN (i.e. a performance model element may represent a set of many design model elements). This paper bridges the abstraction gap between models by proposing traceability links that use new collection types (not defined in the source metamodel) to represent complex source model elements, which are then mapped to simple target model elements.**

Keywords- Software performance model, service oriented systems, SOA pattern, traceability links, change propagation, LQN

I. INTRODUCTION

In Model Driven Engineering (MDE), the performance of a Service Oriented Architecture (SOA) design can be evaluated in early lifecycle phases using a quantitative performance model (hereafter called the *PModel*) generated by a model transformation from the software design model (hereafter called the *SModel*) extended with performance annotations. An example of such a technique is the Performance from Unified Model Analysis (PUMA) [1][8]. Also, system designers often apply SOA patterns [2] to system designs as generic solutions for architectural, design and implementation problems, and study their performance impact with the help of the corresponding PModel. A pattern could have a significant performance cost due to the overheads it may introduce, in which case the performance cost can be balanced against the benefits of the pattern, and alternative pattern configurations can be compared. Traditionally, a new performance model needs to be generated by reusing the PUMA technique to evaluate the impact of the design pattern changes on the design model. However, this has drawbacks: **1)** it masks the causal connections between the design changes and the performance impact which can provide significant insight to the engineer to make design choices; **2)** it is a substantial waste of execution cost, which could be significant if the cycle of choosing a pattern, applying and evaluating it is repeated many times during the development process of large systems. Cross-model traceability can help by maintaining consistent relationships between the source and target model elements from the moment the target model is generated; when changes occur in the source model, only the affected target model elements are identified and changed. Compared to generating a new performance model by techniques such as PUMA every time a pattern is applied, the cross-model traceability links show the causal connections between the SModel changes and the resulting PModel changes with a reduced effort. It also enables incremental studies of numerous design alternatives when applying a large number of SOA design patterns.

Cross-model traceability links are straightforward when the cross-model relationships between elements are one-to-one, or one–to-many. However, when generating a PModel from a SModel, it often happens that one PModel element is created from a set of SModel elements, due to the fact that the level of abstraction of PModel is higher. An example is a collection of SOA activities which make up a single service operation in the PModel. The requirements for these specific collection elements are defined in Section V and VI.

In [3] we proposed a coupled refactoring technique which incrementally propagates the SModel changes (due to the application of a SOA design pattern) to the PModel. It uses entity-to-entity traceability links without considering the abstraction gap and makes it the responsibility of the designer to identify a collection of SModel entities which all trace to the same PModel entity. This process is error-prone and requires deep designer understanding of the process.

The difference in this paper is that we address the challenge of bridging the abstraction gap between the two models by defining new types for complex source model concepts (not corresponding to any meta-class in the source metamodel) and mapping them to the target model concepts. Moreover, in this paper we propose an improved (and in fact simplified) version of the coupled transformation process in [3] based on these extended types, as explained in sections VI and VII. This makes the coupled transformation more accurate and easier to automate. Furthermore, the extended types can also be used to trace the performance results obtained by solving the PModel back to the collections of SModel elements corresponding to the PModel elements.

In this paper, the SModel uses UML extended with the SoaML profile [4, 5] for SOA concepts and the MARTE profile

DOI reference number: 10.18293/SEKE2016-142

(Modeling and Analysis of Real-Time and Embedded systems) [6] for performance-related information. The PModel is expressed in the Layered Queueing Network (LQN [7]) formalism. The initial PModel is created from the annotated UML using the PUMA tools [1, 8]. All of these models are briefly described in Section IV.

II. Related Work

Traceability is frequently employed in approaches to software model transformation. In [9], the authors present a method which attaches traceability generation codes to pre-existing ATL programs [10]. The method produces a loosely coupled traceability, meaning it can be used for any kind of one-to-one traceability. In [11] a method is presented for generating annotated models with traceability information, by merging the models with the trace models. The generated trace-links are embedded in the target model, in elements they refer to, or are stored externally in a separate model.

Managing the complexity of traceability information in MDE is discussed in [12]: a) how to identify different kinds of trace-links that may appear in MDE; and b) propose a rigorous approach for defining semantically rich trace-links between models. In [13] the authors propose a traceability framework, implemented in the model-oriented language Kermeta, to facilitate modeling transformations. Using a trace metamodel, the framework allows for tracing the transformation chain within Kermeta. Model transformation trace-links are defined in the metamodel as a set of source nodes and target nodes.

None of the above works addresses traceability between models at different levels of abstraction. On the other hand, reverse engineering transformations, which do raise the abstraction level, do not emphasize traceability, perhaps because in reverse engineering there is less interest in retaining the connection with the original model. In reverse engineering of design models from code [14] a single design element may be represented by many scattered features of the code, with structured relationships which must be captured in the traceability link. The taxonomy from [14], for example, does not mention traceability links. However, coupled transformations of software and performance models, such as our proposed techniques in [3], require constructing and maintaining these links. Therefore, the technique proposed in this paper that addresses the abstraction gap between software and performance models, does improve our previous approach from [3].

III. Overview of the Improved Approach

Figure 1 shows an overview of the coupled transformation technique [3], enhanced with extended types for traceability links introduced in this paper for propagating changes due to design patterns. The inputs to the process include the initial SOA SModel (top left), and a library of pattern definitions (bottom left). The enhanced traceability links are used in stage C (shown in grey) for translation of the SModel refactoring transformation rules into PModel refactoring transformation rules. The designer steps (supported by tools developed by the authors) are shown on the left side and the automated steps on the right side.

