# Back-annotation of Simulation Traces with Change-Driven Model Transformations

Ábel Hegedüs, Gábor Bergmann, István Ráth and Dániel Varró
*Department of Measurement and Information Systems*
*Budapest University of Technology and Economics*
*Budapest, Hungary*
*Email: {hegedusa,bergmann,rath,varro}@mit.bme.hu*

***Abstract*—Model-driven analysis aims at detecting design flaws early in high-level design models by automatically deriving mathematical models. These analysis models are subsequently investigated by formal verification and validation (V&V) tools, which may retrieve traces violating a certain requirement. Back-annotation aims at mapping back the results of V&V tools to the design model in order to highlight the real source of the fault, to ease making necessary amendments.**

**Here we propose a technique for the back-annotation of simulation traces based on change-driven model transformations. Simulation traces of analysis models will be persisted as a change model with high-level change commands representing macro steps of a trace. This trace is back-annotated to the design model using change-driven transformation rules, which bridge the conceptual differences between macro steps in the analysis and design traces. Our concepts will be demonstrated on the back-annotation problem for analyzing BPEL processes using a Petri net simulator.**

***Keywords*-back-annotation, simulation traces, change-driven model transformations**

This work was partially supported by EU projects SENSORIA (IST-3-016004) and SecureChange (ICT-FET-231101).

## I. Introduction

Model-driven analysis with hidden formal methods has become a popular approach in critical systems and services design. Various approaches with close conceptual correspondence has been proposed in various application domains such as service-oriented computing [1], [2], dependable systems [3]–[5], avionics [6], etc.

In order to detect design flaws early, high-level design (engineering) models (such as UML Statecharts, BPEL [7], AADL, SysML, etc.) are automatically mapped into different analysis models (such as transition systems, Petri nets, process algebras, etc.) in order to carry out formal verification and validation (V&V) by back-end analysis tools. Since formal analysis models are derived by automated model transformations, and the target analysis tools are also typically fully automated, systems and services engineers obtain a push-button technique for formally analyzing their high-level design models. In the current paper, we primarily focus on V&V tools like simulators and model checkers, which aim at retrieving a trace as a counter-example that violates a certain requirement.

However, such counter-example traces can be very complex resulting with well over 100 elementary steps in industrial scenarios. As a result, systems designers need bridge a significant conceptual gap when they try to interpret what a counter-example means in the *original* design model. Back-annotation aims at automatically mapping back the results of V&V tools to the original design model in order to highlight the real source of the flaw.

Due to the semantic differences between high-level design models and lower-level formal analysis models, we argue in this paper that the *general back-annotation problem aiming to map a trace of a target model to a trace in the source model* can be very complicated. This is due to the fact that most traditional source-to-target (design-to-analysis) model transformations carry out an abstraction, thus they are not reversible. However, in order to completely hide the underlying formal model from systems engineers, model-driven analysis need to provide automated support the back-annotation of target (analysis) models to the source (design) model.

Unfortunately, existing back-annotation approaches are either dedicated to the source and target languages, or they make strong assumptions on the back-annotation problem.

In the paper, we first provide a generic formulation of the back-annotation problem (Sec. II) where back-annotation is defined in the context of an arbitrary, but precisely defined pair of source (design) and target (analysis) modeling languages with dynamic behavior. Then we propose a technique for the back-annotation of simulation traces based on change-driven model transformations [8]. (1) First, simulation traces of the target analysis model will be persisted as a hierarchical change model capturing both the macro steps and micro steps in a trace (Sec. III). (2) Then this target trace is back-annotated to a trace of the source (design) model using change-driven transformation rules, which bridge the conceptual differences between macro steps in the analysis and design traces (Sec. IV).

Our concepts will be demonstrated on the back-annotation problem for analyzing BPEL processes using a Petri net simulator. As a result of our back-annotation approach, execution traces derived by a Petri net simulator can be replayed directly on the BPEL level in order to completely hide the underlying formal models.

## II. Generic back-annotation framework for dynamic modeling languages

*Back-annotation* is the reverse model transformation problem to derive corresponding source design model information from the results of a formal analysis carried out on a target model. In case of *discrete event-based dynamic analysis languages*, these results are typically represented as an *(execution) trace* obtained from a simulation run or as a counter-example (i.e. sequence of steps leading to the violation of a requirement) of a model checker. Therefore the back-annotation in this case consists of deriving a trace of the source model from a given trace of the target model.

In the paper, we first propose a generic framework for the back-annotation of simulation traces. In this framework, we assume the existence of the following traditional modeling and transformation concepts:

- **static, dynamic and trace metamodels** for both the source and target domains to precisely define the taxonomy for instance models;
- **operational semantics** to specify the dynamic behaviour of the target analysis model during simulation;
- **structural model transformation** which derives a (static) target model from an arbitrary source model;
- **traceability links** created by the model transformation to store the structural correspondence between the elements of the source and target models.

Using these concepts back-annotation necessitates the definition of precise transformations for:

1) **trace generation rules** derived from the operational semantic rules of the model to record a simulation run as a trace model observing the changes of the dynamic model.
2) **trace processing rules** to replay the execution of a simulation run on the dynamic model using an arbitrary trace of the same model.
3) **trace mapping (back-annotation) rules** to derive a source trace model from the target trace model using traceability links existing between the models.

In the current paper, we exclusively focus on the problem of back-annotation (or target-to-source trace mapping). For the details of trace generation and processing within a dynamic language, the reader is referred to [9].

### A. Metamodels of dynamic modeling languages

We assume the existence of various metamodels in the context of a *dynamic modeling language (DML)*, which are exemplified together with main relationships in Fig. 1.

First, a **static metamodel** $MM_{stat}$ defines the static structure of a language including possible types of model elements, their main attributes and relations with other model elements. An instance of this metamodel is called the **static model** ($M_{stat}$), e.g. a concrete Petri net structure.

