D.3.3 ALGORITHMS FOR INCREMENTAL REQUIREMENTS MODELS EVALUATION AND TRANSFORMATION

Michela Angeli (UNITN), Gábor Bergmann (BME), Fabio Massacci (UNITN), John Mylopoulos (UNITN), Bashar Nuseibeh (OU),Federica Paci (UNITN), Bjornar Solnaug (SINTEF), Thein Than Tun(OU),Yijun Yu (OU),Dániel Varró (BME)

Document information

Document Number	D.3.3
Document Title	Algorithms for Incremental Requirements Models Evaluation and Transformation
Version	1.9
Status	Final
Work Package	WP 3
Deliverable Type	Report
Contractual Date of Delivery	31 January 2011
Actual Date of Delivery	28 January 2011
Responsible Unit	UNITN
Contributors	OU, UNITN, BME, THA
Keyword List	
Dissemination level	PU

Document change record

Version	Date	Status	Author (Unit)	Description
0.1	25October 2010	Draft	Federica Paci (UNITN)	Outline of the deliverable
0.2	31 October 2010	Draft	Gábor Bergmann (BME)	Added first version of BME's section
0.3	15 November 2010	Draft	Federica Paci (UNITN)	First Draft
0.4	23 November	Draft	Federica Paci (UNITN)	Added sections 2, 3 and 4 about applicability to ATM case study
0.5	25 November 2010	Draft	Gábor Bergmann (BME)	Added content to Section 4, ATM-specific example still missing
0.6	1 December 2010	Draft	TheinTun (OU)	Added content of Section 2, including the ATM example.
0.9	5 December 2010	Draft	Gábor Bergmann (BME)	Expanded the text of Section 4.4, cleanups overall.
1.0	6 December 2010	Draft	Gábor Bergmann (BME)	Elaborated BME Section with sample codes and ATM examples
1.1	10 December 2010	Draft for Review	Federica Paci, John Mylopoulos (UNITN)	Added Introduction, and Section 3
1.2	16 December 2010	Draft for Review	Bjornar Solhaug	Review

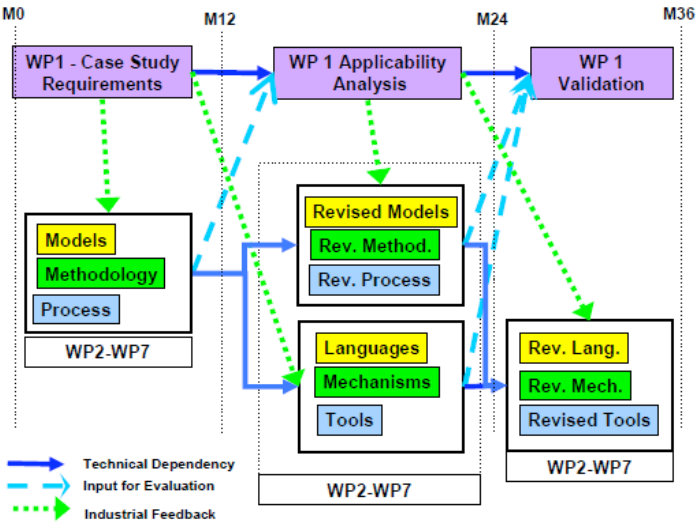
1.3	12 January 2011	Revised Draft	TheinTun (OU)	Revised Section 2
1.4	12 January 2011	Revised Draft	Federica Paci, John Mylopoulos (UNITN)	Revised Exective Summary, Introduction and Section 3
1.5	12 January 2011	Revised Draft	Thein Tun (OU)	Addressed comments by Federica for Section 2.
1.6	13 January 2011	Revised Draft	Gábor Bergmann (BME)	Updated Scenario descriptions to match reviewer comments.
1.7	14 January 2011	Final	Federica Paci(UNITN)	Corrections and formatting issues solved
1.8	17 January 2011	Draft	Michela Angeli (UNITN)	Quality check complete-Minor Remarks
1.9	18 January 2011	Final	Federica Paci (UNITN)	Addressed Quality Check Remarks

Executive summary

This deliverable shows how automatic or semi-automatic model transformation algorithms make it possible to propagate and explore the impact of changes in requirements, and present analytical reasoning techniques to assess the impact of changes on security properties. In particular the deliverable presents a qualitative and a quantitative reasoning technique on evolving requirements models. The qualitative technique is based on argumentation analysis that is adopted by the SeCMER methodology to analyze whether the design has exploitable vulnerabilities that might expose valuable assets to malicious attacks. The quantitative technique, instead, helps the designer to select a system design that is resilient to changing requirements. Moreover, this deliverable introduces the semi-automatic approach to requirements change management based on incremental graph patterns transformations that is part of SeCMER methodology. The reasoning techniques and the approach for requirements change management are illustrated based on the process level and the organizational level change requirements of the ATM case study.

Position of the deliverable in the project timeline

The main artefacts of WP3 are the SeCMER conceptual model, the SeCMER methodology for changing requirements, and a CASE tool prototype that supports the different steps of SeCMER methodology. Considering the SecureChange project timeline depicted above, the SecMER conceptual model and the SeCMER methodology have been conceived during the M0-M24 timeframe, while the CASE tool is going to be developed during the M24-M36 timeframe. The reasoning techniques presented in this deliverable are part of the SeCMER methodology and thus belong to the timeframe M0-M24.



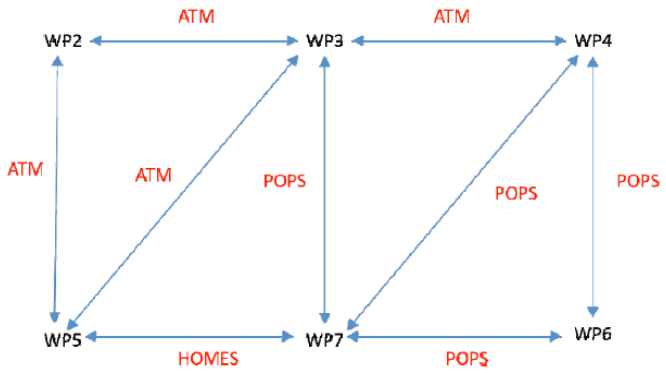
Validation

The WP3 artefacts are SeCMER conceptual model, the SeCMER methodology for changing requirements, and a CASE tool prototype. Each of these artefacts is subject to the validation activities in SecureChange.

The validation activities have not started yet and will be carried out during the third year of the project by organizing a dedicated workshop with ATM experts. For the purpose of the validation, we will use the *process level change* and the *organizational level change* and the security properties *information protection* and *information access*. WP3 uses also the POPS case study, but to a lesser extent to illustrate the integration with testing (WP7). The change requirement that is addressed is *specification evolution*, and the security property is *life-cycle consistency*.

Integration

The strategic position of WP3 in terms of case studies and integration with technical artefacts of the other work packages is shown in the figure below. The ATM case study serves as the example for demonstrating the integration with artefacts of WP2, WP4, and WP5. The POPs case study is used for exemplifying the integration with artefacts of WP7.



WP3-WP2. The integration link between WP3 and WP2 is reported in D3.2 and D2.2. The integration is both at artefacts and at process level. The SeCMER conceptual model of the evolving requirements is a specialisation of the Requirement Model package of the Integrated Meta Model presented in D2.2, while the SeCMER methodology steps are an instantiation of the Overall Process. The integration is demonstrated based on the ATM case study, addressing the organization level change and the security properties of information protection.

WP3-WP4. The integration link between WP3 and WP4 is reported in D3.2 and D4.2. The integration shows how UMLseCh can be used to help with verifying that requirements are actually met by a system and that they are complete with respect to high-level security objectives. The integration is demonstrated with the ATM case study, addressing the organization level change and the security properties of information protection and information provision.

WP3-WP5. The integration link between WP3 and WP5 is reported in D3.2. The integration is both at conceptual level and at process level. At the conceptual level, an integration of concepts is presented and it is explained how requirement model artefacts should be mapped to risk model artefacts and vice versa. The process level integration leverages on the conceptual level integration for the integration of the requirements elicitation and risk assessment methodologies. The integration is demonstrated in the ATM case study, addressing the organization level change and the security properties of information protection and information provision.

WP3-WP7. The integration link between WP3 and WP7 is reported in D3.2. The integration is both at conceptual level and at process level. At the conceptual level, an integration of concepts is presented and it is explained how requirements artefacts should be mapped to test artefacts and vice versa. At the process level, the integration of requirements methodology and testing methodology is described.The integration is demonstrated based on the specification evolution change requirement of the POPS case study.

Index

1 Introduction

Modern software systems become more complex and the environment where these systems operate in become more and more dynamic. The number of stakeholders increase and the stakeholders' needs change constantly as they need to adjust to the constantly changing environment. A consequence of this trend is that the average number of requirements for a software system increases and changes continually.

To deal with evolution, it is important to have analysis techniques to assess the impact on the satisfaction of requirements. Beside reasoning on requirement satisfaction, it also necessary to evaluate the impact of evolution on the security of the system: security properties satisfied before evolution might no longer hold or new security properties need to be satisfied as result of the evolution.

Another important aspect about requirements evolution is the change management process. However, requirements evolution management is a major problem in practice. Requirements change continuously making the traceability of requirements hard and the monitoring of requirements unreliable. Furthermore, requirements management is difficult, time-consuming and error-prone when done manually. Thus, a semi-automated requirements evolution management environment, supported by a tool, will improve requirement management with respect to keeping requirements traceability consistent, realizing reliable requirements monitoring, improving the quality of the documentation, and reducing the manual effort.

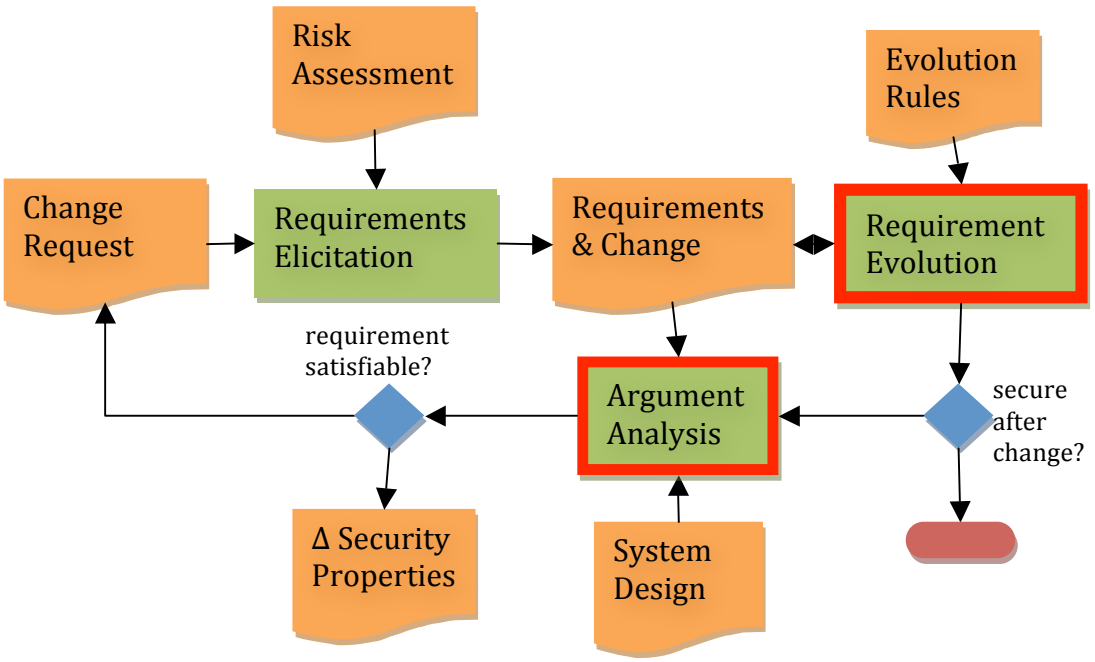


Figure 1. An Overview of SeCMER methodology

In this deliverable we present a qualitative and a quantitative reasoning technique for evolving requirements models, and a semi-automatic approach to requirements change management that is based on incremental graph patterns and change driven transformations.

The qualitative reasoning technique is introduced in Appendix A. It is based on argumentation analysis that is one of the steps of the SecMER methodology outlined in red in Figure 1. Arguments provide a way to structure the system artefacts involving the concepts in the SecMER conceptual model. The requirements engineer uses the SecMER requirement model to sketch informal arguments for the security goals of the system which will be affected by the proposed change. Goal, requirement and security goal are generally regarded as claims, context is the warrant, and propositions are facts. Other concepts such as action, actor, depends, provides relation, are orthogonal to the argument, in the sense that they can be used in the description of any part of an argument.The informal arguments and the formalized requirements are then used to generate arguments formalized in the Event Calculus. Two kinds of reasoning can be performed on the arguments: logical deductive reasoning to check whether claims in the arguments are valid, and logical abductive reasoning to find rebuttals to the claims.