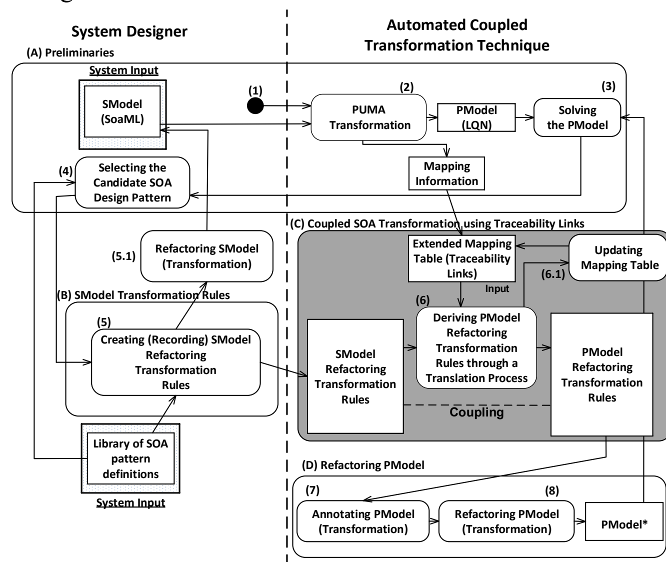

Figure 1: Overview of Improved Approach using Extended Traceability Links for Coupled Transformations

The extended approach has four stages:

A. Preliminaries: This stage gets the SModel as an input and creates the base PModel using PUMA [1]. The mapping information between the SModel and the constructed PModel is created during this initial transformation process (Step (2)). The mapping information is used by the technique in this paper for creating the cross-model traceability links with extended types in a mapping table which will be used in stage C. Pattern application begins at step (4), where the designer selects a candidate pattern for its own reasons (e.g. maintainability).

B. Model Transformation Rules: The selected pattern is specified using Role-Based Modeling RBML [15], a graphical pattern specification approach which uses model roles to identify the participating elements. The designer indicates where the pattern is applied by binding pattern roles to elements in the SModel and then records SModel transformation rules that will satisfy the solution specification (step (5)).

C. Deriving the PModel Transformation Rules: Using the traceability links with extended type extracted from the mapping information generated in Stage (A) and the SModel refactoring transformation rules from Stage (B), the PModel refactoring transformation rules are derived automatically in the coupled transformation process (Step 6). The dashed line between the "SModel Refactoring Transformation Rules" and "PModel Refactoring Transformation Rules" represents the coupling between them. If there are updates to the table of traceability links (mapping table) due to add/deletion of the elements, this is being done as part of Step 6.1.

D. Refactoring PModel: The PModel refactoring transformation rules are executed by a transformation engine to refactor the PModel into the final PModel* (Steps 7 and 8). Although the steps in this stage are explained briefly in Section VII.B , the details are discussed in [3] and are not

within the scope of this paper as they are not impacted by the proposed extended types in this paper.

IV. Models

A. SOA Models

From the range of views in SoaML [4, 5] used to model SOA systems, we use the Business Processes Model (BPM) for behavior and the Service Architecture Model (SEAM) for structure and contracts, together with a UML deployment diagram. The SEAM is specified as a UML collaboration diagram with service participants and contracts (with SoaML stereotypes *«Participant»* and *«ServiceContract»* respectively). Each participant plays a role of Provider or Consumer with respect to a contract. Participants correspond to pools, participants and swimlanes in the BPM. The BPM is specified as a UML Activity Diagram (AD) (see Figure 2). Service invocations are modeled as operation calls, using three types of UML actions: a *CallOperationAction* sends a service request and waits for the reply via its input/output pins; an *AcceptCallAction*, an accept event action, waits for the request arrival; and a *ReplyAction* returns the reply values to the caller. The called operation name appears in '()' as *"(class-name::operation-name)"*. We assume that all BPM edges between ActivityPartitions represent calling interactions, connecting these three types of Actions.

MARTE performance annotations are given in shaded notes. BPM describes the behavior as a sequence of steps *«PaStep»* with a workload attached to the first step stereotyped as *«GaWorloadEvent»*. *«PaStep»* has attributes *hostDemand* (required CPU time), *rep* (mean repetitions) and *prob* (probability of optional step). *«GaWorloadEvent»* defines a population of *Nusers* users, each with a thinking time *ThinkTime* defined by MARTE variables. Concurrent runtime instances *«PaRunTInstance»* are identified with swimlane roles. UML Deployment Diagram (DP) is also defined, as in Processing nodes are stereotyped *«GaExecHost»* and communication network nodes are stereotped *«GaCommHost»*, with attributes for processing capacity, message latency and communication overheads.

B. Performance Model

PModels are expressed in an extended queueing notation called Layered Queuing Networks (LQNs) [8], selected because of its close coupling to the high-level software architecture. An LQN estimates waiting for service due to contention for host processors and software servers, and provides response time and capacity measures. Figure 3 shows the LQN model for the example. For each service there is a task, shown as a bold rectangle, and for each of its operations (contracts) there is an entry, shown as an attached rectangle. The task has a parameter for its multiplicity or thread pool size (e.g. {'1'}). Each entry has a parameter for its host CPU demand, equal to the total *hostDemand* of the set of *«PaSteps»* for the same operation in the SModel.