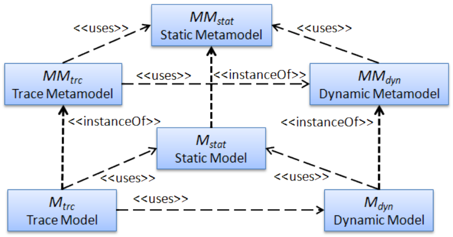

Figure 1. Metamodels and instances of a dynamic model

Next, a **dynamic metamodel** $MM_{dyn}$ uses and extends the static metamodel $MM_{stat}$ for storing information related to dynamic behaviour (e.g. current state, value, configuration) of a structural element. The **dynamic model** ($M_{dyn}$) is an instance of the $MM_{dyn}$, e.g. the current marking of a given Petri net place.

Finally, a **trace metamodel** ($MM_{trc}$) is defined for the language to represent simulation runs of the $M_{dyn}$. The $MM_{trc}$ *uses* the $MM_{dyn}$ for recording how the dynamic model changed and the $MM_{stat}$ for describing which static element is concerned. A **trace model** ($M_{trc}$) is an instance of the $MM_{trc}$, e.g. the sequence of fired transitions of a Petri net. The $M_{trc}$ describes the changes of $M_{dyn}$, therefore it is represented as a *change model* in terms of [10].

While execution traces are traditionally interpreted as a sequence of elementary operations, in the current paper, we use hierarchical trace models consisting of *micro steps* (atomic operations on $M_{dyn}$) and *macro steps* (complex operations on $M_{dyn}$), which is compliant with recent approaches [11] to define semantics for big-step DMLs like statecharts.

Given the set of micro steps as terminal symbols $T$, the hierarchy of macro steps as non-terminal symbols $NT$, and a grammar $Gr$ implied by the operational semantics description of the language (see below), a trace model can be formally interpreted as the abstract syntax tree $AST$ of a well-formed sentence of the grammar $GR$.

### B. Operational semantics and traces for dynamic models

The simulation of a DML is performed in accordance with the *operational semantics* of the language defined by simulation rules. In our framework we assume that simulation rules are defined as intra-model transformations illustrated in Fig. 2 (see also [12]–[14]).

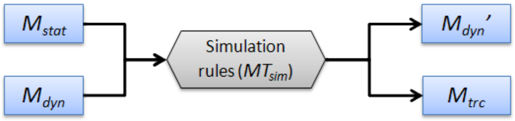

Figure 2. Simulation and trace generating transformation

The execution of a rule in the transformation $MT_{sym}$ : $(M_{stat}, M_{dyn}) \xrightarrow{\Delta} M_{dyn}'$ modifies the $M_{dyn}$ by also taking into account $M_{stat}$ and results in a new $M_{dyn}'$. During a simulation run, the changes of the dynamic model are recorded as a sequence of micro steps as part of the derived trace model $M_{trc}$. Furthermore, the hierarchy of macro steps in $M_{trc}$ is in direct correspondence with the transformation rules fired during the simulation run. Therefore, we assume that the simulation rules $MT_{sym}$ imply a grammar $GR^{,}$ which defines well-formed execution traces.

### C. Forward model transformation with traceability links

We assume the existence of a unidirectional structural model transformation $MT_{src2trg}$ (see Fig. 3) which generates a static target model $M^{trg}_{stat}$ from a given static source model $M^{src}_{stat}$. This $MT$ is also responsible for deriving the initial state of $M^{trg}_{dyn}$ from the initial state of $M^{src}_{dyn}$.

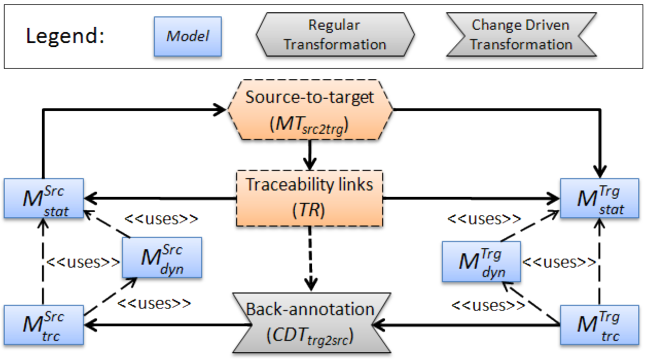

Figure 3. Forward model transformation and back-annotation

We also assume that this transformation generates **traceability links** ($TR$) between the source and target models in order to record the structural correspondence between the model elements. As we only rely upon these traceability links for back-annotation, any kind of forward model transformation approaches and tools can be used.

### D. Back-annotation of dynamic execution traces

In the paper, back-annotation of a target DML to a source DML is defined as a transformation $CDT_{trg2src}$ which is able to generate the $M^{src}_{trc}$ from an arbitrary $M^{trg}_{trc}$ if such source trace exists. The $CDT_{trg2src}$ makes use of the $TR$ to identify corresponding elements in source and target models.

Given that the traces contain model changes, we propose to define the $CDT_{trg2src}$ as a change driven model transformation [8] (see the overview in Sec. IV-A).

In many practical cases, no formal operational semantics is available for the source DML (e.g. in case of UML or BPEL). Or in other terms, their formal semantics is defined in denotational way by mapping them to a formal target DML like Petri nets [15], [16]. It is worth pointing out that our back-annotation framework only assumes that

- a target trace $M^{trg}_{trc}$ is made available by some analysis tool, which is compliant with the formal operational semantics $GR^{trg}$ of the target (analysis) DML,
- the macro steps of the source (design) DML can be identified based on an informal behavioural description.

### E. Challenges for back-annotation

There are various challenges are likely to arise when back-annotating simulation traces between a given source and target language.

*Spurious target traces:* In most cases in practice, the $MT_{src2trg}$ transformation defines an abstraction of the source model to derive the target model. This implies that $M^{trg}_{dyn}$ simulates $M^{src}_{dyn}$, i.e. for every source trace $M^{src}_{trc}$ there is a corresponding target trace $M^{trg}_{trc}$, but not necessarily vice versa.