The qualitative analysis is presented in Appendix B. The analysis helps the designer to select a system design that is resilient to changing requirements. The reasoning is based on the representation of the evolution of a requirement model in terms of *controllable* and *observable* evolution rules. A controllable evolution rule represents an evolution that is under the control of the designer: the rule associates with a requirement model a possible design alternative. An observable rule, instead, represents an evolution which is not under the control of the designer, but it can be somehow detected when it happened or whose future likelihood can be estimated with a certain confidence of the stakeholder. Thus, an observable evolution rule associates with a requirement model a possible requirement model to which the model can evolve with a certain probability. The probability of occurrence of an observable rule is determined based on a game-theoretic approach which involves the stakeholder, the designer and the reality. Based on the representation of an evolving requirement model as a set of controllable and observable rules, it is possible to compute two quantitative metrics called *maximal belief* and *residual risk* that intuitively measure the usefulness of a model element (or a set of elements) after evolution. In fact, the maximal belief tells whether a design alternative is useful after evolution, while residual risk quantifies if a design alternative is no longer useful. Based on these two metrics, a designer can thus determine the system design that is resilient to changes in requirements. The system design should consist of the design alternatives that have a high maximal belief and a low residual risk.

Appendix C introduces change-driven transformations. SeCMER methodology leverages on incremental model transformation technology and change-driven transformation based on *evolution rules* to check argument validity**,** to automatically detect violations or fulfilment of security properties, and to issue alerts prompting human intervention, a manual analysis / argumentation process, or potentially trigger automated reactions in certain cases. Change-driven transformation can also be used to synchronize requirements models based on the SecMER conceptual model with models in different requirements formalisms.

We illustrate the reasoning techniques and the approach for requirements change management based on the ATM case study.

The report is organized as follows.

In **Section 2** we illustrate how argumentation analysis introduced in APPENDIX A can be leveraged to analyze the changes in requirements models. In particular, we illustrate how argumentation analysis is used to determine that after the introduction of Arrival Manager (AMAN), and of the IP communication network, the security property "Information Protection" for Flight Data Domain information no longer holds, and the addition of new security properties that need to be satisfied by the requirement model is required.

Section 3 shows how the maximal belief and the residual risk metrics can be applied to assess the design of the Arrival Manager (AMAN) that regards the authentication process for the Sequence Manager and the air traffic controllers who have access to the information generated by the AMAN.

Finally, in **Section 4** we show how incremental transformations and change-driven transformations are used by the SeCMER methodology to manage evolution of requirements. We provide examples of evolution rules that are applied to detect arguments that are no longer valid, security properties violations, and synchronization of different model's views for the process level change requirement of the ATM case study.

Appendix A presents the argumentation analysis. **Appendix B** proposes the quantitative analysis, while the **Appendix C** introduces change driven transformations.

2 APPLICATION OF ARGUMENTATION ANALYSIS TO ATM CASE STUDY

This section is based on paper [1], which proposes a meta-model of evolving security requirements. The meta-model draws on requirements engineering approaches, security analysis, argumentation and software evolution. The meta-model is instantiated using a formalism of temporal logic, called the Event Calculus. The approach to argument analysis is supported by a plug in for the OpenPF tool, which generates templates for formal descriptions.

The paper [1] and the SeCMER methodology discussed in D3.2 are closely related, in particular with respect to the requirement meta-model and the use of argumentation.

2.1 Relationship between ESR meta-model and SeCMER ontology

This section discusses the relationship between the Evolving Security Requirements (ESR) meta-model used in our work in [1] and the SeCMER conceptual model presented in D3.2. The general relationship between the two models is that the SeCMER conceptual model is a simplification and clarification of the ESR meta-model.

The ESR model is particular geared towards the description of the security requirements using the temporal logic formalism, the Event Calculus and argumentation. As far as the requirements engineering concepts are concerned, it makes use of the notions such as anti-requirements and specifications. In that sense, the meta-model is geared towards the Problem Frames concepts.

The SeCMER ontology simplifies the ESR meta-model by removing concepts related to the argumentation and temporal logic, and by generalising Problem Frames and i* concepts, and extending them. Therefore, the notions of goal and actor are more prominent.

For a more detailed discussion of the ESR meta-model and the SeCMER ontology, please refer to Section IV of [1] and Section 2.1 of D.3.2 respectively.

2.2 Using Argumentation: The ATM Case Study

We illustrate argumentation analysis based on the process level change requirement and the information access and information protection properties. The scenario fragment we are going to consider is the transmission of FDD data to the AMAN via the new communication network, denoted as SWIM. We want to focus on how to enforce access control policies on FDD transmission and how to ensure confidentiality of FDD. In terms of security means, we show the SeCMER models before and after changes of introducing the AMAN tool and the communication network, and the argumentation analysis for the security goal of protecting FDD from malicious attack.

Following the SeCMER approach, we will begin by describing the structure of the system before the change (2.2.1). We then show how the structure is affected by the change, whilst indicating the security property that needs to be maintained after the change (2.2.2). Arguments for why the system was secure before the change, and how it may or may not be secure after the change is developed (2.2.3). We then use the mitigations for rebuttal to derive additional security properties that need to be discharged to maintain the security after the change is introduced (2.2.4).

2.2.1 The Structure of ATM system before the change

Figure 2 shows a SeCMER requirement model fragment capturing structure of the ATM system before the introduction of AMAN and SWIM. The model captures the given domains and their connections as they currently are in ATM domains. The diagram shows that the Airport Management is connected to the Meteo Data Center and the Area Control Centre through interfaces ‘a’ and ‘b’ respectively. ‘a’ and ‘b’ are point-to-point communication systems. The main security goal is “Protection of flight data domain information”.

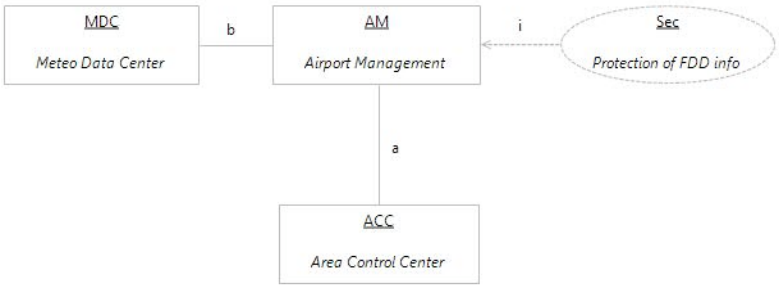


Figure 2. A SeCMER requirement model capturing the relevant domains before the changes

2.2.2 The Structure of ATM after the change

Figure 3 shows the SeCMER requirement model showing how the introduction of the SWIM Network, an IP based data transport network, changes the structure of the ATM components, together with a simple description of the interfaces between the components. In the after change diagram, the two specific legacy systems, namely, Airport Management and Meteo Data Center are connected through the SWIM network. This change, in a sense, replaces the interface ‘b’ with the SWIM network, SWIM boxes and adapters. The diagram also makes explicit the security goal “Protection of flight data domain information” that needs to be maintained after the change has been introduced.

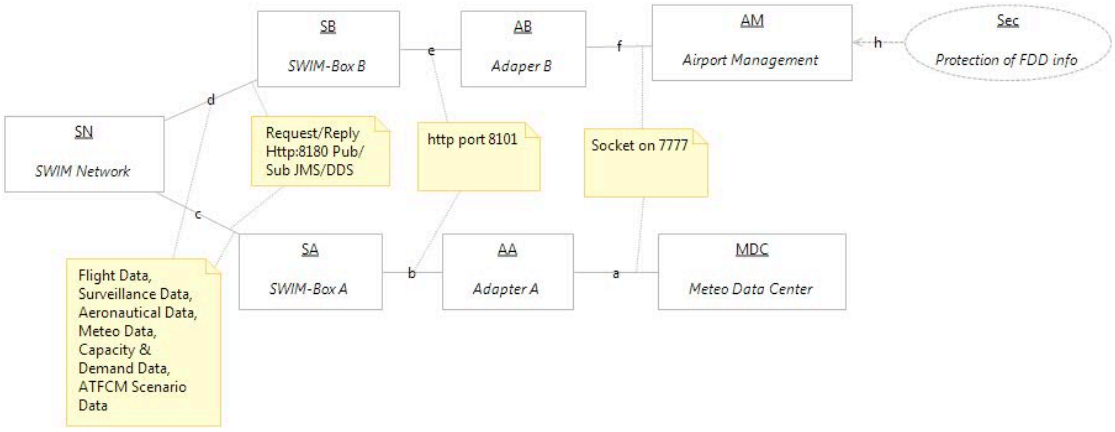


Figure 3. A SeCMER requirement model capturing the relevant domains after the change

2.2.3 Argumentation Analysis of change

The structured models allow us to analyze the impact of the changed requirements. This allows us to discuss and communicate relevant changes to requirements with domain experts. Figure 4 shows a fragment of the SeCMER argument model for the introduction of the AMAN and the SWIM network. Notice that the changes made to the requirement models are reflected by the addition of rebuttals and mitigations in various rounds. In that sense, the changes are recorded in structured way in arguments.

Figure 4. A fragment of a SeCMER argument model

The diagram shows that the AMAN system is claimed to be secure before the change (Round #1), and the claim is warranted by be the facts the system is known to be a close system (F1), and the physical location of the system is protected (F2). This

argument is rebutted in Round #2, in which another argument claims that the system is no longer secure because SWIM will not keep AMAN closed. The rebuttal argument is mitigated in Round #3 by three arguments, which suggest that the AMAN may still be secure given that the physical infrastructure is secure, personnel are trustworthy and access to data is controlled.

2.2.4 Additional security property as a result of change

Having done the argument analysis, we discovered that the original requirement for system security, which is the protection of FDD (Flight Data Domain) info, cannot be maintained after the change has been introduced. The introduction of the AMAN and the SWIM Network requires additional security properties. Mitigations in the arguments led to discovery of these additional security properties, relating to the physical infrastructure, personnel and access control that need to be discharged in order to maintain the overall system security.

Figure 5 shows the after SeCMER requirement model displayed in Figure 4 where one of the additional security property relating to access control has been added. They include: *'Queue Management Information shall not be accessible by meteo data centres'*, or *'Queue Management Information shall not be accessible by anyone other than those working with AMAN'*. The structured SeCMER's argumentation supports the verification of such additional properties.

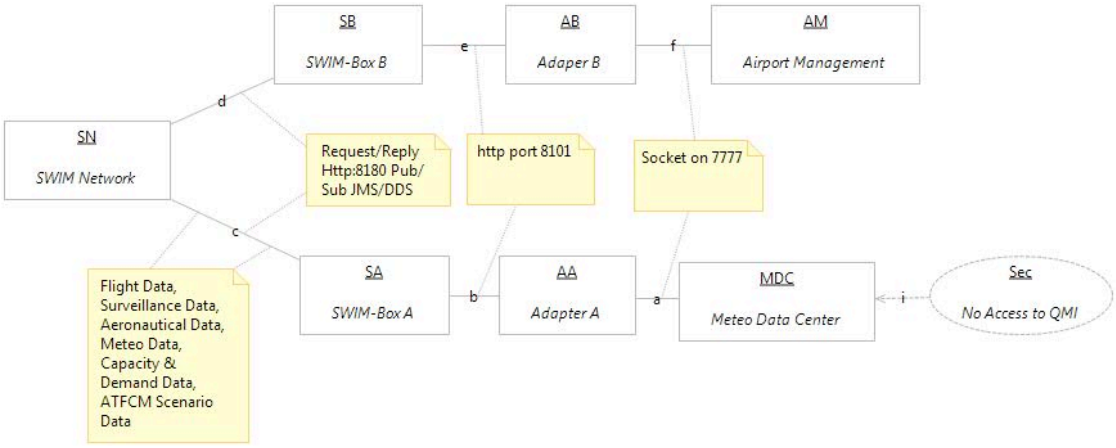


Figure 5. A SeCMER requirement model for a relevant Security Property with respect to Changes

2.2.5 Formalization of the argument using the Event Calculus

We now illustrate how the initial argument for the system security for the running ATM example is broken, by generating counter-examples in the Event Calculus using the

abductive reasoning of OpenPF. Our discussion will focus on the protection of FDD data of the Airport Management system.

Since we assumed that the existing system before the change is secure, in particular, the FDD data is initially protected. In the event calculus, we will write:

!HoldsAt(Accessed_FDD_data_SN(),0).

For the current system, this property can easily be proven. What is of interest is to check whether the property remain true after the change has been introduced. To do this, we input the diagram in Figure 4 into OpenPF. We the follow the steps discussed in the sections IV and V of Appendix A. Figure 6 shows the textual input to create the diagram in Figure 5.