Calls from one entry to another are indicated by arrows between entries (a solid arrowhead indicates a synchronous call for which the reply is implicit, while an open arrowhead indicates an asynchronous call). The arrow is annotated by the number of calls per invocation of the sender. For deployment, an LQN host node is indicated by a round node associated to each task. While Figure 3 shows entries with host demands and calls, there is an optional level of detail which is not shown here, which defines an activity subgraph for each entry with predecessors, successors, forks and joins, similar to a UML activity diagram. The host demands and calls are then defined for each activity.

V. The Abstraction Gap Between SModel and PModel

Each type of traceability link produced by the SModel-to-PModel transformation describes the mapping relationship between a SModel element type (i.e., a meta-class of the UML metamodel) and a PModel element type (i.e., a meta-class of the LQN metamodel). Establishing the traceability links

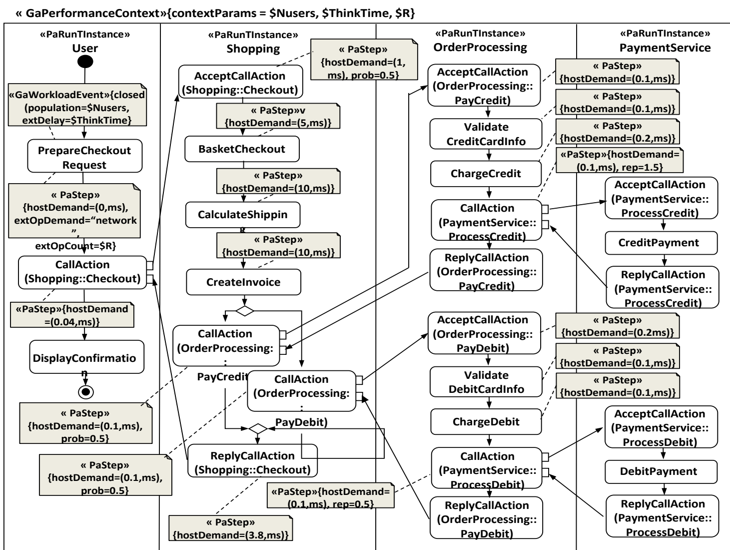

Figure 2: *Checkout Business Process Model for the Online Shop*

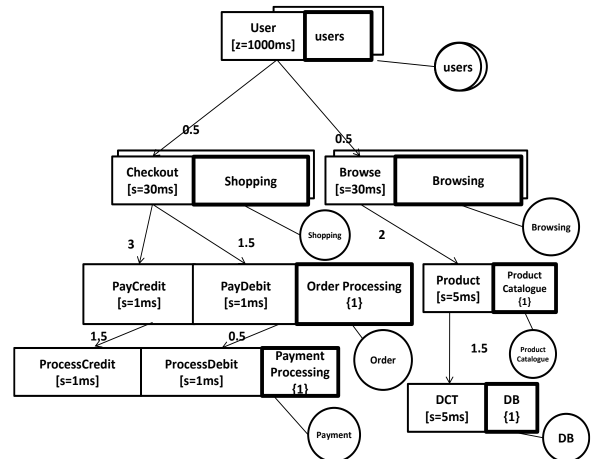

Figure 3: *LQN (PModel) corresponding to SOA design (SEAM, BPM and Deployment Diagram)*

between elements of SModel and PModel which are in a one-to-one relationship is straightforward. A partial list of one-to-one mappings is shown in the following table:

SModel (SoaML) Type	PModel(LQN) Type
SEAM Participant	Task
BPM Swimlane (PaRunTInstance)	Task
BPM Action	Activity
BPM Control Flow	Sequence
BPM Async Call	Async Call
DP Processing Node (ExecHost)	Host
DP Artifact	Task

However, SModel to PModel relationships are not always one-to-one. We have identified cases where a group of interconnected SModel elements (called subgraph) is mapped to one or more PModel elements (i.e. many-to-one or many-to-many relationship). In most cases, there are more SModel elements mapped to fewer PModel elements, indicating that the latter has a higher level of abstraction. These mappings are described below.

A. LQN Entry

An LQN entry of a task represents the entire operation carried out by the task in response to a call. Thus, the entire subgraph of SModel activity diagram actions, control flows, hyper edges and attributes invoked by an AcceptCallAction is mapped to an LQN entry. An example of this type of mapping is shown in Figure 4.

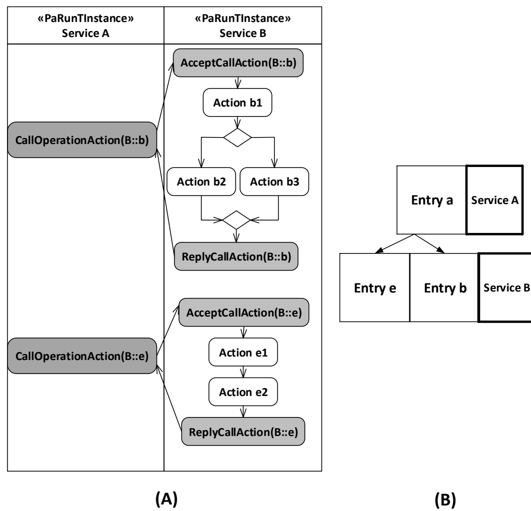

Figure 4: (A) Partial SModel AD Subgraph and (B) Corresponding Partial LQN PModel with mapped Entry

The partial SModel AD and the corresponding partial LQN PModel are shown in Figure 4.A and Figure 4.B, respectively. There are two swimlanes representing "Service A" and "Service B", where "Service B" contains two subgraphs, each accepting a call from a CallOperation-Action in "Service A". The subgraph between *AcceptCallAction(B::b)* and *ReplyAction(B::b)* is mapped to "Entry a", and the one between *AcceptCall-Operation(B::e)* and *ReplyAction(B::e)* to "Entry e" in the partial LQN model shown in Figure 4.B. This is a case of many-to-one mapping.