In the ideal case when the source language has formal operational semantics, one can check the compatibility of the derived source trace $M^{src}_{trc}$ with respect to the semantics $GR^{src}$ by using model checking techniques for DMLs [17]. If the formal semantics $GR^{src}$ is undefined, it becomes the role of back-annotation $CDT_{trg2src}$ to detect if a target trace is spurious or not.

*Mismatch between trace granularity:* In many practical cases, there is a mismatch between the granularity of traces in the source and target models (see also Fig. 8) due to semantic difference between the DMLs. In fact, many kinds of mappings are possible in $CDT_{trg2src}$ to relate macro and micro steps of the source and target DMLs:

- **1-to-1**: One step in the target trace $M^{trg}_{trc}$ may correspond to one step in the source trace $M^{src}_{trc}$.
- **1-to-n**: Several steps in the target trace $M^{trg}_{trc}$ may correspond to one step in the source trace $M^{src}_{trc}$ (or vice versa).
- **m-to-n**: Mappings can be defined between sequences or groups of steps in $M^{trg}_{trc}$ and $M^{src}_{trc}$.
- **Ignore**: Steps in the target trace $M^{trg}_{trc}$ may be completely ignored during back-annotation.
- **Reorder**: The ordering of certain target steps in $M^{trg}_{trc}$ may not coincide with the ordering of the source steps in$M^{src}_{trc}$, which requires reordering by $CDT_{trg2src}$.

*Interleaving of simulation steps:* This last problem of reordering leads to the generic problem caused by the interleaving of simulation steps. In the context of back-annotation, interleaving occurs if the sequence of macro steps in $M^{trg}_{trc}$ cannot be mapped into $M^{src}_{trc}$ by simply slicing the sequence into subsequences. Simulation steps $NT^1_{trg}, NT^2_{trg}, NT^3_{trg}$ are interleaving if $NT^1_{trg}, NT^3_{trg}$ are mapped to $NT^1_{src}$ and $NT^2_{trg}$ is mapped to $NT^2_{src}$.

In our framework, interleaving simulation steps will be handled using *change patterns* of change driven model transformations which are able to recognise the aggregated result of interleaving steps as they are persisted in the change model of the $M^{trg}_{trc}$.

## III. Definition of dynamic modeling languages

In our paper we motivate the back-annotation problem of analyzing BPEL processes using a simulator. This back-annotation needs to derive an execution trace of the business process from the simulation run of the Petri net.

In this section, we first give a short introduction to the back-annotation problem itself (Sec. III-A), and then the metamodels of the DMLs of BPEL and Petri nets (Sec. III-B), followed by the macro steps of the two DMLs (Sec. III-C), the operational semantics of the target Petri nets (Sec. III-D), and the traceability links derived by the BPEL-to-Petri net transformation (Sec. III-E),

### A. *Motivating scenario*

As the BPEL standard uses plain English for specification, a formal analysis of BPEL necessitates a mapping to a formal language, such as Petri nets, abstract state machines, process algebras, etc. [18] in order to precisely define the dynamic behavior of BPEL processes. In our motivating example we use a Petri net based approach [15], [16], which maps BPEL structures into Petri net subnets that can be embedded and combined using interface places. Fig. 4 shows the overview of the motivating example.

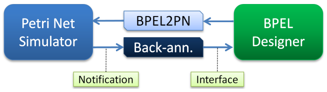

Figure 4. Motivating Scenario

The BPEL process is created in a graphical editor, and a transformation (*BPEL2PN*) is executed to generate the formal Petri net model which is subject to formal analysis using a Petri net simulator (or other analysis tool). In the paper, we show how to provide back-annotation assuming that (1) the simulator reports the changes in the Petri net model (e.g. by notification support) and (2) the BPEL designer has an interface for setting the actual state of a process instance.

### B. *Static and dynamic metamodels*

*1) Business Process Execution Language:* BPEL is an industrial standard that defines an executable orchestration language for specifying business processes based on Web Services. During process execution, external Web Services are called. The building blocks of BPEL processes are the description of the parties interacting during the process, the data variables used by the process and the activities describing the behavior of the process. The activities can be grouped into two categories: *basic* and *structured* activities. Basic activities are responsible for communicating with external Web Services and for data manipulation of internal variables of the business process. Structured activities implement the control flow of business processes including activities like ”receive”, ”reply”, ”invoke”, ”assign”, ”sequence”.

The **BPEL metamodel** (see Fig. 5) describes the abstract syntax of the BPEL language. Nodes (e.g. *Activity*) of the metamodel are called **classes**. A class may have **attributes** (e.g. the *variable* in *Receive*) that define some kind of properties of the specific class. **Inheritance** may be defined between classes, which means that the inherited class has all the properties its parent has, but it may further contain some extra attributes (e.g. elements in BPEL inherit from *ExtensibleElements*). **Associations** like *contains* define connections between classes. Furthermore, we use traceability edges (denoted by dashed lines in instance models) connecting source and target model nodes.

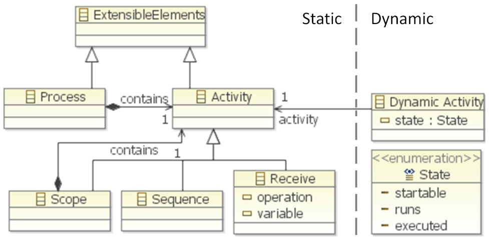

Figure 5. BPEL metamodel fragment

In order to model process instances in execution we define additional dynamic information for BPEL elements, e.g. *Dynamic Activity* is associated with an activity and has a dynamic state. This state can be *startable*, *runs* and *executed* for all activities, but further refinement is possible with additional states for complex structures (such as scopes).

*2) Petri Nets:* Petri nets (in our case, Place/Transition nets with inhibitor arcs, see the metamodel in Fig. 6) are widely used to formally capture the dynamic semantics of concurrent systems. Petri nets are bipartite graphs, with two disjoint sets of nodes: Places and Transitions. Places may contain an arbitrary number of Tokens. The places and transitions of the Petri Net are grouped into subnets to help the identification of correlated elements.