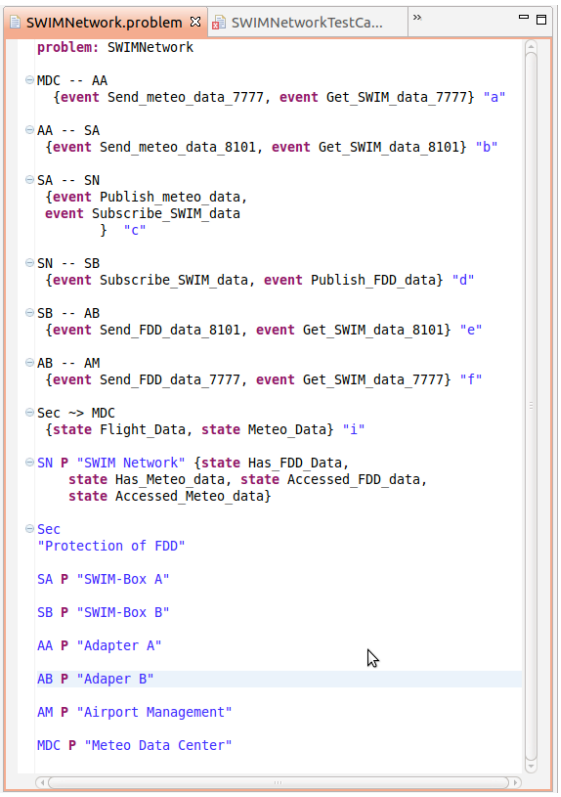


Figure 6. Textual input to create the diagram in Figure 5

In the next step, we generate the diagram shown in Figure 4. We then invoke an OpenPF plug-in that generates the Event Calculus template for the above diagram. A partial template is shown in Figure 7.

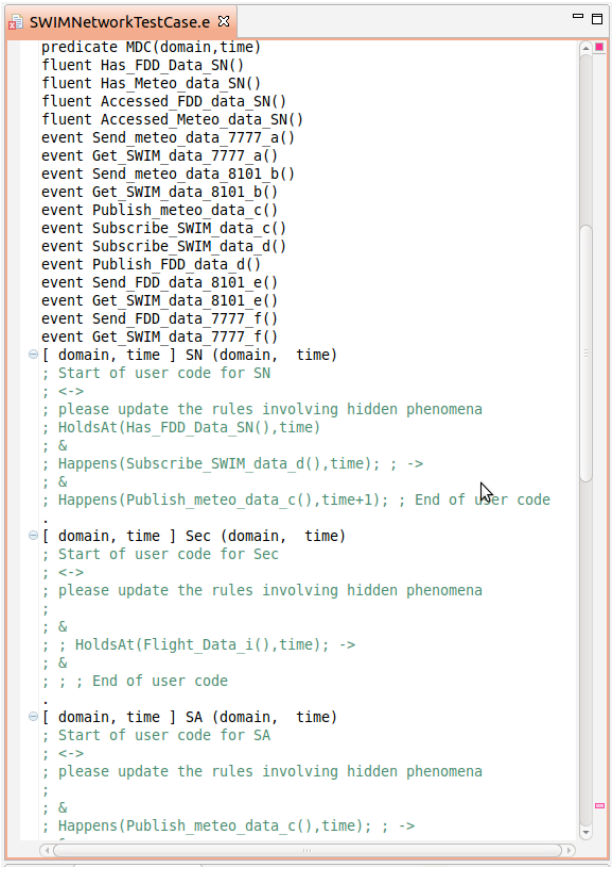


Figure 7. The Event Calculus template generated by the OpenPF tool from Figure 6

In the next step, we describe the behaviour of the domains in Figure 4. For instance, to say that the Adapter B instantly forward FDD data from the Airport Management (via interface f) to the SWIM-Box B (via interface e), we write:

[time] Happens(Send_FDD_data_8101_e(),time+1) <->
Happens(Send_FDD_data_7777_f(),time).

Similarly, to say that the SWIM-Box B instantly publishes the information to the SWIM Network (via interface d) when it receives FDD data from the Adapter B (via interface e), we write:

[time] Happens(Publish_FDD_data_d(),time) <-> Happens(Send_FDD_data_8101_e(),time).

When the FDD data is published with the SWIM Network, the SWIM Network has the FDD data.

[time] Initiates(Publish_FDD_data_d(), Has_FDD_Data_SN(),time).

If SWIM-Box A has subscribed to the SWIM Network, and if the SWIM Network has the SWIM data when SWIM-Box A attempts to get it, then the FDD data has been accessed. This is described by the following rule.

[time,time1] Happens(Subscribe_SWIM_data_c(),time1) & (time1 < time) &

HoldsAt(Has_FDD_Data_SN(),time) ->

Initiates(Get_SWIM_data_c(), Accessed_FDD_data_SN(),time).

Requests by Airport Management and Meteo Data Center for FDD data and Meteo data can be described in the same way.

In the next step, we invoke the abductive reasoned the OpenPF tool to see if the security property !HoldsAt(Accessed_FDD_data_SN(),0) has been broken. The reasoner returns the two models shown in Figure 7. The first model says that the security property Accessed_FDD_data_SN() will become true, i.e. the security is broken, if the Airport Management publishes FDD data to the SWIM Network to which the Meteo Data Center has subscribed for FDD data. The FDD data available to the Meteo Data Center may be outdate because the Airport Management has published more FDD data since the Meteo Data Center has requested it. The second model is similar to the first: the difference being that the FDD data is most up-to-date.

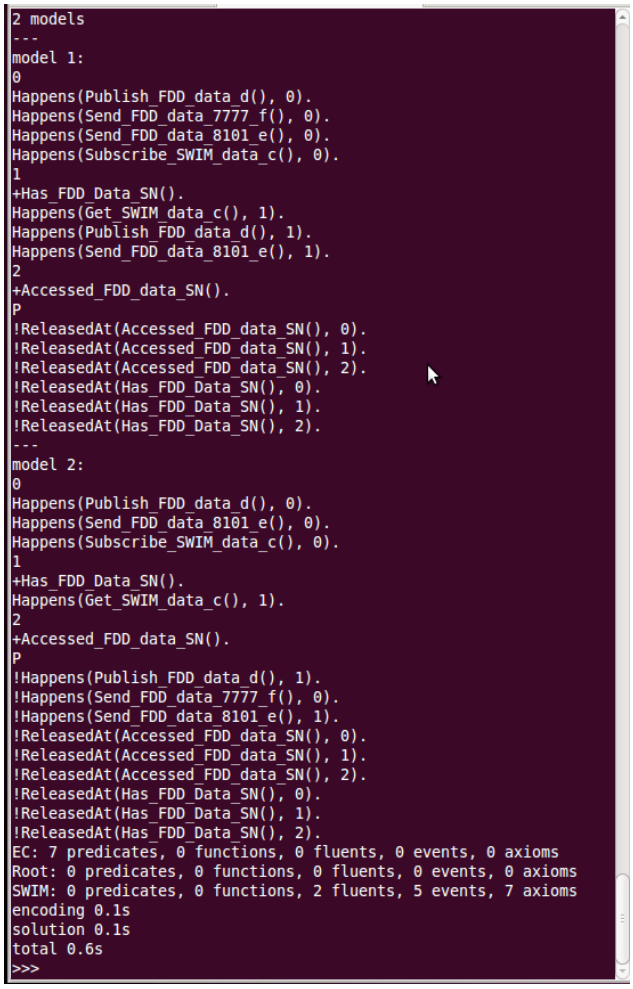


Figure 8. Results of the abductive reasoning on the change

3 HOW TO CHOOSE A SYSTEM DESIGN RESILIENT TO EVOLVING REQUIREMENTS

In this section we show how we can compute maximal belief and residual risk metrics by using the organizational level change requirement of the ATM case study as illustrative example. The maximal belief and residual risk, intuitively, indicate if a set of design alternatives is going to be useful after requirements change. The Organizational Level Change introduces changes both at process and at organizational level. At organizational level, the AMAN supports the Sector Team by providing sequencing and metering capabilities for a runway, airport or constraint point, the creation of an arrival sequence using 'ad hoc' criteria, the management and modification of the proposed sequence, the support of runway allocation at airports with multiple runway configurations, and the generation of advisories for example on the time to lose or gain, or on the aircraft speed. The Sector Team consists of two Air Traffic Controllers (ATCOs), namely the Tactical Controller (TCC) and the Planner Controller (PLC). The Sector Team is responsible for managing the air traffic of an allocated sector of the airspace.

The introduction of the AMAN requires the addition of a new type of ATCO, called Sequence Manager (SQM), who will monitor and modify the sequences generated by the AMAN and will provide information and updates to the Sector Team.

Due to the introduction of the AMAN, there is the need for an Identity and Key Management Infrastructure (IKMI) to support the authentication process for the different ATCOs who have access to the information generated by the AMAN.

Let assume that the requirement model for the Organizational Level Change (denoted as RM_1) states that the following requirement needs to be satisfied:

- **R_1** : Manage keys and identities of system entities (human, software, devices etc).

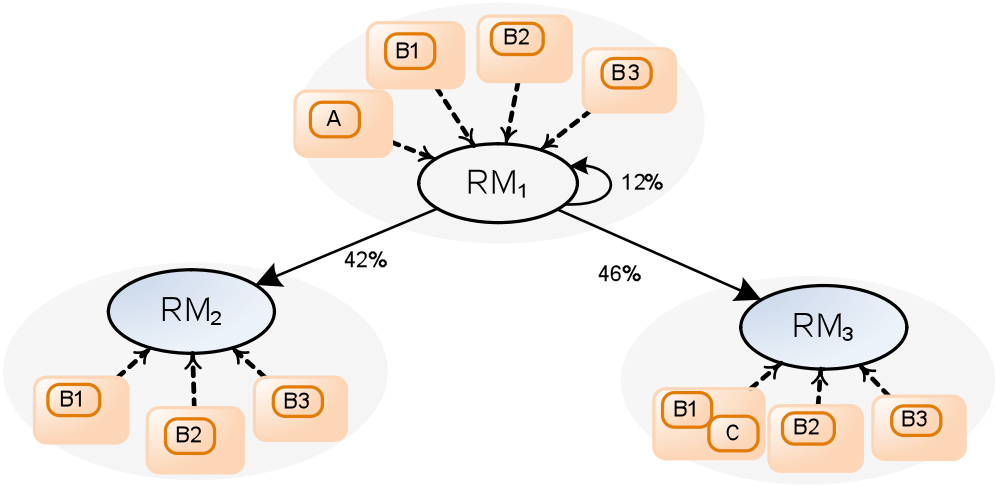


Figure 9. Examples of Evolution of requirement model RM_1

The evolution of requirement model RM_1 is described by a set of controllable and observable evolution rules where a controllable rule associates with RM_1 a possible design alternative. An observable rule associates with RM_1 another requirement model to which RM_1 can evolve with a certain probability. r_c is the set of controllable rules for

RM_1 that state that RM_1 can be fulfilled either by design alternative A, design alternative B_1, design alternative B_2, or design alternative B_3.

$$r_c = \{ RM_1 \rightarrow A, RM_1 \rightarrow B_1, RM_1 \rightarrow B_2, RM_1 \rightarrow B_3\}$$

$$r_o = \{ RM_1 \xrightarrow{12\%} RM_1, RM_1 \xrightarrow{42\%} RM_2, RM_1 \xrightarrow{46\%} RM_3\}$$

r_o is the set of observable rules that state RM_1 can evolve to a requirement model RM_2 with a probability of 42%, or to a requirement model RM_3 with probability 46% or it can remain unchanged with probability 12%. RM_2 is a requirement model that requires the satisfaction of the following requirements:

- **R_1** : Manage keys and identities of system entities (human, software, devices etc).
- **R_2** : Support a robust IKMI that can be scaled up to large number of application and users;

while RM_3 is a requirement model which requires the satisfaction of the following requirements:

- **R_1** : Manage keys and identities of system entities (human, software, devices etc).
- **R_2** : Support a robust IKMI that can be scaled up to large number of application and users;

- **R_3** : Single Sign-On (SSO) support.

Figure 9 illustrates the evolution of RM_1 as modelled by the set of controllable and observable rules. Each requirement model is represented as a bubble in which there is a controllable rule with several design alternatives. Each design alternative is an element set represented as a rounded rectangle that contains elements (such as A, B_1, and B_2) which fulfil the requirements within that requirement model. Table 1 summarizes the possible design alternatives that implement requirements R_1, R_2, and R_3.

Identifier	Design alternative	Requirement
A	Simple IKMI	R_1
B1	OpenLDAP based IKMI	R_1, R_2
B2	Active Directory based IKMI	R_1, R_2, R_3
B3	Oracle Identity Directory based IKMI	R_1, R_2, R_3
C	Ad-hoc SSO	R_3

Table 1. Model elements fulfilling requirements

Table 2 reports the value of maximal belief and residual risk: the first column displays design alternatives sets, and the two next columns show the values of maximal belief and residual risk. Notice that the maximal belief and residual risk in the first row, where the element set is {C} is n/a which means that there is no evolution where the set {C} fulfils all the requirements R1, R2, and R3.

Model Element Set	Maximal Belief	Residual Risk
{C}	n/a	n/a
{A}	12%	88%
{B_1}	42%	46%
{B_2}	46%	0%
{B_3}	46%	0%
{B_1, C}	46%	0%

Table 2. Examples of Max Belief and Residual Risk

Looking at the table we can conclude that {B_2}, {B_3} and {B_1, C} seem to be the best design alternatives, since they have a high maximal belief [1](46%) and low residual risk[2] (0%) which means these element sets are surely still useful after evolution. Thus, a system design that is resilient to the evolution of RM_1 should include design alternative B_2, design alternative B_3, or design alternatives B_1 and C.