B. LQN Synchronous Call

An LQN synchronous call corresponds to two messages in the SModel, the call and the corresponding reply. This is also a case of many-to-one mapping. Two examples of this type of mapping are shown in Figure 5. The synchronous call from CallOperationAction(B::b) in "Service A" to AcceptCallAction(B::b) in "Service B" and the reply from ReplyCallAction(B::b) to the caller action in Figure 5.A are mapped to a single LQN call from "Entry a" to "Entry b" in Figure 5.B. Figure 5 shows also an example of nested synchronous calls, where a call to "Service C" is made before the reply from "Service B" to the initial caller, "Service A".

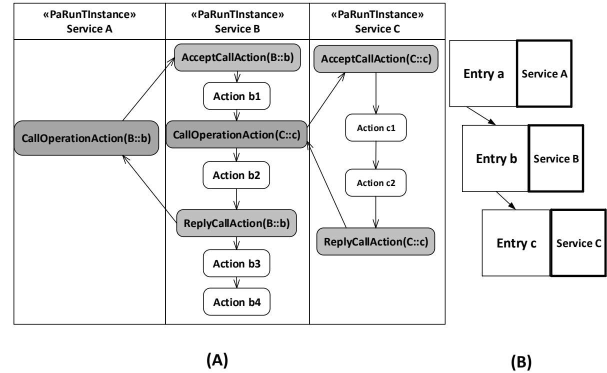

Figure 5: (A) Partial SModel AD with synchronous calls and (B) Corresponding LQN PModel with synchronous calls

C. LQN Asynchronous Call

An LQN asynchronous call represents a SModel call without reply, which also might be part of a forwarding call (to be dealt with next). While this correspondence is one-to-one, it is mentioned here because it only occurs only in a certain context in the SModel.

D. LQN Forwarding Call

An LQN forwarding call represents delegation of responsibility for an operation. It is a chain of calls in the PModel that corresponds to a chain of messages in the SModel. There is an initial synchronous call from a *CallOperationAction* which eventually receives its corresponding reply from a different swimlane than the one it called, and one or more asynchronous calls that forward the caller request to another swimlane; the final one in the chain replies to the initial caller.

This collection of SModel messages is mapped to the following collection of PModel elements: a LQN synchronous call and one or more forwarding calls that forward the request to a final entry, which implicitly provides the reply to the initial caller. This is a case of many-to-many mapping between the SModel and PModel elements. Figure 6 shows an example. *CallOperationAction(B::b)* from "Service A" in Figure 6 .A initiates a call to "Service B", which forwards it to "Service C", which replies to the initial caller. Figure 6.B shows corresponding (mapped) partial LQN PModel with one synchronous call and one forwarding call (i.e. dashed arrow line).

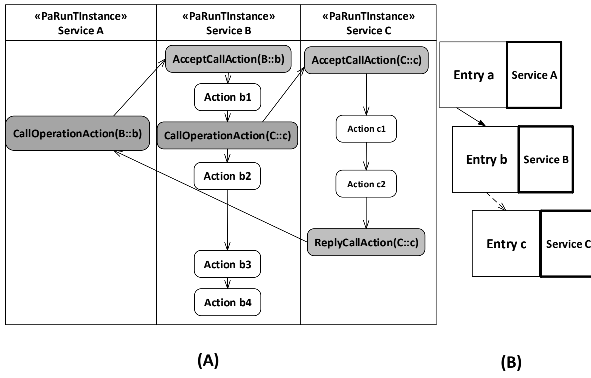

Figure 6: (A) Partial SModel AD forwarding scenario and (B) Corresponding Partial LQN PModel with synchronous & forwarding calls

VI. TRACEABILITY LINKS AND MAPPING TABLE

We assume that the transformation that derives the initial PModel from the SModel (e.g. PUMA) also generates the basic element-to-element mapping between SModel and PModel elements (i.e. in form of a mapping table), as described in [16]. In this paper, the initial mapping table provided by PUMA is extended with the additional higher-level types of traceability link, as described in this section. Mapping table is discussed in Section VI.A and the traceability links metamodel is described in Section VI.B.

A. Mapping Table

The mapping table is a collection of the traceability links of the form:

Traceability Link = (link name, SME , PME)

where SME stands for an SModel Element or attribute, and PME for the corresponding PModel Element or attribute. For example, the traceability link named BTL1 between the BPM swimlane "Service B" in Figure 5.A and the LQN Task "Service B" in Figure 5.B has the form:

Traceability Link = (BTL1, BPM::Swimlane:ServiceB, LQN::Task:ServiceB)

B. Traceability Links Metamodel

To bridge the abstraction gap, in this paper we propose that traceability links in the mapping table use a metamodel that includes the following:

- UML types describing the SModel elements,
- LQN metamodel types describing the PModel elements,
- the following four additional types:
  1. *EntrySME*: a collection of Elements in the SModel containing the elements of the Activity Subgraph invoked by a Call,
  2. *SyncCallSME*: a pair of Call and Reply Actions and their corresponding AcceptCallActions in SModel, that make up a synchronous call
  3. *ForwardingSME*: a collection of CallActions and one ReplyAction in SModel, with their corresponding AcceptCallActions, forming a forwarding pattern as described in Section V.
  4. *LQNFwdCall* or *ForwardingPME:* a corresponding collection of LQN calls in PModel with one synchronous call and one or more forwarding calls.