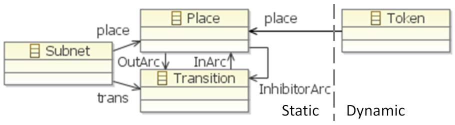

Figure 6. Petri net metamodel

A token distribution defines the state of the modeled system. The state of the net can be changed by firing enabled transitions. A transition is enabled if each of its input places contains at least one token and no place connected with an inhibitor arc contains a token (if no arc weights are considered). When firing a transition, we remove a token from all input places (connected to the transition by Input Arcs) and add a token to all output places (as defined by Output Arcs).

### C. Trace metamodels as change commands

We define trace metamodels in Fig. 7 as change commands (in the sense of [8]) to represent the different macro steps of Petri nets and BPEL. These commands are specialized from the *ChangeCommand* type of the general change metamodel. The specific commands are defined in accordance with the simulation step types.

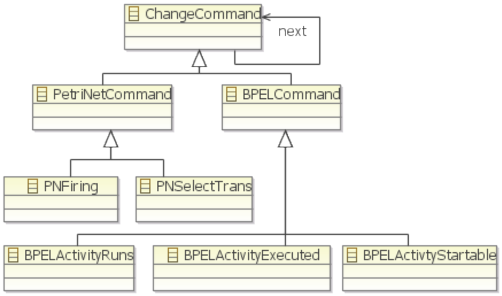

Figure 7. Change commands for Petri Nets and BPEL

As illustrated in Fig. 8 the trace model is a hierarchy of macro and micro steps of the two DMLs. Steps on the same hierarchy level are ordered to ensure that operations were carried out in the given sequence during simulation run.

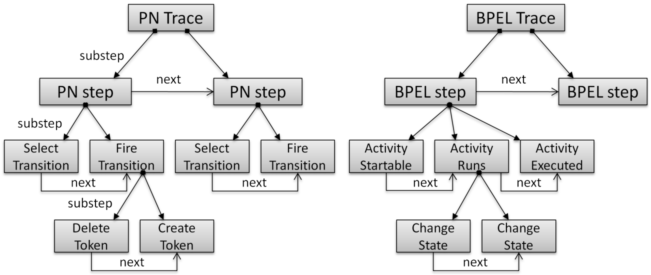

Figure 8. Step hierarchy for Petri nets and BPEL

Top level macro steps on the left side of Fig. 8 represent steps which are visualized (either textually or graphically) in the Petri Net simulator (**PN step**). These steps are a pair of macro steps representing the selection of an enabled transition and the firing of the selected transition. The substeps of firing a transition consist in deleting and creating appropriate number of Token elements in $M^{PN}_{dyn}$).

On right side of Fig. 8, the top level steps of BPEL are events of the business process which include macro steps corresponding to the states (startable, runs, completed) of the activities of the process. These in turn include micro steps representing changes of the activity states in the dynamic model. Note that the BPEL standard does not define a formal operational semantics therefore these steps had to be identified by us prior to the back-annotation.

### D. Operational semantics for simulation

The simulation of dynamic models is enabled by specifying their operational semantics using model transformation rules. In this section we briefly revisit the approach of [14] to formally define a simulator for Petri nets using the VIATRA2 transformation language [19]. However, the overall approach can be used to define the behavior of other discrete event-based languages.

The dynamic semantics of a DML is described by *simulation rules* $R = (EC, AS)$ where $EC$ is an *enabledness condition* and an $AS$ is a *command sequence*, which describes model manipulation.

*Enabledness condition:* The enabledness condition of a simulation rule decides its applicability on a given model and they are formally represented as *graph patterns*. These patterns define conditions and constraints that have to be fulfilled by a part of the model. To compose complex conditions, graph patterns can call each other in a positive and negative way. A *negative application condition* (NAC) prescribes contextual conditions for the original pattern which are forbidden in order to find a successful match.

```
pattern transitionFireable(T)=
{transition(T);
 neg pattern notEnabled(T)={
  place(P);outArc(OutArc,P,T);
  neg pattern placeToken(P)={
   token(Tok);
   token.place(Rp, Tok, P);}}
 or { // inhibited pattern
  place(P);token(Token);
  inhibitorArc(OutArc, P, T);
  token.place(Rp, Tok, P);}}
```

Listing 1. Enabledness condition for firing a transition

The enabledness condition for a Petri net transition can be expressed using a *graph pattern* as shown in Listing 1. This pattern uses nested negative application conditions to express that a transition $T$ is enabled if every input Place $P$ connected to the Transition instance has at least one Token associated and no input Places connected with an inhibitor arc contains a token. In this example, embedded NACs are used to express universal quantification with double negation of existence.

*Commands for firing a transition:* Complex model manipulation commands of simulation rules can be expressed by graph transformation [20] provides a high-level rule and pattern-based manipulation language for graph models. Complex action sequences can be assembled by abstract state machines [21], which drive the core model manipulation commands of graph transformation rules.

Listing 2 demonstrates a sequence actions to simulate the firing of a selected Petri net transition $T$, which consists in the removal of tokens from input places and the addition of tokens to output places.

```
rule fireTransition(in T) = seq {
  /* remove tokens from all input places */
  forall Place with find inputPlace(T, P)
   do apply removeToken(T,Place);//GT rule invocation
  /* add tokens to all output places */
  forall Place with find outputPlace(T, P)
   do apply addToken(T, Place);}
```

Listing 2. Simulation rule transition firing

### E. Static traceability between BPEL and Petri Nets

The forward (structural) model transformation from BPEL to Petri nets can be defined using an arbitrary model transformation tool, thus we omit its detailed discussion from the paper. We only assume that the transformation provides static traceability by storing the structural correspondence between the two models (e.g. linking the generated Petri net to the BPEL process). Fig. 9 shows a generic mapping which depicts the typical structure of traceability links in the running example. For each BPEL element, a corresponding subnet is derived in the target Petri net model, which contains at least five places for different execution phases called *initial, stop, stopped, failed* and *final*.