[1] Maximal belief denotes that a design alternative is useful after evolution.

[2] Residual risk denotes that a design alternative is no longer useful after evolution.

4 APPLICATION OF CHANGE-DRIVEN TRANSFORMATIONS TO SECMER AND ATM CASE STUDY

4.1 Overview

Thanks to Evolution Rules, our proposed SecMER tool environment can take advantage of incremental model transformation technology and change-driven features, including:

- **argument validity maintenance**, where a previously conducted formal or informal argumentation can be invalidated by changes affecting model elements that contributed to the argument as evidence;
- **automated detection of violations and fulfilment**, where some security-related properties and undesired situations are formally defined and incrementally evaluated, to issue alerts prompting human intervention, a manual analysis / argumentation process, or potentially even automated reactions in certain cases;
- **incremental model synchronization**, where different domain-specific views of an abstract integrated model are incrementally kept synchronized using change-driven model transformation, such as different security requirement formalisms and the unified SecMER requirement model.

The key capability is the automated change-driven reaction mechanism of Evolution Rules. For design-time modification of the models, these reactions can be executed on-the-fly using incremental model transformation techniques. Alternatively, off-line editing of models and later change reconciliation is also possible. Likewise, various change scenarios can be supported transparently with the same declarative definition of Evolution Rules. The foundations of this versatility of the proposed change-driven approach are explained in APPENDIX C.

As envisioned in D.3.2., evolution rules can be **defined and refined** at two stages of the development process at least. Some rules are expected to be defined a priori, when the engineering process is set up, relying on general domain knowledge or a domain-agnostic library of evolution rules. New rules can be defined and existing rules can be refined during the engineering process itself; adapted to the stakeholder needs, internal policy, and system-specific experience gathered; emerging as the result of manual decisions, e.g. in the argumentation part of the methodology. The integration of Evolution Rules into the methodology is presented in Section 7.2 (“Application of evolution rules in SeCMER”) of D.3.2.

4.2 Example model

We now briefly introduce a heavily simplified ATM requirements model that will be used below for explanatory examples. The model is visualised here by the Si* graphical syntax (**Figure 11**), as well as the textual format of the SecMER metamodel (**Figure 10**).

The entities involved in the simple scenario used for this example are the AMAN, the Meteo Data Center (MDC), the SWIM-Box and the SWIM-Network. The SWIM-Box is the core of the SWIM information management system which provides access via defined services data that belong to different domain such as flight, surveillance, meteo, etc.

The introduction of the SWIM requires many security properties to be satisfied which prevent corruption, accidental or intentional loss of data and guarantee the integrity and confidentiality of the aircraft sensible data against malicious attacks or intrusions. Here we will focus on information access (access control) and information protection (e.g. integrity) properties on the requirements level. In particular, we will show how to ensure integrity of FDD data by using digital signature or a trusted communication path.

As exploring design alternatives is inherent to the methodology, we will also show how a cheaper alternative is investigated after the default option of the planned change. One of the elements in the model will be modified to represent the second alternative, demonstrating how the analysis is adapted to the new situation. Note that even though technically the model is modified for a second time, the whole engineering session revolves around a single evolution of the system.

In the post-state model we are investigating here, the two main actors are AMAN and MDC, no longer interfacing over a direct connection. MDC provides the asset Meteo Data (MD) which is sent to AMAN, but the communication is indirect: MD is first given to the SWIM-Box of MDC, which moves it to the SWIM network, from there it is propagated to the SWIM-Box of AMAN, and AMAN retrieves MD from there. AMAN has a security goal MDIntegrity requiring the integrity of MD, and MDC is trusted to comply with this security requirement, but the rest of the actors are not yet trusted. A second security goal, MDAccessControl (corresponding delegation and trust relationships omitted from the model) is investigated by an argument carried out by experts. AMAN also performs an Action, SecurityScreening, to regularly conduct a background check on its employees to ensure that they do not pose a risk of internal compromise; this Action is used in the argument as a ground Fact.

```
model  ATM2:

interfaces (AMAN,MDC) "dc"

provides (MDC, MD)
consumes (AMAN, MD)

protects (IntegrityMD, MD)
wants (AMAN, IntegrityMD)

delegates (MDC, AMAN, MD)
trusts (AMAN, MDC, IntegrityMD)

carries out (AMAN, SecurityScreening)

MDC actor   "Meteo Data Center"
AMAN  actor   "Airport Manager"
MD resource "Meteo Data"
IntegrityMD sec "Integrity of MD info"
SecurityScreening action
    "Regular screening of AMAN personnel"
```

Figure 10. ATM evolution pre-state in SecMER concrete syntax

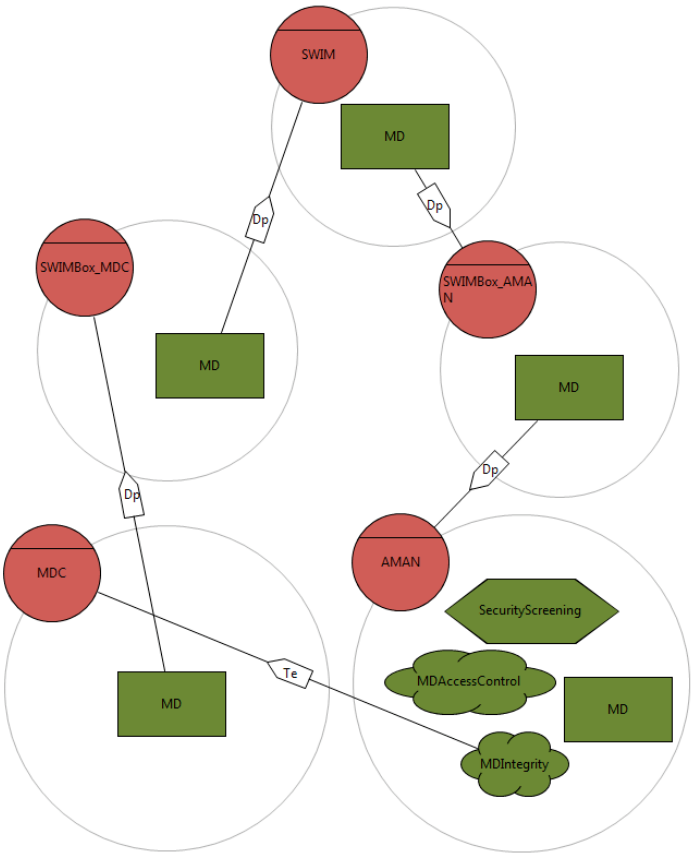


Figure 11. ATM evolution post-state on Si* diagram

4.3 Argument validity maintenance

Formal and informal argumentation is carried out using the SecMER requirements model, to determine which security goals are met. Argumentation is a laborious and costly process requiring significant human expertise. In security-critical applications, it is important to avoid carrying out the argumentation workflow for every single security goal after each insignificant change to even remote parts of the model.

Our assumption is that the argumentation model relies on facts, some of which are grounded in the model. Traceability is required between facts used in argumentation and the requirements model elements corresponding to these facts. This traceability mapping enables automated mechanisms to detect changes that are inflicted upon these model elements corresponding to ground facts. Even if an argumentation was previously carried out, such changes can influence the validity of the argument, and it would have to be revisited.

We propose establishing a set of Evolution Rules that would flag such arguments as invalid and request argumentation analysis. The key difference between each of these rules is the guard Graph Change Pattern (GCP, defined in D.3.2.) defining the Event and Condition part of the rule. The recommended strategy is to identify types of changes that warrant a re-evaluation of the argument, and define an evolution rule for each of them.

To aid in building Evolution Rules and GCPs, some auxiliary *Graph Patterns* are defined first. Graph Pattern `factOfValidArgument(A,F)`captures an Argument A that has not been invalidated, and a Fact F that is listed as a ground fact for A. Graph Pattern `factMentionsElement(F,M)`captures the traceability relationship between a Fact F and a model element M of the integrated model, that is "mentioned" or used in the fact statement as an evidence.

The Evolution Rule `invalidateUponElementDeletion()` is activated when a model element mentioned in a Fact is deleted. The rule is guarded by a GCP that contains a disappearance query of `factMentionsElement(F,M)` linked to a match of `factOfValidArgument(A,F)`. The following code sample shows an initial version of this rule in simplified syntax:

```
change pattern factEvidenceDeleted(A,F,M) {
      find factOfValidArgument(A,F); // static condition
      disappear factMentionsElement(F,M); // event: element disappeared!
}
evolution rule invalidateUponElementDeletion(A,F,M){
    guard factEvidenceDeleted(A,F,M);
    action {
        call flag_as_invalid(A);
    }
}
```

As an example for the application of this evolution rule, suppose the argumentation analysis described in Section 2.2 has already been conducted. This means that Mitigation A ("SWIM personnel are trustworthy") and its underlying Fact F ("People working with SWIM data will be vetted"), both shown in Figure 4**Error! Reference**

source not found., constitute a match(A,F) of the Graph Pattern factOfValidArgument(). At the same time, this Fact is linked by inter-model traceability to its evidence M in the requirements model, namely the Action "Security Screening" carried out by AMAN. The Fact and the Action together form a match(F,M) for the Graph Pattern factMentionsElement(). If an evolution deletes this Action (e.g. the SWIM operator is planning to cut costs), the match of factMentionsElement(F,M) will disappear, making (A,F,M) a match of the change pattern and activating the Evolution Rule. The rule will flag the Mitigation argument for re-evaluation; argumentation experts will be alerted to revisit the argumentation and decide whether the top-level Claim ("The AMAN system is secure") can still be considered true.

To avoid triggering the rule where a mistakenly created "fact mentions model element" relationship is retracted, the GCP should be improved further. For example, an additional disappearance query can be used to ensure that the model element M is in fact deleted.

The Evolution Rule invalidateUponAttributeChange() is proposed to activate when an attribute of a model element mentioned in a Fact is updated. The rule is guarded by a GCP that contains an attribute update query of an arbitrary attribute of model element M, with M involved in a match of factMentionsElement(F,M) that is linked to a match of factOfValidArgument(A,F). Although attribute update queries are omitted from D.3.2 for sake of brevity, they are discussed in detail in **[3]**. An example use case for this rule would be a moderate budget cut where the SWIM operator down scales the quarterly security screening to a once-a-year frequency; the rule flags the corresponding mitigation for re-evaluation, and argumentation experts will have to decide whether the looser security policy is enough to support the root claim.

Additional similar Evolution Rules can be created depending on system-specific policies; for instance the argument should be invalidated if a model element mentioned in a fact is of a certain type, and edges of certain types are connected to (or disconnected from) it.

Finally, invalidity should be propagated upwards in the argumentation tree. The Evolution Rule propagateInvalidatation() is proposed to activate when a sub-argument (through nesting or linking) is invalidated. Further optimizations are possible, e.g. an invalidated rebuttal does not invalidate an argument, but an invalidated mitigation of the rebuttal does.

4.4 Automated detection of violations and fulfilment

We claim that in certain scenarios, the formal approach and (first-order) expressivity of Evolution Rules is sufficient to detect various kinds of failures or integrity validation in formal models such as the requirements model. Conversely, these techniques may be enough in some cases to determine that a certain security property is met.

Recalling that Evolution Rules are defined using *Graph Patterns* to capture static or dynamic parts of the model, the general strategy is to compose graph patterns that capture desired or undesired situations, and to establish Evolution Rules to react on them. The proposed underlying technology supports the incremental evaluation of these conditions.

Some examples have already been introduced in D.3.2. Here we show additional rule templates (and constituent graph patterns) to demonstrate the power and applicability of the approach.