Therefore, the SME column of the mapping table has UML types plus four additional types for SModel and PME column has one additional type for PModel. The traceability links based on these extended types are called "traceability links with extended types" and are defined in the following sub sections:

1) Traceability Link for an LQN Entry

An entry in the PModel corresponds to a portion of the behaviour specified in an activity diagram, defining the response by a PaRunTInstance (defined by a swimlane) to a call. The subgraph starts with an *AcceptCallAction* following a call from a *CallOperationAction* in another swimlane, and ends where it provides a reply with a *ReplyAction,* or ends, or executes a *CallOperationAction* to another swimlane.

The EntrySME can be discovered automatically by an analysis of the flow of Actions, based on this definition. In presenting it here, it is shown as a list of the elements in the subgraph within '{}' brackets, separated by commas. In this list, the action names are shown first, then the activity control flows each defined as a couple (source, destination) and finally the hyper edges (Decision, Merge, Fork, Join). Each hyper edge is defined as:

Type *(predecessor list, successor list)*
*if **Type** is **Decision** or **Fork**, there is only one predecessor;*
*if **Type** is **Merge** or **Join**, there is only one successor.*

To distinguish the hyperedges in the text, their types are shown in bold.

The EntrySME is mapped to a LQN Entry and each element in the defining list is mapped to a corresponding element of the LQN activity subgraph inside the LQN entry. An example of this type of traceability link, referencing elements in the scenario in Figure 4, is given below:

Link = (BTL2,BPM::EntrySME:{AcceptCallAction(B::b), Action(B::b1), Action(B::b2), Action(B::b3), ReplyAction(B::b), (AcceptCallAction(B::b), Action(B::b1)), ***Decision*** *(Action (B::b1), Action(B::b2), Action(B::b3)),* ***Merge****(Action (B::b2), Action(B::b3), ReplyCallAction(B::b))},LQN::LQN Entry: {(Entry b)})*

2) Traceability Link for an LQN Synchronous call

A synchronous call (i.e., call-reply) in the PModel corresponds to a pair of messages (that is, of ActivityEdges that cross the boundary between two ActivityPartitions), called here a Synchronous Call. The first message is from a *CallOperationAction* in the first BPM swimlane to an *AcceptCallOperation* in the second swimlane; the second message is from a *ReplyAction* in the second swimlane to the initiating *CallOperationAction*. A SyncCallSME is defined as a two-element list as follows:

{(CallOperationAction, AcceptCallOperation), (ReplyAction, CallOperationAction) }

which is mapped to the corresponding LQN Synchronous Call. An example of this type of traceability link based on the scenario in Figure 5 is given below:

Link = (BCTL3,BPM::SyncCallSME: {(CallOperationAction(B::b), AcceptCallAction(B::b)), (ReplyAction(B::b),CallOperationAction(B::b))}, LQN::LQNSyncCall: {(Entry a, Entry b)})

Table 1: Examples of corresponding SModel and PModel element types in the traceability links of the Mapping Table

	Sub-table (A) Types for Structural Elements	
	SME	**PME**
C1	Participant (in SEAM)	LQN Task
C2	Host Node (in Deployment)	LQN Host
C3	ActivityPartition/Swimlane (in BPM) stereotyped «PaRunTInstance»	LQN Task
C4	EntrySME (collection of elements forming an activity subgraph)	LQN Entry
C5	BPM Action	LQN Activity
	Sub-table (B) Calls	
C6	SyncCallSME (collection of calls)	LQN Synchronous Call/LQNSyncCall
C7	ForwardingSME (collection of calls)	LQN Forwarding Call/LQNFwdCall (collection of calls)
C8	Asynchronous Call	LQN Asynchronous Call/LQNAsyncCall
	Sub-table (C) Attributes	
C9	MARTE WorkloadEvent.*extDelay*	Think Time of a workload
C10	MARTE ExecHost.*resMult*	Processor Multiplicity
C11	MARTE PaRunTInstance.*poolsize*	Task Multiplicity

Table 2: Mapping Table with examples of traceability links for Shopping and Browsing SModel and PModel

Sub-table (A) Structural Elements		
Link	**SME**	**PME**
DTL3	Deployment Node Order Host	LQN Host Order
DTL2	Deployment Artifact Browsing	LQN Task Browsing
STL3	SEAM Participant User	LQN Task User
STL2	SEAM Participant Browsing	LQN Task Browsing
BTL1	EntrySME:{AcceptCall (Shopping::Checkout), BasketCheckout, CalculateShipping, CreateInvoice, CallOperation(OrderProcessing::PayCredit), CallOperation(OrderProcessing::PayDebit), ReplyCallAction(Shopping::Checkout), (AcceptCall (Shopping::Checkout), BasketCheckout), (BasketCheckout, CalculateShipping), (CalculateShipping, CreateInvoice), **Decision**(CreateInvoice, CallOperation(OrderProcessing::PayCredit) , CallOperation(OrderProcessing::PayDebit)), **Merge**(CallOperation(OrderProcessing::PayCredit) , CallOperation(OrderProcessing::PayDebit), ReplyCallAction(Shopping::Checkout))}	LQNEntry: {(Checkout)}
Sub-table (B) Calls		
BCTL1	SyncCallSME:{(CallOperationAction(Shopping::Checkout), (AcceptCallAction(Shopping::Checkout)), ReplyAction(Shopping::Checkout), CallOperationAction(Shopping::Checkout))}	LQNSyncCall: {(User,Checkout)}
Sub-table (C) Attributes		
BATL1	MARTE Attribute *hostDemand* for the AcceptCallAction(Shopping::Checkout)	Host Demand attribute of LQN AcceptCallAction in Entry Checkout