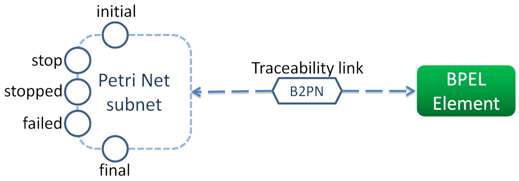

Figure 9. Traceability mapping

## IV. Change-driven mapping of simulation traces

The trace mapping transformation derives the $M^{src}_{trc}$ representing a BPEL process execution from the Petri net simulation run persisted as $M^{trg}_{trc}$. In this section we detail the definition of this transformation on an example from our motivating scenario. First, a brief introduction is given on change driven model transformations, followed by a list of challenges in the PN-to-BPEL back-annotation. The challenges are addressed using a mapping transformation rule. The application of this rule is illustrated on a step-by-step example. Finally, we give a short insight on the implementation details.

### A. Change-driven model transformations

Change driven model transformations [8] are model transformations which consume changes of the source model $M_A$ and produce changes for the target model $M_B$. In order to perform change driven transformations, some form of traceability information $TR$ should also be available between the two models(see Fig. 10).

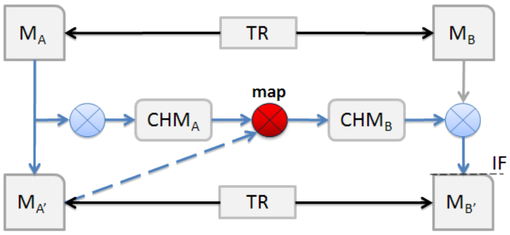

Figure 10. Change-driven transformations

Essentially, a change driven transformation rule (CD rule) is *enabled by change in the source model*. The actual change representation can be of different nature, e.g. a sequence of model manipulation operations or a change delta. When a change is detected, the effects of a change driven rule are executed to create or remove elements of the target model (and possibly to create or remove some traceability links between the two models).

The changes are detected using special graph patterns also called *change patterns*. Such patterns can be (1) *appear patterns* which detect changes in the model related to new matches for a given pattern while (2) *disappear patterns* detect that a preexisting match for a pattern is no longer present in the model. These patterns can be specified in the *guard* of a CD rule.

### B. Challenges for the PN-to-BPEL back-annotation

The mapping between the Petri net and BPEL traces is, essentially, that a sequence of transition firings will correspond to one or more macro steps for BPEL activity execution phase changes. However, several challenges arise in the actual definition of this complex mapping which is in line with challenges in Sec. II-E.

*1) Identify corresponding BPEL activity:* When mapping successive Petri net steps, the activity (scope) of the corresponding BPEL step has to be identified. This identification is impossible based exclusively on the Petri net, thus the trace mapping incorporates static traceability information.

*2) Identify BPEL step type:* The trace mapping process needs to identify the type of the BPEL step which directly refers to the state change of the corresponding activity (startable, runs, executed).

*3) Mismatch between trace granularity:* Trace granularity can be different in two DMLs when (1) several Petri net steps correspond to one BPEL step in case of inner transitions of a subnet and (2) one Petri net step represents several BPEL steps when the transition of a structural activity subnet fires (e.g. several activity becomes startable).

*4) Handle interleaving steps:* Interleaving of Petri net simulation steps may occur in the example when dealing with parallel execution of BPEL activities. However Petri net transition which belong to the BPEL branch are not necessarily fired in direct succession.

### C. Mapping traces with change-driven rules

The back-annotation of Petri Net steps to BPEL process execution can be accomplished by providing a mapping between the traces. We use the motivating example to illustrate the trace mapping using change-driven rules. Change-driven rules recognise the appearance of Petri Net specific change commands in the change model and create corresponding BPEL change commands based on the structure of the models and the static traceability information.

*1) Identify corresponding BPEL activity:* The subnets which encapsulate transitions and places correspond to one BPEL element. Therefore, the traceability links which associate subnets to BPEL activities and were created during the forward transformation are used for identifying the appropriate activity. In the example, the corresponding BPEL activity (*BA*) can be found by using the link (*B2PN*) that connects it to the subnet (*SN*) in which the transition is defined (*BPELActivityForTransition* graph pattern in Listing 3).

```
pattern BPELActivityForTransition(Tr,BA,SN) = {
 Transition(Tr);
 Subnet(SN);
 Subnet.trans(Rt,SN,TR);
 TRLink(B2PN);
 TRLink.subnet(Rs,B2PN,SN);
 BPELActivity(BA);
 TRLink.activity(Ra,B2PN,BA);}
```

Listing 3. Identify corresponding BPEL activity

*2) Identify BPEL step type:* The BPEL2PN transformation defines interface places in the Petri net subnets which represent execution states that are common for all BPEL activities. In order to derive the appropriate BPEL step type, we use these interface places. The existence of such places allows a more generic implementation of the mapping, instead of requiring the definition of multiple mapping rules separately for each activity type. The mapping for the three common BPEL step types are given using graph patterns defined on the input and output places of the transitions.

- **Activity startable** An activity is startable when the *initial* place in its subnet has a token. A transition firing that puts a token on this place is mapped to a *BPELActivityStartable* change command.
- **Activity runs** An activity starts its execution when the firing of a transition inside the corresponding subnet removes a token from the *initial* place and does not place token in the *final, stopped* or *failed* places. Such steps are mapped to *BPELActivityRuns* commands.
- **Activity executed** An activity finishes its execution when one of the *final, stopped* or *failed* places in its subnet have a token. A transition firing that puts a token on one of these place is mapped to a *BPELActivityExecuted* change command.