One useful Graph Pattern called `fulfilledGoal()`captures a desired situation where a security goal is known to be fulfilled by an explicitly specified action. The pattern consists of Actor A that wants a security goal SG that is fulfilled by Action X performed by Actor B; while at the same time A delegates the SG to B and also trusts B over the SG. The applicability of this pattern can be improved by increasing its complexity, e.g. taking into account the fulfilment of a decomposed goal through constituent goals, and of course considering the case when an Actor fulfils a Goal herself (A=B).

```
pattern fulfilledGoal(A,SG,X) {
	find want(A,SG);
	find securityGoal(SG);
	find fulfil(X,SG);
	find carryOut(B,X);
	find delegate(A,B,SG);
	find trust(A,B,SG);
}
```

A second proposed graph pattern called `assetLeak()`captures an undesired situation where an asset that should be protected is handed over to an actor without trust. The pattern consists of an Actor A that wants a Security goal SG that protects asset S; while at the same time S is delegated (by A or another actor) to an actor B that is not trusted by A over S.

```
pattern assetLeak(A,B,SG,S) {
	find want(A,SG);
	find securityGoal(SG);
	find protect(SG,S);
	find delegate(_,B,S);
	negfind trust(A,B,S);
}
```

Depending on process-specific policies, a meaningful Evolution Rule could be `alertUnfulfilledGoal()` as described in the following. Security engineers are alerted to a problem with a security goal SG, whenever a new match of the graph pattern `unfulfilledGoal()` emerges. The latter pattern holds for Security goal SG if SG is not supported by an argumentation that is marked as valid, and furthermore either `assetLeak()` holds for SG and some Asset S, or there is no action X such that `fulfilledGoal()`holds for SG and X. The Evolution Rule is triggered when this undesired pattern appears, regardless what kind of evolution caused it: a change introducing a new security goal (to which no explicit fulfilment relationship is connected yet), a change where the Action no longer fulfils the Security goal, a change where an asset becomes leaked (due to a new delegation or a revoked trust relationship), or a change where an existing argument is marked as invalid (possibly automatically by Evolution Rules described in Section 4.3).

```
pattern unfulfilledGoal(A,SG) {
	find want(A,SG);
```

```
        find securityGoal(SG);
        neg find validArgumentSupports(_Arg,SG);
        find assetLeak(A,B,SG,S);
}or {
        find want(A,SG);
        find securityGoal(SG);
        neg find validArgumentSupports(_Arg,SG);
        neg find fulfilledGoal(A,SG,_X);
}
evolution rule alertUnfulfilledGoal(A,F,M) {
    guard appear unfulfilledGoal(A,SG);
    action {
        call raise_alert(A,SG);
    }
}
```

In the ATM scenario, the running example for evolution triggers this rule in the following way. Actor MDC (“Meteo Data Center”) provides Asset MD (“Meteo Data”). Initially, the asset is delegated directly to AMAN. The change replaces the direct connection with the SWIM Network, and a separate Swim Box each of for MDC and AMAN. MD is communicated (delegated) by MDC to its SWIM box, that delegates it to SWIM Network, that in turn delegates MD to the SWIM Box of AMAN, and finally it is delegated to AMAN. Before any trust edges are introduced, delegating MD to the various actors in the communication chain constitutes a leak. As now the security goal MDIntegrity protecting MD is made uncertain by the leak of MD, and there is no argument to support MDIntegrity by explicitly explaining why the situation is fine nevertheless, the rule `alertUnfulfilledGoal()`springs into action and alerts the engineers of the problem. The situation can be remedied if trust is added to the model. While the simple graph patterns above are not prepared to handle transitive trust relationships, a more elaborate set of patterns and rules would accept the case where AMAN trusts SWIM over MD, and SWIM trusts the SWIM Boxes over MD, which might be more realistic than AMAN trusting every involved SWIM Box.

4.5 Incremental model synchronization

Incremental graph pattern matching, which powers Evolution Rules can also be exploited to maintain a model mapping relationship between different models involved in the process. These mappings can include the synchronization between various requirement modelling formalisms and the unified model of the SecMER methodology, and the communication between requirements models and other domain models. Due to the incremental pattern matching technology, source incrementality is achieved and the mapping can be performed on-the-fly.

When establishing incremental synchronization between two domains, several transformation rules have to be created. For model entities (graph nodes), mapping rules have to be defined for the following cases:

- when a new entity is created in one domain, it should be mapped to the other domain and the corresponding element created there,
- when an entity is deleted from one domain, its image (if any) should also be deleted from the other domain,
- when the name or another attribute of an element is changed in one domain, the image of the element in the other domain should be changed accordingly.

Similar rules can be created for relationships (graph edges).

For cases where there is no one to one mapping, rules have to be adapted accordingly; if a model in domain A has multiple images in domain B, then one of them is created as the image of A, but it should be possible later to choose a different one and still keep the model in domain A intact, and yet consider them synchronized.

Traceability is a further important concern for the rule-based transformation algorithm. Elements created by the synchronizing transformation should be provided with traceability information, so that they can be traced to their pre-image. This information is necessary for the incremental rules outlined above.

Applying the general ideas to the problem of security requirements modelling, the requirements model (or parts of it) can be represented in multiple formalisms, with incremental synchronization provided between them. The possible formalisms include the SecMER conceptual model outlined in D.3.2 as our core conceptual model as well as Security DSML, Problem Frames, or Tropos. Whenever one of the formalisms is manually changed, the relevant fragments that can also be represented in other formalism are incrementally propagated there. If some aspects of the change cannot be represented in the domain where the initial manual modification is made, they can later be applied in other formalisms.

For example, in the ATM scenario, the introduction of the SWIM implies that new nodes and edges are created, and obsolete ones are deleted. The initial manual change can be indicated in a Problem Frames model by representing SWIM by a new causal domain (along with corresponding interfaces).These new elements are mapped to the SecMER conceptual model as actors and delegation of information resources. They are in turn propagated to Tropos as Tropos actors and delegations.

References

[1] Thein Than Tun, Yijun Yu, Charles Haley, Bashar Nuseibeh. “Model-based Argument Analysis for Evolving Security Requirements”. In Fourth IEEE International Conference on Secure Software Integration and Reliability Improvement, 88-97, Washington, DC, USA, 9-11 June 2010.

[2] Minh Sang Tran, Fabio Massacci. “Dealing with Known Unknowns: Towards a Game-Theoretic foundation for Software Requirement Evolution”. Submitted to CAISE 2011.

[3] Gábor Bergmann, István Ráth, Gergely Varró, Dániel Varró. “*Change-Driven Model Transformations. Change (in) the Rule to Rule the Change*”. In Software and System Modeling, *under review*.

APPENDIX A. Model-based Argument Analysis for Evolving Security Requirements

2010 Fourth IEEE International Conference on Secure Software Integration and Reliability Improvement

Model-based Argument Analysis for Evolving Security Requirements

Thein Than Tun[1], Yijun Yu[1], Charles Haley[1], Bashar Nuseibeh[12]
[1]*Department of Computing, The Open University, Milton Keynes, UK*
[2]*Lero, Limerick, Ireland*
{*t.t.tun, y.yu, c.b.haley, b.a.nuseibeh*}*@open.ac.uk*

***Abstract*—Software systems are made to evolve in response to changes in their contexts and requirements. As the systems evolve, security concerns need to be analysed in order to evaluate the impact of changes on the systems. We propose to investigate such changes by applying a meta-model of evolving security requirements, which draws on requirements engineering approaches, security analysis, argumentation and software evolution. In this paper, we show how the meta-model can be instantiated using a formalism of temporal logic, called the Event Calculus. The main contribution is a model based approach to argument analysis, supported by a tool which generates templates for formal descriptions of the evolving system. We apply our approach to several examples from an Air Traffic Management case study.**

***Keywords*-Security argumentation; Requirements Engineering; Evolution; Event Calculus; OpenPF**

I. Introduction

Long-lived software systems often evolve over an extended period of time. Evolution of these systems is inevitable as they need to continue to satisfy changing business needs, new regulations/standards and the introduction of novel technologies.

Such evolution may add, remove, or modify the requirements and parts of the system contexts, and migrate the system from one operating platform to another. These changes may result in requirements that were satisfied in a previous release of a system not being satisfied in the newer release of the system. When evolutionary changes violate security requirements, a system may be left vulnerable to attacks.

As a software system evolves, security concerns need to be analysed in order to evaluate the impact of changes on the requirements. Traditionally, changes that could affect the system security have been handled in an ad-hoc way. For instance, changes are often described in an informal language, whilst the information about the existing system design is partial. Analysing the security impact of changes is therefore a complex challenge.

By adopting a model-based engineering methodology, we propose to investigate such changes using a meta-model of Evolving Security Requirements (ESR). The ESR meta-model has the following characteristics.

1) Security problems are examined at the requirements level
2) Security-specific concepts such as attacker, assets and vulnerabilities are made explicit
3) Argumentation is used to describe relationship between formal and informal descriptions of system artefacts in order to show why the system is thought to be secure
4) Evolutionary changes are considered orthogonal to all artefacts

The ESR meta-model draws on the concepts in requirements engineering, security analysis, argumentation, and software evolution.

In this paper, we show how the ESR meta-model can be used to generate templates for formal descriptions of a system, in a way similar to model-driven code generation. Security problems of software systems are described using the problem diagrams [1]. Our tool OpenPF performs the model-to-text transformation to generate Event Calculus descriptions compliant with the ESR meta-model. As a result, changes in the meta-model can be reflected by the generated Event Calculus descriptions before modifying and feeding into a reasoning engine for security analysis. Since the meta-model is rich enough to express core concepts in security requirements, many of the existing RE languages can be mapped to the meta-model so that the argumentation can be performed to analyse the changes in the evolution.

The major advantage of our approach is that it frees the requirements engineers from having to write mundane parts of the formal descriptions as the system evolve: they can focus on the more abstract and critical part of the descriptions instead.

We have applied the transformations to examples from an Air Traffic Management system. The study shows that a large part of the formal descriptions can be generated using the OpenPF tool, whilst improving the interface with the Event Calculus reasoning engine.

The rest of the paper is organized as follows. Section II relates our work to similar approaches in the literature. Section III presents an illustrative example from the Air Traffic Management (ATM) system. Section IV shows the key parts of the ESR meta-model, which captures the key concepts involved in our analysis. Section V gives the syntax and semantics of the Event Calculus formalism, which is the target language for the transformations. The proposed tool, OpenPF is described in Section VI. Section VII uses

978-0-7695-4086-3/10 $26.00 © 2010 IEEE
DOI 10.1109/SSIRI.2010.36

the running example to go through the transformation and reasoning processes and explains the results. Section VIII gives concluding comments.

II. Related Work

By way of putting our discussions into context, this section provides a survey of related work, which covers existing research into meta-models of requirements, security requirements, change management and formalisation of argumentation.

Meta-models of (security) requirements: Gunter et al. [2] formalise the relationships between key artefacts in requirements engineering, namely, requirement, problem world context, specification, program, and computer. They further define the responsibilities of requirements engineers and software developers. Earlier, Parnas and Madey proposed the four-variable model in [3]. Jureta et al. [4] extend the Jackson-Zave framework in order to include concepts such as beliefs, desires, intentions, and attitudes. These meta-models are not explicit about security requirements: they do not distinguish between users and attackers, for instance.

There are several proposals for meta-models of security requirements engineering: Hartong et al describe a meta-model of misuse cases [5]. Susi et al [6] give a meta model of Tropos. van Lamsweerde [7] suggests that KAOS provide necessary concepts for analysing intentional security requirements. Elahi et al. [8] propose a ontology of security requirements which focuses on the notion of vulnerability. Beydoun et al [9] incorporate security issues into a meta-model of multi-agent systems. Basin et al describe a meta-model-based approach to analysing access control problems. A taxonomy of information security is provided by Savolainen et al in [10]. Lee et al [11] propose a domain ontology based on regulatory documents.

Although these proposals are useful in modelling different aspects of security requirements, they do not capture evolutionary nature of security requirements. Furthermore, the fact that security analysis has to draw on the formal and informal descriptions of the system is often overlooked. Our meta-model is geared towards evolutionary security requirements, and argumentation for bringing together formal and informal descriptions.

Security requirements engineering: Several requirements engineering approaches for security engineering have been proposed [12], [13], and many of the approaches have been surveyed in [14], and here we briefly recall some of them to put this work into context.

In [13], precondition calculus has been used to regressively compute obstacles. Label propagation has been used in goal-oriented requirements engineering in order to analyse satisfaction of security requirements [15]. However, the issue of maintaining the security while introducing change to an existing system has not been extensively studied.

Change Management: The importance of managing change in software development has been recognized for a long time [16]. Several approaches for investigating software evolution have been developed, focusing on various issues including: empirical observation of change [17], feature location [18], version control of development artefacts [19], evolving the software systems during the runtime [20], change recommendations based on historical data and heuristics [21], [22], and economic analysis of change [23].

Although many of these issues are present in the evolution of secure software systems, in this work we focus on the unresolved issue of model-based generating of formal descriptions in order to facilitate automated analysis of change.

Arugmentations: Human intuitions about argumentation have been formalized recently using various logics [24]–[29], in which the use of formal and informal argumentations in the context of requirements engineering have been discussed. For example, argumentation has been useful to document the correctness and completeness of goal decomposition using GSN in [29]. Earlier, an argumentation framework to security requirements has been developed and applied in [30]. This work can be considered as an extension to [30] that addresses the argumentation problem together with tool supported change analysis.

III. ATM Example: Sending Weather Data

One of the key services of ATC systems is to maintain a degree of separation distance between aircrafts. This involves the surveillance of aircrafts in airspaces, and the determination of the flight paths and the separation necessary. Separation distance may vary for a number of reasons, including the type of involved aircrafts, and the stages of journey they are at. For all airborne aircraft in a controlled airspace, human air traffic controllers (ATC operators) on the ground need to know where each aircraft is in the airspace in order to determine the flight paths.

One of the main requirements of the ATC systems is to ensure that a certain separation distance (SD) is maintained between aircrafts in the airspace controlled by an ATC system. The SD requirement needs to be satisfied by the system at all times. Furthermore, ATC operators can send various data and directions to the aircrafts using ATM system. Some of the data can potentially change the flight paths and are therefore security-related. One of such requirements considered in the rest of the paper is about sending weather data to the aircraft.