3) Traceability Link for an LQN Forwarding Call

A Forwarding Call identifies a call pattern which includes synchronous and asynchronous calls. It begins with a synchronous call from a *CallOperationAction* in one BPM swimlane to an *AcceptCallOperation* in a second swimlane. However instead of a reply, this operation ends with a call that forwards the request to a third swimlane. It may be forwarded any number of times, until a reply is sent back to the originating swimlane. The fact that the reply is coming from a swimlane which is different than the swimlane of the receipt of initial call shows that the request has been forwarded to other swimlanes for processing. A ForwardingSME is defined as follow: it begins with *SyncCallSME* (defined above for the synchronous calls) which is followed by a comma-separated list of forwarding calls, as:

ForwardingSME:{ SyncCallSME, (frwdCall1), (frwdCall2),…}

A ForwardingSME is mapped to a collection of PModel elements containing an LQN synchronous call and one or more LQN forwarding calls. The last forwarding call implicitly generates the reply to the originating entry. This requires another extended type for the LQN column of the mapping table, called *LQNFwdCall*. A *LQNFwdCall* begins with *LQNSyncCall* and it is defined similarly by a list:

LQNFwdCall: { LQNSyncCall, (frwdCall1), (frwdCall2),…}

An example of this type of traceability link based on the scenario in is:

*Link = (BCTL4,**BPM::ForwardingSME**: {(CallOperationAction(B::b), AcceptCallAction(B::b)), (ReplyAction(C::c), CallOperationAction(B::b)), (CallOperationAction(C::c), AcceptCallAction(C::c))}, **LQN::LQNFwdCall:**{(Entry a, Entry b),(Entry b, Entry c)})*

Table 1 shows examples of corresponding SModel and PModel element types in the traceability links of the mapping table. Table 1 is organized into three groups for Structural Elements, Calls, and Attributes. Table 2 gives some examples of traceability links established between the SModel (BPM is shown in Figure 2) and PModel (Figure 3) for the shopping and browsing SOA.

VII. Coupled Transformation using Extended Traceability Links

The coupled transformation technique in [3] includes a formal recording of the refactoring of the SModel, and automatic derivation of the refactoring transformation of the PModel as well as its automatic application to the PModel. This section describes how these steps must be modified (enhanced) to accommodate the extended types of cross-model traceability links with extended types proposed in this paper. An overview of the modified process is also shown in Figure 1 (Stages B and C).

A. Coupled PModel Refactoring Rules

The coupled transformation in [3] begins with recording the SModel refactoring that arises from the pattern application. A tool was implemented in [3] to assist the system designer with this process. With the extended link types introduced in this paper, the system designer can now create rules at various levels of abstraction. Therefore in this paper, the tool in [3] is enhanced to support extended types. The designer creates the rules using the provided tool by selecting SMEs from a table created from the UML specification, and can choose the level of abstraction by choosing from the extended element types. For example, when refactoring behavior, rules can be applied either to an entire EntrySME or to its individual activities, as the designer wishes. The automated translation of the SModel refactoring rules into PModel refactoring rules is based on the cross-model traceability links in the mapping table described in Section VI, using both the Type Correspondences table (e.g. Table 1) and the Mapping Table (e.g. Table 2). With the extended model and link types the process of generating the

PModel refactoring rules is made more accurate, simpler and more uniform. Therefore the automated process in [3] is modified to use the extended types as follow. Figure 7 shows a partial screenshot of the tool that takes care of the enhanced automated translation using the traceability links. Each SModel refactoring rule has an operation name and some arguments, which are processed as follows:

1. The operation name generates one or more PModel operations. The action part of the name (add/ delete/ modify) is retained, and the operand-type part (e.g. Participant) is mapped according to the Type Correspondences table, similar to one shown in Table 1 For example the SModel operation addParticipant is translated to addTask, and deleteEntrySME to deleteEntry.
2. The arguments of the PModel operation (e.g. the element or elements to be added, deleted, or modified) are translated from the arguments of the SModel operation using the Mapping Table (e.g. Table 2). In an "add" operation the name of the new PModel element is taken as the name of the corresponding SModel element.

For example, a SModel "addParticipant" operation is mapped to "addTask" for the PModel, and the "addParticipant" argument becomes the new task name. Modifications to calls require special consideration in the translation. The SModel "modifyActionCall" operation changes a service invocation from a *CallOperationAction* to an *AcceptCallAction*. As this might apply to more than one call to the same *AcceptCallAction,* the mapping table is searched (using a MappingTableSearchByKey command) to identify all the PModel activities making the call. Then the operation is mapped to one or more "modifyActivity" operations in the PModel domain, to change all the calls. Some of the PModel transformation rules derived for the Façade pattern are presented in Figure 7.