An example back-annotation rule is defined for transition firing corresponding to the *Activity runs* case. Listing 4 is the VIATRA2 representation of the CD rule for this mapping. It starts with the *guard* pattern that specifies the change command that triggers the application of the CD rule. The macro step of the Petri Net representing the firing of a transition *Tr* appears as a change command *PNF*. The precondition deals with finding the corresponding BPEL activity *BA* for the transition that was fired and checks that the transition matches the *Activity runs* step type. Finally, the *BPELActivityRuns* change command *BAR* for the BPEL event is created in the action part of the back-annotation rule.

```
cdrule bpelActivityStarts() = {
 guard change pattern PNFiringCreated(PNF) = {
  appear pattern() = { PNFiring(PNF);}}
 precondition pattern BPELActivity(BA) = {
  Transition(Tr);
  trans(Rt,PNF,Tr);
  // find BPEL activity using traceability link
  find BPELActivityForTransition(Tr,BA,SN);
  // check inbound place initial
  find HasInputPlaceInitial(Tr,SN,Pi);
  // outbound not final, stopped, failed
  neg find HasOutputPlaceFinished(Tr,SN,Po);}
 action {
  // create macro step for BPEL
  new(BPELActivityRuns(BAR));
  new(activity(Ra,BAR,BA));}}
```

Listing 4. Trace mapping change driven rule

*3) Mismatch between trace granularity:* When we map several Petri net steps to one BPEL step, we use "lookup" graph patterns to check the input and output places of the firing transition (*TR*) for a specific type of interface place. Listing 5 shows a graph pattern (*HasInputPlaceInitial*) that searches for *initial* places among the input places of *TR*. The Petri net step is only mapped if the "lookup" patterns defined in the precondition of the mapping rule match. Otherwise the internal transitions of subnets are ignored.

```
pattern HasInputPlaceInitial
   (Tr,SN,P) = {
 Transition(Tr); Place(P);
 OutArc(Ro,P,Tr);
 Subnet(SN); Subnet(S2);
 Subnet.trans(Rt,SN,Tr);
 Subnet.initial(Ri,S2,P);}
```

Listing 5. Input initial place "lookup" graph pattern

When one Petri net step is mapped to several BPEL steps, the number of affected BPEL activities are derived from the number of different subnets in the interface places among the input and output places of a transition. This case appears in the *BpelActivityStartable* mapping, we included the essence of the solution in Listing 6. The action part of the CD rule is altered to find the activities for every corresponding subnet (*HasOutputPlaceInitial* and *BPELActivityForPlace*).

```
cdrule bpelActivityStarts() = {
[...]
action {
 // handle mapping to more BPEL steps
 forall P with find
   HasOutputPlaceInitial(Tr,SN,P) do
  choose BA,S2 with find
    BPELActivityForPlace(P,BA,S2) do
 [...]}
```

Listing 6. Create severla BPEL steps for one Petri net step

*4) Handle interleaving steps:* The Petri net steps in the simulation interleave when transitions in different subnets are enabled and the ones in the same subnet are not fired in succession. On the other hand, the transition firings contained within the same subnet will not interleave, as they model the execution of a single BPEL activity. Therefore, the subnets can be used to partition Petri net steps. In our approach, change patterns abstract the processing of sequential execution by filtering the Petri net steps based on subnets (as shown in Listing 7).

```
change pattern(PNF1,PNF2) = {
 appear pattern() = {
  PNFiring(PNF1);
  PNFiring(PNF2);}
 find TrSameSubnet(PNF1,PNF2);}
```

Listing 7. Filter steps by subnet Change pattern

*"Spurious counter-examples"* are possible due to the non-interpreted data modeling (e.g. no concrete variable values) and the control-flow abstraction of BPEL loops.

*5) Change driven rule execution: a step-by-step example:* Fig. 11 illustrates the execution of the CD rule. The rule is triggered when the command representing the firing of a transition appears in the change-driven transformation framework (*Step 1*). Next the traceability information (B2PN) is used to find the BPEL activity (BA) corresponding to the subnet which contains the transition (*Step 2*). Then the new command (BAR) is generated according to the matching step type (*Step 3*).

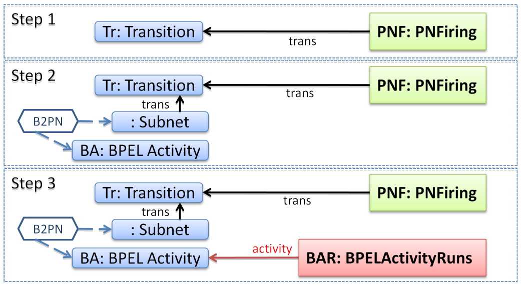

Figure 11. Example execution

### D. Implementation details

The metamodels for BPEL and Petri nets, along with the trace generator and mapping transformations are implemented in the VIATRA2 model transformation framework [22]. Change driven model transformations can be both developed and executed in the VIATRA2 framework over the VPM model repository. Furthermore, the Petri net simulator presented in [14] is used as it is implemented in VIATRA2.

The engineering tool used as a front-end of our back-annotation framework is the Eclipse BPEL Designer tool [23]. This tool provides graphical editing support for developing BPEL business process, although it lacks any representation options for a dynamic process state. Therefore we extended the tool using the Eclipse plug-in architecture with an interface for changing the dynamic state of the process elements and a component that updates the graphical representation of the BPEL elements.

We also implemented a similar back-annotation transformation for a different back-end tool. In that case counter-examples of the SAL [24] model checking framework were modeled as traces and mapped back to BPEL process execution [25]. Similarly, the process execution is presented using the BPEL Designer tool here as well.

## V. RELATED WORK

Now an overview is given on related approach from the domains of model-based simulation and back-annotation.

*Model-based simulation and simulation traces:* Simulation of domain-specific models using model transformations has been described among others in [12]–[14]. [13] uses Triple Graph Grammar rules for defining the semantics of the simulator, [14] uses the ASM rules and graph patterns available in the VIATRA2 framework for the simulator specification. Both approaches simulate the execution of dynamic models, although neither record the trace of simulation.