During the analysis of requirements such as this, the requirements engineer will have to describe the behaviour of various parts of the system, including the ATC operator, ATM system, the aircraft, and the pilot, identify the assets and security vulnerabilities in each of the components and their configuration, provide mitigation when necessary and show that the security requirements can be met by the system

through formal/informal arguments. In addition, when the system evolves, the behaviour of the system components and the requirements may change. Therefore, automated generation of partial descriptions of the system components facilitates the analysis process.

IV. META-MODEL

Our ESR meta-model, shown in Figure 1, combines concepts from requirements engineering, evolution and security analysis. Some of the key concepts in the meta-model are explained below.

Evolution Concepts: A model captures a *situation* at a given *time*, which contains a set of *contexts* and a set of *subjects*. The model can generally evolve by modifying the contexts and by introducing new subjects. When the time intervals are sufficiently small, the changes of the model are assumed to be small. Yet the impact of such smaller changes may still cover a large portion of the model. In order to show that important security requirements are satisfied after some change has been implemented, it is necessary to consider how the change should be propagated. In establishing that, the following concepts can be useful.

Requirements Engineering Concepts: A knowledge context has a set of *propositions*. A proposition can be assumed a fact or a rule, which is part of a given *domain* context, or can be supported by a set of propositions in an *argument* context. Both contexts and propositions are concerned with a set of *subject* matters. A subject matter in requirements engineering is either a *resource*, a *process* or an *actor*. A resource is a subject that may have multiple data *states*, a process is a subject that may have control behaviours that can be described by domains representing pre- or post-conditions, and an actor is a subject that may want *requirements* and may conduct some processes. A requirement relates an actor to a number of wanted propositions.

Problem Frames Concepts: The Problem Frames approach [1] emphasises the relationship between three main artefacts: a specification of a system (called machine), within a particular problem world context, satisfies a given requirement. *Fulfils* between a specification S and a requirement R can be represented by a logic entailment relation that $W, S \vdash R$, where W is the contexts in the situation. Some of the domains are physical domains with causal behaviour [1].

Temporal Logic Concepts: The temporal logic we use is a first-order predicate logic with discrete time. It has three main sorts: time, event and fluent (time varying property). Later in the discussion, we will explain how these concepts are used to describe requirements engineering artefacts.

Security and Argumentation Concepts: An *asset* is a resource that has desired value to the stakeholders (actors). It must be protected according to a *security goal* from damages that may be introduced by a potential *attack*. An *attacker* wants to achieve *anti-requirements*, which would *obstruct* the fulfilment of the security goals. An attack exploits *vulnerability* propositions inside the domains. By challenging the domain knowledge with additional propositions, a *rebuttal* is effectively constructed to demonstrate that the security requirements are not achievable under possible attacks.

A *mitigation* may introduce further changes to the domain knowledge such that the satisfaction argument of security requirements is valid again. Both rebuttals and mitigations are forms of arguments in different situations, hereby we choose not to represent them as separate concepts.

Table I: Elementary Predicates of the Event Calculus

Predicate	Meaning
Happens(a, t)	Action a occurs at time t
Initiates(a, f, t)	Fluent f starts to hold after action a at time t
Terminates(a, f, t)	Fluent f ceases to hold after action a at time t
HoldsAt(f, t)	Fluent f holds at time t
$t1 < t2$	Time point $t1$ is before time point $t2$

$$Clipped(t1, f, t2) \stackrel{\text{def}}{\equiv} \exists a, t[Happens(a, t) \wedge t1 \leq t < t2 \wedge Terminates(a, f, t)] \quad \text{(EC1)}$$

$$Declipped(t1, f, t2) \stackrel{\text{def}}{\equiv} \exists a, t[Happens(a, t) \wedge t1 \leq t < t2 \wedge Initiates(a, f, t)] \quad \text{(EC2)}$$

$$HoldsAt(f, t2) \leftarrow [Happens(a, t1) \wedge Initiates(a, f, t1) \wedge t1 < t2 \wedge \neg Clipped(t1, f, t2)] \quad \text{(EC3)}$$

$$\neg HoldsAt(f, t2) \leftarrow [Happens(a, t1) \wedge Initiates(a, f, t1) \wedge t1 < t2 \wedge \neg Declipped(t1, f, t2)] \quad \text{(EC4)}$$

$$HoldsAt(f, t2) \leftarrow [HoldsAt(f, t1) \wedge t1 < t2 \wedge \neg Clipped(t1, f, t2)] \quad \text{(EC5)}$$

$$\neg HoldsAt(f, t2) \leftarrow [\neg HoldsAt(f, t1) \wedge t1 < t2 \wedge \neg Declipped(t1, f, t2)] \quad \text{(EC6)}$$

Figure 2: Event Calculus Domain Independent rules

V. THE EVENT CALCULUS

t First introduced by Kowalski and Sergot [31], the Event Calculus (EC) is a system of logical formalism, which draws from first-order predicate calculus. It can be used to represent actions, their deterministic and non-deterministic effects, concurrent actions and continuous change [32]. We chose EC as our formalism, because it is suitable for describing and reasoning about event-based temporal systems such as the Air Traffic Management systems. Several variations

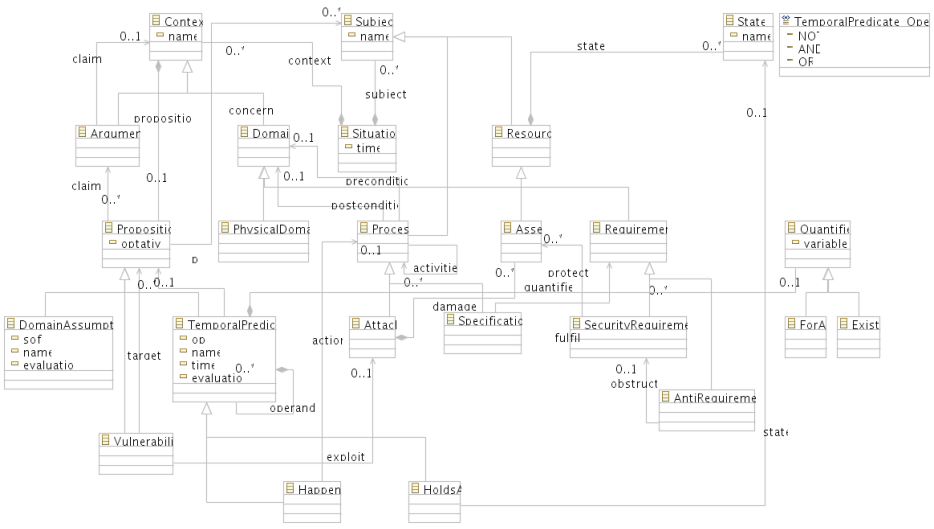


Figure 1: A ESR meta-model for Analysing Evolving Security Requirements

of EC have been proposed, and the version we have adopted here is based on the discussions in [33].

The calculus relates events and event sequences to *fluents*, or time-varying properties, which denote states of a system. Table 1, based on [33], gives the meanings of the elementary predicates of the calculus we use in this paper. The domain-independent rules in Fig. 2, taken from [33], state that: $Clipped(t1, f, t2)$ is a notational shorthand to say that the fluent f is terminated between times $t1$ and $t2$ (EC1), $Declipped(t1, f, t2)$ is another notational shorthand to say that the fluent f is initiated between times $t1$ and $t2$ (EC2), fluents that have been initiated by occurrence of an event continue to hold until occurrence of a terminating event (EC3), fluents that have been terminated by occurrence of an event continue not to hold until occurrence of an initiating event (EC4), and truth values of fluents persist until appropriate initiating and terminating events occur (EC5 and EC6).

Following Shanahan, we assume that all variables are universally quantified except where otherwise shown. We also assume linear time with non-negative integer values. In EC, we follow the rules of circumscription in formalizing commonsense knowledge [34], by assuming that all possible causes for a fluent are given in the database and our reasoning tool cannot find anything except those causes.

A. Describing ESR Models using the EC

In our approach to specifying event-based systems, requirements are constraints on the combinations of fluents capturing the required states of resources. Problem world domains capture behaviours of processes that are described by causality between pre- and post-conditions, called domain obligations. We now define them more formally.

Definition 5.1: Observations consist of a finite conjunction of $(\neg)HoldsAt$ predicates. *Reference phenomena* (Γ) are observations describing the given state of the resources or given action of the processes, while *controlled phenomena* (Γ') are observations describing the desired state of the resources or triggering of actions. They are also captured by the pre/post conditions of the process. A *requirement* is expressed either as

- ground observations Γ', without any reference to the given state of the resource or given action of the processes, or
- as a relationship between the reference and the controlled phenomena, such as a *constraint* of the form $\Gamma \rightarrow \Gamma'$, or an *action precondition axiom* of the form $(\neg)Happens(f1, t) \rightarrow \Gamma'$ where the antecedent is an occurrence of an action in the system (for example, to say that when an event a1 happens at time t, the fluent $f1$ must be true at $t1$).

For example, the requirement $HoldsAt(AircraftOnGround, t) \wedge 0 \leq t \leq 9$ says that the aircraft must be on the ground between the timepoints 0 and 9 range; the requirement $HoldsAt(Airborne, t) \rightarrow HoldsAt(TransponderOn, t)$ says that as long as the aircraft remains airborne, the transponder must be on; and the requirement $Happens(BreachSD, t) \wedge \neg Happens(Clearance, t2) \wedge t \leq t1 \leq t2 \rightarrow HoldsAt(AlarmRaised, t1)$ says that as soon as the separation distance is breached, the alarm should be raised until the clearance happens.

Since requirements tend to be about desired properties of the system over time, they will be formulated in terms of fluents holding, rather than in terms of (instantaneous) event occurrences. In other words, they specify 'what' the

system should achieve rather than 'how' the specification or processes achieve it.

Anti-requirements are requirements of an attacker, and the system must ensure that those requirements are not satisfiable. Since anti-requirements are also requirements, they can expressed in a similar way. For instance, $\exists t \cdot HoldsAt(AircraftsCollide, t)$, is a requirement of an attacker who wants to collide some aircraft.

B. Specifications/Obligations

We assume that a domain has the *potential* to generate instances of events it controls: it may generate all, some or none, of the event instances, even if that leads to undesired states.

In order to see its significance, we will briefly discuss some possible alternatives. In one other option, we may assume that the specification by default does not generate any event: In that case, if the specification describes events that must be generated, the specification is closed; if the specification describes events that may be generated, it is impossible to prove any "liveness" property. Similarly, we may assume that the specification by default generates all events, and specifications should restrict certain event occurrences. This again is not satisfactory for reasons similar to those given above. Therefore we have to categorize events into "must", "must not" and "may".

We therefore recognize three modes of describing specifications: in the *Act* mode, we describes events that must be generated (using Happens); and in the *Prohibit* mode, we describe events must not be generated (using Prohibit).

$$Prohibit(a, t1, t2) \stackrel{\text{def}}{\equiv} \neg\exists t \cdot Happens(a, t) \wedge t1 \leq t \leq t2 \quad \text{(EC7)}$$

In the third implicit mode, all other possible events are left undescribed because their occurrence or non-occurrence is assumed not to affect the requirement satisfaction.

In a *closed* specification, the union of "must" and "must not" covers all possible event sequences of the software (there is no event that may or may not happen). In a *partially open* specification, occurrence or non-occurrence of at least one event is not described: therefore there can be more than one specification that fulfil the requirement.

Definition 5.2: A *specification* is expressed as a finite conjunction of the *event occurrence constraints* (Ψ) of the form $(\neg)Happens(a_1, t) \wedge (\neg)HoldsAt(f, t) \rightarrow (\neg)Happens(a_2, t)$ where a_1, a_2, t, and f are terms for the action, time point, and fluent respectively.

Definition 5.3: A *domain description* in our approach to ATM is expressed as event-to-condition and condition-to-event causality. The first causality deals with what happens to the fluents when events occur, and the second causality deals with the domain properties that lead to the occurrence of certain events. In the Event Calculus, the event-to-condition causality is described as a finite conjunction *positive effect axioms* and *negative effect axioms* (Σ) of the form $Initiates(a, f, t) \leftarrow \Pi$ or $Terminates(a, f, t) \leftarrow \Pi$ where Π has the form $(\neg)HoldsAt(f_1, t) \wedge \cdots \wedge (\neg)HoldsAt(f_n, t)$ and t, and f_1 to f_n are terms for the time and fluents respectively. The condition-to-event causality is described as a finite conjunction of *trigger axioms* (Δ_2) of the form $Happens(a, t) \leftarrow \Pi$. For example, the following statement says that if the aircraft has transponder, an occurrence of the event interogateTransponder has an effect of making BroadcastACInfo true.

$$Initiates(interogateTransponder, BroadcastACInfo, t) \leftarrow HoldsAt(HasTranspnder, t)$$

Similarly, the following statement says that the fluent OperatorHasWeatherInfo on becoming true, generates the event sendWeatherInfo because of the functionality `SendWeatherInfo`.

$$Happens(sendWeatherInfo, t) \leftarrow HoldsAt(OperatorHasWeatherInfo, t) \wedge \neg HoldsAt(OperatorHasWeatherInfo, t-1)$$

Note that the condition $\neg HoldsAt(OperatorHasWeatherInfo, t-1)$ is necessary to prevent stuttering of the event sendWeatherInfo when the fluent OperatorHasWeatherInfo holds continuously.