B. PModel Refactoring

The extended types introduced in this paper do not impact the process of applying the PModel refactoring rules (stage D in Figure 1) discussed as part of the coupled transformation technique in [3]. Briefly, first the PModel is annotated with transformation directives indicating the changes, then the changes are applied by a transformation engine implemented using QVT Operational (Query, View, and Transformation, a OMG standard model transformation language) which processes the directives. The details are provided in [3].

VIII. Case Study

To illustrate the application of the coupled transformation and the role of the extended types and traceability links in the mapping, two SOA patterns will be applied to the Shopping and Browsing SOA given in Figure 2 and Figure 3.

Suppose a designer must re-design a Shopping and Browsing SOA to support three different user types (mobile phone, desktop, kiosk) through a multi-channel endpoint. First, the designer applies the pattern "Concurrent Contracts"[2], which addresses the following problem: *A service's contract may not be able to support all potential types of clients, because of access and interface differences*. The pattern also suggests the following solution [2]: *To accommodate different types of clients, separate service contracts ("channels") can be created for the one underlying service implementation*. Using the Concurrent Contracts pattern, separate shopping and browsing operations are provided for each group of users. Separate sets of actions (in form of activity subgraphs, i.e. EntrySMEs) are created in the Shopping swimlane (see Figure 3) and also the Browsing swimlane (not shown in Figure 3). Using the corresponding types table (using row *C4* in Table 1), traceability links are created for these EntrySMEs mapping the activity subgraphs to new PModel LQN entries (LQNEntry) of the Shopping and Browsing tasks. Although this allows each contract to be extended and managed individually, it introduces duplication in the functional design. Furthermore, when a service is subject to change due to contract changes, the core service logic needs to be extended

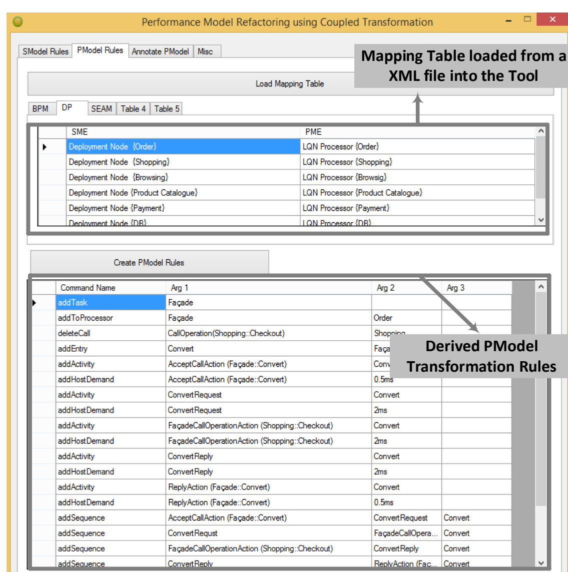

Figure 7: Tool for automatic derivation of PModel refactoring rules

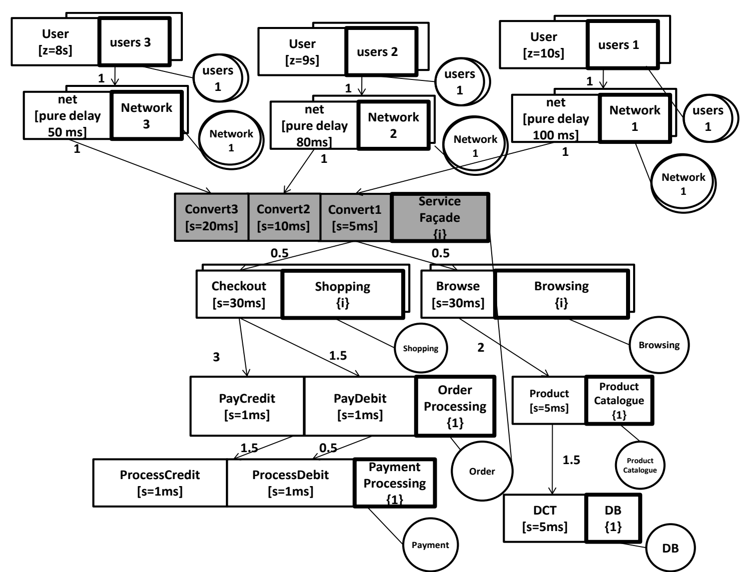

Figure 8: Shopping and Browsing LQN PModel with the Façade Design Pattern

and augmented to accommodate the change. This leads the designer to consider another SOA design pattern called "Service Façade", which addresses the following problem according to [2]: *The tight coupling of the core service logic to its contracts can obstruct its evolution and negatively impact service consumers.* The solution suggested in [2] is: *Façade logic is inserted into the service architecture to establish a layer of abstraction that can adapt to future changes to the service contract.*