In [12] the semantical correctness and completeness of model transformations based on simulation specifications are investigated. In the approach, these properties are ensured by a one-to-one mapping between the simulation rules of the original and the generated simulator. Our proposed approach deals with hierarchical traces and arbitrary mapping between pre-existing semantics (i.e. only the target model is generated, not the simulation rules).

Execution traces are, on the other hand, used in many cases, for understanding distributed systems [26], recovering behaviour [27]. Dynamic traces were defined for individual languages such as UML sequence diagrams [28], UML Activity Diagrams [29].

*Back-annotation:* Many approaches exist [2], [30]–[34] that define model transformations from a source language to target language for V&V purposes with back-annotation support dedicated to the transformation problem.

In [30] an approach is presented for the back-annotation of feature models. Feature models be translated into propositional formulae for analysis and configuration purposes. Czarnecki et al. show that the opposite translation problem is also solvable and feature models can be extracted from these formulae. Note that, contrary to their work, our approach is defined on dynamic simulation traces and not static models.

[31] uses traceability links of the transformation which generates Alloy models from UML. The back-annotation transformation is automatically generated based on these traceability links using a QVT-based implementation. Here the back-annotation is supported for static model instances, and not for execution traces of dynamic UML models models, which is a conceptually easier setup than in our current work.

In [32] a conversion tool is presented which is able to generate Message Sequence Charts (MSCs) from the trails (i.e. traces) of the SPIN model checker tool. Although it works on execution traces, the approach is only back-annotation in the sense that the trails may be the result of model checking on SPIN models created from design models. However the conversion is restricted to MSCs and does not back-annotate the results to a specific design model.

In [2] Gilmore et al. present a software tool platform for security and performance analysis of systems. UML models created in the tool are transformed in the form of process calculi descriptions and after analysis the results are reflected back into a modified version of the input UML model. They also stress the importance of reflection (i.e. back-annotation) in their approach. Their approach primarily supports the back-annotation of quantitative and qualitative parameters to the source UML model, while execution traces are again represented as UML sequence diagrams.

In [33] Foster et al. describe a model-based approach to verify compositions of web services. They use the LTSA tool suite [35] for model checking a BPEL process transformed into FSP description. Similarly to [32] they uses MSCs as a target formalism for back-annotating the results (which are process traces) which is, in this way, different from the original design formalism (BPEL).

Our first results on back-annotation was presented in [25], where BPEL processes are verified by transforming them to SAL [24]. While the tool includes back-annotation support, the paper itself does not present (1) the generic back-annotation framework (of Sec. II), (2) it uses plain (non-hierarchical) trace models, and (3) it uses regular model transformations for trace mapping which resulted in more complex back-annotation rules.

None of these approaches allows to replay the execution traces of analysis results *directly in the original design model*. The most advanced back-annotation techniques is reported in [34], where triple graph patterns to overcome the 1-to-1 restriction on back-annotation. However, this back-annotation approach still mostly provides a structural (and not trace) mapping, thus hierarchical trace models with micro and macro steps are not supported. Moreover, the use of change driven transformations allows a more succinct formulation of trace mappings in our case.

## VI. Conclusions and Future Work

In our paper, we discussed how a simulation run of a formal analysis model can be persisted in trace models and mapped back to the design model using change driven model transformations for back-annotation. We presented the challenges and concepts of a generic back-annotation framework for simulation traces of discrete event-based languages, and detailed our approach on the motivating scenario of BPEL verification using Petri nets.

The main practical novelty of our generic back-annotation framework is that it allows to replay the execution traces of analysis results directly in the original design model, which offers a very intuitive solution from the viewpoint of systems and services engineers. Furthermore, we can back-annotate traces even if only the macro steps of the source design language are precisely defined (but not its full operational semantics). Finally, complex mismatches are allowed between the traces of the source and target dynamic models.

In the paper, we will illustrated our back-annotation approach on BPEL and Petri nets. However, there are many other design and analysis languages and tools which can be target for our back-annotation framework. Natural candidates for design languages include the Unified Modeling Language (UML), Systems Modeling Language (SysML), Modeling and Analysis of Real-time and Embedded systems (MARTE), Architecture Analysis and Design Language (AADL) and Business Process Model and Notation (BPMN). The analysis formalisms in our scope are *discrete event-based dynamic modeling languages* like labeled transition systems, multiple phased systems, hierarchical state machines, process calculi, data flow systems and high-level Petri net formalisms. Several actual back-annotation mappings are part of our ongoing work.

In the future we will focus on defining an algorithm for creating the trace generator transformation from the simulation rules and plan to investigate the possibility of deriving the trace mapping from the structural model transformation and the traceability model.

## References

[1] M. Wirsing, M. M. Hölzl, L. Acciai, F. Banti, A. Clark, A. Fantechi, S. Gilmore, S. Gnesi, L. Gönczy, N. Koch, A. Lapadula, P. Mayer, F. Mazzanti, R. Pugliese, A. Schroeder, F. Tiezzi, M. Tribastone, and D. Varró, "SensoriaPatterns: Augmenting Service Engineering with Formal Analysis, Transformation and Dynamicity," in *ISoLA*, ser. Communications in Computer and Information Science, T. Margaria and B. Steffen, Eds., vol. 17. Springer, 2008, pp. 170–190.

[2] M. Buchholtz, S. Gilmore, V. Haenel, and C. Montangero, "End-to-end integrated security and performance analysis on the DEGAS Choreographer platform," in *Formal Methods 2005*, ser. LNCS, 2005.

[3] A. Bondavalli, M. D. Cin, D. Latella, I. Majzik, A. Pataricza, and G. Savoia, "Dependability analysis in the early phases of UML-based system design," *Comput. Syst. Sci. Eng.*, vol. 16, no. 5, pp. 265–275, 2001.

[4] "The assert-project : Automated proof-based System and Software Engineering for Real-Time Systems (ASSERT)," http://www.assert-project.net/.