C. Important Properties

Before we describe the important properties, we will make certain assumptions clear. First, these have to rely on the consistency of the domain theory Σ and observations Γ and Γ'. Second, we assume uniqueness of fluent and event names, meaning that no two names denote the same thing. This uniqueness axiom is represented by Ω.

A simple specification in this approach is a proactive specification that addresses a subtype of problem known as *Required Behaviour*. In this type of problem, a specification is required to bring about certain states in the system resources, without relying on the feedback from the other processes. In such cases, the basic property of the descriptions we want is:

$$\Sigma \wedge \Psi \models \Gamma'$$

That is, given a theory of physical domains (Σ), a specification (Ψ), and an appropriate deductive system, we want to show that the requirements are satisfied non-trivially.

In more common cases, the system has to rely on the feedback from the environment (Δ_2) and observations about the environment (Γ) .

$$\Sigma \wedge \Gamma \wedge \Delta_2 \wedge \Psi \models \Gamma'$$

D. Analysis

Finding vulnerabilities is done through logical abduction.

1) We first pose a *logical abduction* problem in order to find all constructive hypotheses (Δ_1) explaining how, given the domain theory ($\Sigma \wedge \Gamma \wedge \Delta_2$), the requirement ($\Gamma'$) can be satisfied, i.e.

$$CIRC[\Sigma; Initiates, Terminates] \wedge$$
$$CIRC[\Delta_1 \wedge \Delta_2; Happens] \wedge \Gamma \wedge \Omega \models \Gamma'$$

where Δ_1 is consistent with the domain theory. Δ_1 is a partially ordered sequences of event occurrences that, given the physical domains, leads to the requirement being satisfied. The circumscription operator assumes that no events other than those by Δ_1 and Δ_2 may occur (otherwise the requirement is not satisfied). Therefore, Δ_1 tells us events that must happen and that may happen. Event occurrences that do not appear in Δ_1 must not happen.

However, some of the hypotheses in Δ_1 may not be "realistic": for example, a scenario may assume competence and co-operation of users to a level that cannot be guaranteed. Furthermore, Δ_1 may also contain stuttering events that can be eliminated without affecting requirements satisfactions[1].

2) The developer then identifies the 'unrealistic' hypotheses in Δ_1 and eliminates them by providing further information about the problem world domains. Similarly, event stuttering is removed by adding further constraints (which is then used to weaken the specifications). These are assertions, or mitigation, we want the tool to consider.
3) When no vulnerabilities can be found, there are no "internal" vulnerabilities[2].

Notice that Ψ is not circumscribed: it will have to make explicit all events that must not happen. Therefore, Ψ describe all events that must happen, and all events that must not happen, the remainder being events that may happen.

E. Event Caclulus Reasoner

We choose Decreasoner to implement the verification for the generated EC rules. Decreasoner translates the EC rules into SAT formulae automatically, and invokes `Rel-Sat` solver to check whether they are satisfiable, given the bounded time range. In principle, abductive process may not terminate if the goal cannot be satisfied; however, since the time range is discrete and bounded, the tool forces a termination when the time limit is reached. Therefore, it is important to choose a reasonably large time range. This range, however, is not enlarged if a counter example can already be found.

[1]This is often called "invariance under stuttering" (for example [35]). Model checking techniques, known as "partial order reduction", exploit this property to reduce state space [36].

[2]These are vulnerabilities that can be found with the model.

```
1  grammar uk.ac.open.problem.Problem with uk.ac.open.Istar
2  import "platform:/resource/openome_model/model/openome_model.ecore" as
       openome_model
3  generate problem "http://open.ac.uk/problem"
4
5  ProblemDiagram: ("problem" ':' description=STRING)?
6          ((nodes+=Node|links+=Link))*;
7
8  Node:
9    name=ID (type=NodeType)?
10   (':' description=STRING)?
11   ("{" (subproblem=ProblemDiagram | "see" "domain" problemRef=[Node] |
12                 istar=Model | "see" "intention" istarRef=[openome_model::
                       Intention] |
13                 (hiddenPhenomena+=Phenomenon (',' hiddenPhenomena+=Phenomenon)*)
14   ) "}")?;
15
16 enum NodeType:
17   REQUIREMENT="R" | MACHINE="M" | BIDDABLE="B" | LEXICAL="X" | CAUSAL="C" |
           DESIGNED="D" | PHYSICAL="P";
18
19 Phenomenon:
20   (type=PhenomenonType)? name=ID (':' description=STRING)?;
21
22 enum PhenomenonType:
23   UNSPECIFIED="phenomenon" | EVENT="event" | STATE="state";
24
25 Link:
26   from=[Node] (type=LinkType) to=[Node] ('{' phenomena+=Phenomenon (',' phenomena
         +=
27   Phenomenon)* '}')? (':' description=STRING)?;
28
29 enum LinkType:
30   INTERFACE="-->" | REFERENCE="~~" | CONSTRAINT="->";
```

Figure 3: Partial listings of the concrete syntax of the the ESR meta-model in Figure 1

VI. OpenPF

This section explains how Event Calculus templates are generated from Problem Frame diagrams using OpenPF.

A. Concrete Syntax

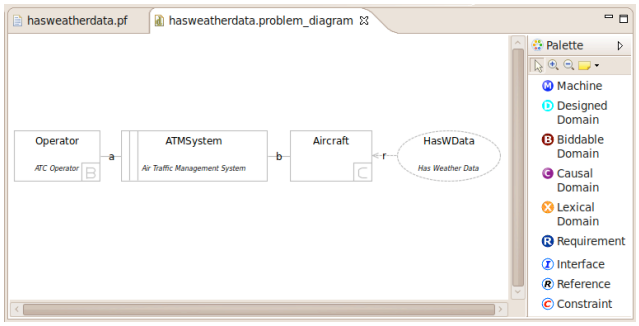
Figure 3 lists the most of concrete syntax for the Problem Frames concepts in the EMF meta-model shown in Figure 1. This concrete syntax is given in order to show the textual representations of the meta-model, and also to indicate that the root concept in the representation is the problem diagram.

The syntax is composed of a number of BNF-like rules, each defines one non-terminal at the left hand side and a number of refinement parsable elements, including both non-terminals and terminals. The words occurring as strings in the rules are treated as keywords, which is only necessary in the concrete syntax. For the same abstract syntax in Figure 1, there can be more than one way to express the concrete syntax. In this example, we try to express it using intuitive keywords consistently. Comparing the graph-based meta-model with the tree-based concrete syntax, one useful feature of `xtext` is to use `ID` to provide shared references inside other types.

An example of the ESR model is given in Figure 4. As the example shows, the syntax for writing a problem diagram is straightforward. First, the name of the problem diagram ("Has Weather Data") is defined (Lines 1-2). "Has Weather Data" is a requirement node (indicated by R) and is identified by HasWData. Aircraft is a causal domain with no

```
problem:
"Has_Weather_Data"

HasWData R: "Has_Weather_Data"

Aircraft C

ATMSystem M: "Air_Traffic_Management_System"

Operator B: "ATC_Operator"

Operator -> ATMSystem: "a"

ATMSystem -> Aircraft: "b"

HasWData -> Aircraft: "r"
```

Figure 4: Full listings of one concrete example of the ESR model



a: O!{InputWeatherData}
b: AS!{SendWeatherData}
r: A!{HasWeatherData}

Figure 5: Problem Diagram: Has Weather Data

description. Air Traffic Management System is a machine domain, whilst ATC Operator is a biddable domain. The interface between the operator and ATM System is named a, and the interface between ATM System and aircraft is named b. The requirement constrains the aircraft domain, and is named r.

B. Diagramming

OpenPF also provides editor and automatic diagramming of problem diagrams from the syntax given above. A problem diagram for the problem of sending weather data is shown in Figure 5, together with a description of the events.

C. Generating Event Calculus Templates

Generating Event Calculus templates from the problem diagram is done using the OMG Text-to-Model (Xtext) and Model-to-Text transformation (Acceleo) framework inside the Eclipse modelling Project. The expression syntax of the Model-to-Text transformation is similar to that of OCL, and the concrete syntax resembles the JavaScript language, except that code generation tags are enclosed by square brackets rather than by XML brackets.

First, it is necessary to determine the output file name and generate a common header to include the predefined

```
1  [module generate('http://open.ac.uk/problem')/]
2  [template public generate(d: ProblemDiagram)]
3  [file (d.description.toString().concat('.ec'), false)]
4  load foundations/Root.e
5  load foundations/EC.e
6      [for (dom: Node | d.nodes)]
7        [for (hidden: Phenomenon | dom.hiddenPhenomena)]
8         [if (hidden.type.toString() ='state')]
9  fluent [hidden.name/]_[dom.name/]()
10        [/if]
11        [/for]
12       [/for]
13       [for (interface: Link | d.links)]
14       [for (shared: Phenomenon | interface.phenomena)]
15                [if (shared.type.toString() = 'event')]
16 event [shared.name/]_[interface.description/]()
17         [/if]
18                [if (shared.toString() = 'state')]
19 fluent [shared.name/]_[interface.description/]()
20         [/if]
21       [/for]
22     [/for]
23     [for (dom: Node | d.nodes)]
24      [if (dom.type.toString() <> 'R')]
25 ;--[dom.name/]--
26 ['['/]time[']'/]
27       [for (interface: Link | d.links)]
28        [if (interface.to = dom)]
29         [for (shared: Phenomenon | interface.phenomena)]
30                  [if (shared.type.toString() = 'event')]
31 Happens([shared.name/]_[interface.description/](),time),
32          [/if]
33                [if (shared.toString() = 'state')]
34 HoldsAt([shared.name/]_[interface.description/](),time),
35         [/if]
36         [/for]
37         [/if]
38        [/for]
39         ->
40       [for (interface: Link | d.links)]
41        [if (interface.from = dom)]
42         [for (shared: Phenomenon | interface.phenomena)]
43                  [if (shared.type.toString() = 'event')]
44 Happens([shared.name/]_[interface.description/](),time+1),
45          [/if]
46                [if (shared.toString() = 'state')]
47 HoldsAt([shared.name/]_[interface.description/](),time+1),
48         [/if]
49         [/for]
50         [/if]
51       [/for].
52      [/if]
53    [/for]
54 range time 0 3
55 range offset 1 2
56 [/file]
57 [/template]
```

Figure 6: Part of listings of the code generation rules: transforming from the ESR meta-model into EC

EC rules (Line 3-5). Names of fluents that are internal to the domains are identified, suffixed with the domain names, and declared (Line 9). Similarly, names of events and fluents shared between domains are also identified, suffixed with the domain names, and declared (Line 16, 19). This suffixing of the fluent and event names ensures that it is possible to trace the results of the reasoning tool back to specific parts of the problem diagrams.

Apart from the common footer (Lines 54-55), the rest of the template is pattern-based. Some of the patterns are now discussed. A typical instance of the domain specification (Definition 5.2) is if event e1 happens at time t, then another event e2 happens at a time after t, t+1 (Lines 25-53). Given the diagram matching this pattern, a transformation is applied to generate the following rule snippet for the template:

```
; -- Domain --
[time] Happens(e1, t) ->
       Happens(e2, t + 1).
```

Here the "[time]" denotes a universally quantified variable. Note that this is a generic template in which one can change the time delay between the two event occurrences. Generally speaking, this pattern will put all events and fluents a domain observes on the left-hand side, and all events and fluents a domain controls on the right-hand side of an implication.

Formulae for event-to-fluent and fluent-to-event causalities are also common (Definition 5.3). For this causality, typically there is a domain with a fluent and two events, where one of the events is observed by the domain, and the other event is controlled by the domain. In such cases, the pattern will produce Initiates and Terminates formulae, and Happens formulae.

As a result of these transformations, syntactically correct EC descriptions are generated, which can be further modified by the tool users before analysing them.

VII. Example Analysis and Results

We now work through the example introduced in Section III. First, we describe the abstract domain behaviour in terms of its obligations. We begin by drawing a simple diagram showing the problem world domains, their relationships and the property the system needs to hold. The diagram (Figure 5) shows that the operator sends weather information to the aircraft using the ATM system.

A. Describing Specifications/Obligations

We then describe the behaviour of the domains in terms of the events they observe and events they control. OpenPF templates mentioned above produces the following descriptions.

```
; -- AS1 --
[time] Happens(InputWeatherData_a(),time) ->
       Happens(SendWeatherData_b(), time+1).
```

When the ATM system observes the input, it will send the weather information to the aircraft at the next time point. Similar descriptions can be generated for the other domains. For the operator, we may assume that when the operator is told about weather information is available, s/he inputs the information to the ATM system at the next time point.

```
; O1 -- Operators --
[time] Happens(TellWeatherInfo,time) ->
       Happens(InputWeatherInfo, time+1).

; A1 -- Aircraft --
[time] Initiates(SendWeatherInfo_f(),
                 HasWeatherInfo_Aircraft(), time).
```

When the aircraft observes the weather information being sent, the aircraft will has the aircraft information.