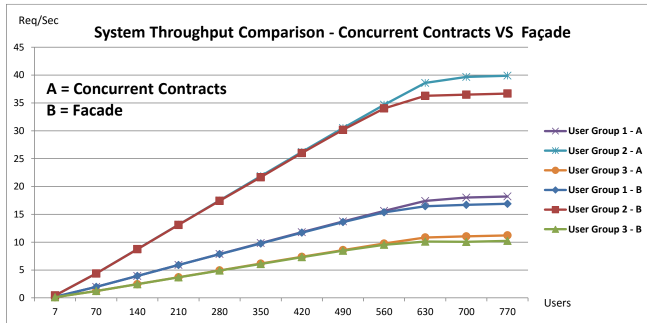

Figure 9: System Throughput (Requests/Sec) for Concurrent Contracts and Façade

To apply the Façade design pattern, a new Façade swimlane is created in the BPM with one new EntrySME activity subgraph for each service contract. The new swimlane is mapped to a new LQN task ServiceFacade in the PModel (using row *C3* in Table 1), and the new EntrySMEs are mapped its entries (LQNEntry) (using row *C4* in Table 1). The BPM synchronous calls from the users to the Shopping swimlane are redirected to pass through the façade EntrySMEs. The synchronous calls are mapped to LQN synchronous calls through the traceability links with SyncCallSME types (*C6* in Table 1). The LQN PModel after application of the Façade Design pattern is shown in Figure 8 (PModel before application is shown in Figure 3). The refactored PModel is solved by LQN Solver tool [8] for performance analysis (i.e. throughput, response time, utilization. etc). Figure 9 compares the system throughput (request/sec) given by the LQN model solver for the two cases (Concurrent Contracts and Facade) when the total number of users varies for three types of users. Figure 9 shows that application of Façade design pattern is consistently making the system throughput worse compared to Concurrent Contracts, due to the additional overhead. This shows that Façade has a performance cost to balance against its architectural benefits.

IX. Conclusion

Establishing cross-model traceability links is challenging when there is an abstraction gap between the source and target models. In case of the SOA SModel and the corresponding PModel created by the PUMA [1] transformation chain, the abstraction difference involves collections of elements in both models. In this paper, four relationships were identified which involve many-to-one or many-to-many mappings which reveal the gap in the level of abstraction between SModel and PModel. To bridge the abstraction gap, these four additional types for subgraphs of SModel and PModel elements are defined and used in establishing the extended traceability links. These additional types do not correspond to any meta-class of the source or target metamodel. The syntactic correctness of the corresponding artifacts is verified by the model transformation that generates the target model from the source model, and also identifies the model elements contained in every artifact. We also modified the coupled transformation technique in [3] to use the traceability links with extended types which keeps consistent the SModel and PModel after the application of a SOA pattern. Examples of use are described with coupled refactoring transformations that represent the application of two SOA patterns: "Concurrent Contracts" and "Service Façade".

Acknowledgement

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

References

[1] M. Woodside, D. Petriu, J. Merseguer, D. Petriu, and M. Alhaj, "Transformation challenges: from software models to performance models," *Software & Systems Modeling,* vol. 13, pp. 1529-1552, 2014.

[2] T. Erl, *SOA Design Patterns* Boston, MA: Prentice Hall PTR, 2009.

[3] N. Mani, D. Petriu, and M. Woodside, "Exploring SOA Pattern Performance using Coupled Transformations and Performance Models," *the 27th International Conference on Software Engineering and Knowledge Engineering (SEKE 2015)*, Pittsburgh, PA, USA, 2015, pp. 552-557.

[4] Object Management Group, "Unified Modeling Language (UML)," Version V2.4.1,formal/2011-08-05

[5] Object Management Group, "Service oriented architecture Modeling Language (SoaML) " Version 1.0.1, formal/2012-05-10

[6] Object Management Group, "A UML Profile for MARTE (Modeling and Analysis of Real-Time and Embedded systems)," Version 1.1, formal/2011-06-02.

[7] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi, "Enhanced Modeling and Solution of Layered Queueing Networks," *IEEE Trans. on Software Eng.,* vol. 35, pp. 148-161, 2009.

[8] M. Woodside, D. C. Petriu, D. B. Petriu, H. Shen, T. Israr, and J. Merseguer, "Performance by Unified Model Analysis (PUMA)," *WOSP '05 Proc. of the 5th international workshop on Software and performance*, Palma de Mallorca, Illes Balears, Spain, 2005, pp. 1 - 12

[9] F. Jouault, "Loosely Coupled Traceability for ATL. In: Traceability Workshop," *Traceability Workshop at European Conference on Model Driven Architecture (ECMDA-TW)*, Nürnberg, Germany, 2005, pp. 29–37.

[10] F. Jouault and I. Kurtev, "Transforming Models with ATL," *MoDELS'05 Proc. of the 2005 international conference on Satellite Events at the MoDELS*, Montego Bay, Jamaica, 2005, pp. 128-138.

[11] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "On-Demand Merging of Traceability Links with Models," *3rd ECMDA Traceability Workshop*, 2006.

[12] R. F. Paige., N. Drivalos., D. S. Kolovos., K. J. Fernandes., C. Power., G. K. Olsen., *et al.*, "Rigorous identification and encoding of trace-links in model-driven engineering," *Software and Systems Modeling (SoSyM),* vol. 10, pp. 469-487, 2011.

[13] J.-R. e. Falleri, M. Huchard, and C. e. Nebut, "Towards a Traceability Framework for Model Transformations in Kermeta," *Traceability Workshop at European Conference on Model Driven Architecture (ECMDA-TW)*, 2006, pp. 31-40.

[14] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: a taxonomy," *Software* vol. 7, pp. 13 - 17, 1990.

[15] R. B. France, D.-K. Kim, S. Ghosh, and E. Song, "A UML-Based Pattern Specification Technique," *IEEE Trans. Software Eng.,* vol. 30, pp. 193-206, 2004.

[16] M. Alhaj and D. Petriu, "Traceability Links in Model Transformations between Software and Performance Models," in *SDL 2013: Model-Driven Dependability Engineering*. vol. 7916, F. Khendek, M. Toeroe, A. Gherbi, and R. Reed, Eds., Springer, 2013, pp. 203-221.