[5] W. Herzner, B. Huber, A. Balogh, and P. Csertan, "The DECOS tool-chain: Model-based development of distributed embedded safety-critical real-time systems," *ERCIM News*, vol. 67, pp. 22–4, 2006.

[6] A.-E. Rugina, K. Kanoun, and M. Kaâniche, "The adapt tool: From aadl architectural models to stochastic petri nets through model transformation," *CoRR*, vol. abs/0809.4108, 2008.

[7] OASIS, "Web Services Business Process Execution Language Version 2.0 (OASIS Standard)," 2007, "http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html".

[0] All related papers of the authors are available at http://home.mit.bme.hu/~hegedusa/sefm_pubs.html.

[8] I. Ráth, G. Varró, and D. Varró, "Change-driven model transformations," in *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings*, ser. LNCS, vol. 5795. Springer, 2009, pp. 342–356.

[9] A. Hegedüs, R. István, and D. Varró, "Back-annotation framework for Simulation Traces of Discrete Event-based Languages," BME, Tech. Rep., April 2010, http://home.mit.bme.hu/~hegedusa/publist.html.

[10] G. Bergmann, I. Ráth, A. Ökrös, and D. Varró, "Change-Driven Model Transformations: Taxonomy and Language," *Journal of Software and Systems Modeling*, 2010, submitted.

[11] S. Esmaeilsabzali and N. A. Day, "Prescriptive semantics for big-step modelling languages," in *Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Proceedings*, ser. LNCS, D. S. Rosenblum and G. Taentzer, Eds., vol. 6013. Springer, 2010, pp. 158–172.

[12] H. Ehrig and C. Ermel, "Semantical correctness and completeness of model transformations using graph and rule transformation," in *ICGT*, ser. LNCS, vol. 5214. Springer, 2008, pp. 194–210.

[13] J. de Lara and H. Vangheluwe, "Translating model simulators to analysis models," in *FASE*, ser. LNCS, J. L. Fiadeiro and P. Inverardi, Eds., vol. 4961. Springer, 2008, pp. 77–92.

[14] I. Ráth, D. Vágó, and D. Varró, "Design-time Simulation of Domain-specific Models By Incremental Pattern Matching," in *2008 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2008.

[15] C. Stahl, "A Petri Net Semantics for BPEL," Humboldt-U. zu Berlin, Institut für Informatik, Tech. Rep. 188, 2005.

[16] N. Lohmann, "A feature-complete Petri net semantics for WS-BPEL 2.0," in *Services and Formal Methods, Forth International Workshop, WS-FM 2007*, pp. 28–29.

[17] D. Varró, "Automated formal verification of visual modeling languages by model checking," *Journal of Software and Systems Modeling*, vol. 3, no. 2, pp. 85–113, May 2004.

[18] F. V. Breugel and M. Koshkina, "Models and Verification of BPEL," 2006, "http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf".

[19] D. Varró and A. Balogh, "The model transformation language of the VIATRA2 framework," *Science of Computer Programming*, vol. 68, no. 3, pp. 214–234, October 2007.

[20] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, Eds., *Handbook on Graph Grammars and Computing by Graph Transformation*. World Scientific, 1999, vol. 2: Applications, Languages and Tools.

[21] E. Börger and R. F. Stärk, *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[22] D. Varró and A. Pataricza, "VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML," *Software and Systems Modeling*, vol. 2, no. 3, pp. 187–210, 2003.

[23] "Eclipse BPEL Designer," http://www.eclipse.org/bpel/.

[24] S. Bensalem, V. Ganesh, Y. Lakhnech, C. M. noz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari, "An overview of SAL," in *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, C. M. Holloway, Ed., Hampton, VA, jun 2000, pp. 187–196.

[25] L. Gönczy, A. Hegedüs, and D. Varró, "Methodologies for Model-Driven Development and Deployment: an Overview," in *Rigorous Software Engineering for Service-Oriented Systems: Results of the SENSORIA project on Software Engineering for Service-Oriented Computing*, M. Wirsing, Ed. Springer-Verlag, 2010, to appear.

[26] J. Moe and D. A. Carr, "Understanding Distributed Systems via Execution Trace Data," in *IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2001, p. 60.

[27] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge, "Recovering Behavioral Design Models from Execution Traces," in *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2005.

[28] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue, "Extracting Sequence Diagram from Execution Trace of Java Program," in *IWPSE '05: Proceedings of the Eighth International Workshop on Principles of Software Evolution*. Washington, DC, USA: IEEE Computer Society, 2005.

[29] A. Sela, M. Fritzsche, A. Zherebtsov, J. Johannes, and A. Terekhov, "MODELPLEX Deliverable D4.2a: Metamodels for simulation," IBM, Tech. Rep., Decembre 2007.

[30] K. Czarnecki and A. Wasowski, "Feature Diagrams and Logics: There and Back Again," in *SPLC '07: Proceedings of the 11th International Software Product Line Conference*. Washington, DC, USA: IEEE Computer Society, 2007.

[31] S. M. A. Shah, K. Anastasakis, and B. Bordbar, "From UML to Alloy and back again," in *MoDeVVa '09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*. ACM, 2009, pp. 1–10.

[32] T. Kovše, B. Vlaovič, A. Vreže, and Z. Brezočnik, "Spin Trail to Message Sequence Chart Conversion Tool," in *The 10th International Conference on Telecomunications*, Zagreb, Croatia.

[33] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "WS-Engineer: A Model-Based Approach to Engineering Web Service Compositions and Choreography," in *Test and Analysis of Web Services*. Springer Berlin Heidelberg, 2007, pp. 87–119.

[34] E. Guerra, J. de Lara, A. Malizia, and P. Díaz, "Supporting user-oriented analysis for multi-view domain-specific visual languages," *Information & Software Technology*, vol. 51, no. 4, pp. 769–784, 2009.

[35] J. Magee and J. Kramer, *Concurrency: State Models & Java Programs*. New York, USA: John Wiley & Sons, Inc., 1999.