B. Describing Domain Behaviour

Let us suppose that the operator has the following behaviour.

```
; O2
[time] Initiates(ReceiveWeatherData,
                 WeatherData_Known, time).
; O3
[time] !HoldsAt(WeatherData_Known,time)
         & HoldsAt(WeatherData_Known,time +1) ->
           Happens(InputWeatherData,time + 1).
```

The first statement says that receiving weather data by the operator means that the weather data is known to the operator. The second statement says that as soon as the operator knows weather information, the InputWeatherData event is generated.

In order to define the partial Behaviour of the aircraft, we first define a few additional sorts.

```
fluent Collided(ac,ac)
fluent At(ac,pos)
event Move(ac,pos,pos)
fluent Avoid(pos)
```

The fluent Collided(ac,ac) is true when two aircraft collided; the fluent At(ac,pos) is true when the aircraft is at the position; the fluent Avoid(pos) is true when no aircraft is at the position; and Move(ac,pos,pos) says that the aircraft moves from one position to another.

```
[time,ac,pos,pos1]
Initiates(Move(ac,pos,pos1),At(ac,pos1),time).

[time,ac,pos,pos1]
Terminates(Move(ac,pos,pos1),At(ac,pos),time).

[time,ac,pos,pos1]
Happens(Move(ac,pos,pos1),time) -> (pos<pos1).

[time,pos,pos1,ac]
HoldsAt(At(ac,pos),time) &
HoldsAt(At(ac,pos1),time) -> (pos=pos1).

[time,ac,ac1,pos,pos1]
HoldsAt(At(ac,pos),time+1) & (ac!=ac1) ->
Initiates(Move(ac1,pos1,pos),
          Collided(ac,ac1),time).

[time,ac,ac1,pos,pos1]
!HoldsAt(At(ac,pos),time+1) & (ac!=ac1) ->
Terminates(Move(ac1,pos1,pos),
           Collided(ac,ac1),time).

[time,ac,ac1]
HoldsAt(Collided(ac, ac1),time) -> (ac!=ac1).

[time,ac,ac1]
HoldsAt(Collided(ac, ac1),time) <->
HoldsAt(Collided(ac1, ac),time).

[time,ac,ac1,pos,pos1]
(time=0) -> (HoldsAt(At(ac,pos),time) &
HoldsAt(At(ac1,pos1),time) & (ac!=ac1) ->
(pos!=pos1)).

[time,ac,pos] HoldsAt(Avoid(pos),time) ->
!HoldsAt(At(ac,pos),time).
```

The above formulae describe how planes move along paths, and when happens when planes converge on a position. Depending on the weather information received from the operator, various constraints can be placed on the flight by stating positions that need to be avoided. For instance, the following says that the position 1 should not be on the flight path.

```
HoldsAt(Avoid(1),1).
```

C. Analysing Domain Obligation

First, we can check that there is at least one model of the operator behaviour (O2 and O3) that satisfies its specification (O1). Here the tool finds several models including the following:

```
0
Happens(ReceiveWeatherData(t), 0).
1
Happens(InputWeatherData(g), 1).
2
Happens(InputWeatherData(g), 2).
Happens(SendWeatherData(f), 2).
3
+HasWeatherData_Aircraft().
P
```

At time 1, the operator receives the weather information, which the operator inputs at the next time point. The weather information is sent at time 2, and the aircraft has the weather information at time point 3.

Next, we can check whether the operator behaviour can fail to satisfy its obligation. Again, the tool finds several models showing how the operator can fail to satisfy his/her obligations, including the following:

```
0
WeatherDataKnown_Operator().
Happens(ReceiveWeatherData(g), 0).
1
Happens(ReceiveWeatherData(g), 1).
2
Happens(ReceiveWeatherData(g), 2).
3
P
```

In one of the models, the operator may know the weather information, and still receive weather information, but fails to input the information to the ATM system. This is a case of operator witholding the information. This is a rebuttal generated by the tool.

The rebuttal shows that the domain behaviour allows the operator to input the weather information without being told. This of course poses a security risk, if the operator has malicious intent. This calls for a strengthening of the domain behaviour by stating that the operator will send if and only if s/he was told about the weather information.

```
; O3'
[time] !HoldsAt(WeatherData_Known,time)
        & HoldsAt(WeatherData_Known,time + 1) <->
          Happens(InputWeatherData,time+1).
```

In this case, the domain obligation is weaker than the domain behaviour. Finally as a mitigation to this problem, we can strengthen the domain obligations.

```
; O1'
[time] Happens(TellWeatherData,time) <->
       Happens(InputWeatherData, time+1).
```

Once the domain behaviour and the obligations are strengthened, it is no longer possible to show that the operator can fail to satisfy his/her obligation. The strengthening is mitigation.

The security requirement to prevent collision can be checked in the same way.

VIII. Conclusion

We have investigated some of the challenges of analysing the security impact of evolutionary changes made to software systems. First, we applied a meta-model of evolving security requirements, which draws on concepts in requirements engineering, security analysis, argumentation and software evolution. We instantiated the meta-model using a formalism of temporal logic, called the Event Calculus. We have proposed a tool called OpenPF that generates templates for Event Calculus descriptions of the evolving system, and analyse them using a reasoning tool called Decreasoner. The approach is illustrated with a simple example from an Air Traffic Management system.

Acknowledgement

Financial support of the SecureChange project, funded by the European Union, is gratefully acknowledged. We thank the industrial and academic partners of the SecureChange project for providing case studies and feedback on ideas leading to this work.

References

[1] M. Jackson, *Problem Frames: Analyzing and structuring software development problems*. Addison Wesley, 2001.

[2] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave, "A reference model for requirements and specifications," *IEEE Softw.*, vol. 17, no. 3, pp. 37–43, 2000.

[3] D. L. Parnas and J. Madey, "Functional documents for computer systems," *Sci. Comput. Program.*, vol. 25, no. 1, pp. 41–61, 1995.

[4] I. Jureta, J. Mylopoulos, and S. Faulkner, "Revisiting the core ontology and problem in requirements engineering," in *RE '08: Proceedings of the 2008 16th IEEE International Requirements Engineering Conference*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 71–80.

[5] M. Hartong, R. Goel, and D. Wijesekera, "Meta-models for misuse cases," in *CSIIRW '09: Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research*. New York, NY, USA: ACM, 2009, pp. 1–4.

[6] A. Susi, A. Perini, J. Mylopoulos, and P. Giorgini, "The tropos metamodel and its use," *Informatica (Slovenia)*, vol. 29, no. 4, pp. 401–408, 2005.

[7] A. van Lamsweerde, "Elaborating security requirements by construction of intentional anti-models," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 148–157.

[8] G. Elahi, E. S. K. Yu, and N. Zannone, "A modeling ontology for integrating vulnerabilities into security requirements conceptual foundations," in *ER*, ser. Lecture Notes in Computer Science, A. H. F. Laender, S. Castano, U. Dayal, F. Casati, and J. P. M. de Oliveira, Eds., vol. 5829. Springer, 2009, pp. 99–114.

[9] G. Beydoun, G. Low, H. Mouratidis, and B. Henderson-Sellers, "A security-aware metamodel for multi-agent systems (mas)," *Inf. Softw. Technol.*, vol. 51, no. 5, pp. 832–845, 2009.

[10] P. Savolainen, E. Niemela, and R. Savola, "A taxonomy of information security for service-centric systems," in *EUROMICRO '07: Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 5–12.

[11] S.-W. Lee, R. Gandhi, D. Muthurajan, D. Yavagal, and G.-J. Ahn, "Building problem domain ontology from security requirements in regulatory documents," in *SESS '06: Proceedings of the 2006 international workshop on Software engineering for secure systems*. New York, NY, USA: ACM, 2006, pp. 43–50.

[12] P. Giorgini, F. Massacci, and N. Zannone, "Security and trust requirements engineering," in *Foundations of Security Analysis and Design III*, 2005, pp. 237– 272.

[13] A. van Lamsweerde, "Elaborating security requirements by construction of intentional anti-models," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering, Washington, DC, USA*, 2004, pp. 148– 157.

[14] A. Nhlabatsi, B. Nuseibeh, and Y. Yu, "Security requirements engineering for evolving software systems: A survey," *Journal of Secure Software Engineering*, vol. 1, pp. 54–73, 2009.

[15] P. Giorgini, J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani, "Reasoning with goal models," in *Conceptual Modeling ER 2002*, 2003, pp. 167–181.

[16] R. S. Arnold, "Software change impact analysis," *Computer*, 1996.

[17] C. F. Kemerer and S. Slaughter, "An empirical approach to studying software evolution," *IEEE Trans. Softw. Eng.*, vol. 25, pp. 493–509, 1999.

[18] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Trans. Softw. Eng.*, vol. 29, pp. 210–224, 2003.

[19] R. Conradi and B. Westfechtel, "Version models for software configuration management," *ACM Comput. Surv.*, vol. 30, pp. 232–282, 1998.

[20] Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture- based approach to self-adaptive software," *IEEE Intelligent Systems*, vol. 14, pp. 54–62, 1999.

[21] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Trans. Softw. Eng.*, vol. 30, pp. 574–586, 2004.

[22] A. Egyed, "Fixing inconsistencies in uml design models," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007.

[23] C. Jones, "Software change management," *Computer*, vol. 29, pp. 80–82, 1996.

[24] P. Besnard and A. Hunter, "Argumentation based on classical logic," in *Argumentation in Artificial Intelligence*, 2009, pp. 133–152.

[25] ——, "Practical first-order argumentation," in *Proceedings of the 20th national conference on Artificial intelligence - Volume 2 Pittsburgh, Pennsylvania: AAAI Press*, 2005.

[26] C. I. Chesnevar, A. G. Maguitman, and R. P. Loui, "Logical models of argument," *ACM Comput. Surv.*, vol. 32, pp. 337–383, 2000.

[27] G. Governatori, M. J. Maher, G. Antoniou, and D. Billington, "Argumentation semantics for defeasible logic," *J Logic Computation*, vol. 14, pp. 675–702, 2004.

[28] I. Jureta, J. Mylopoulos, and S. Faulkner, "Analysis of multi-party agreement in requirements validation," in *17th IEEE International Requirements Engineering Conference (RE'09), Atlanta, GA, USA, 2009,*, 2009, pp. 57 – 66.

[29] I. Habli, W. Wu, K. Attwood, and T. Kelly, "Extending argumentation to goal-oriented requirements engineering," in *Advances in Conceptual Modeling Foundations and Applications*, 2007, pp. 306–316.

[30] C. B. Haley, R. C. Laney, J. D. Moffett, and B. Nuseibeh, "Security requirements engineering: A framework for representation and analysis," *IEEE Trans. Software Eng.*, vol. 34, pp. 133–153, 2008.

[31] R. Kowalski and M. Sergot, "A logic-based calculus of events," *New Gen. Comput.*, vol. 4, no. 1, pp. 67–95, 1986.

[32] M. Shanahan, "The event calculus explained," *Lecture Notes in Computer Science*, vol. 1600, pp. 409–430, 1999.

[33] R. Miller and M. Shanahan, "The event calculus in classical logic - alternative axiomatisations," *Electronic Transactions on Articial Intellligence*, vol. 3, pp. 77–105, 1999.

[34] J. McCarthy, *Formalization of common sense, papers by John McCarthy edited by V. Lifschitz*. Ablex, 1990.

[35] L. Lamport, "What good is temporal logic?" in *IFIP Congress*, 1983, pp. 657–668.

[36] D. Peled and T. Wilke, "Stutter-invariant temporal properties are expressible without the next-time operator," *Inf. Process. Lett.*, vol. 63, no. 5, pp. 243–246, 1997.

APPENDIX B. Dealing with Known Unknowns: Towards a Game-Theoretic Foundation for Software Requirement Evolution

Dealing with Known Unknowns: Towards a Game-Theoretic Foundation for Software Requirement Evolution *

Le Minh Sang Tran and Fabio Massacci

Università degli Studi di Trento, I-38100 Trento, Italy
{tran, fabio.massacci}@disi.unitn.it

Abstract. Requirement evolution has drawn a lot of attention from the community with a major focus on management and consistency of requirements. Here, we tackle the fundamental, albeit less explored, alternative of modeling the future evolution of requirements.
Our approach is based on the explicit representation of *controllable* evolutions vs *observable* evolutions, which can only be estimated with a certain probability. Since classical interpretations of probability do not suit well the characteristics of software design we also introduce a game-theoretic approach to give an explanation to the semantic behind probabilities. Based on this approach we also introduce quantitative metrics to support the choice among evolution-resilient solutions for the system-to-be. To illustrate and show the applicability of our work, we present and discuss examples taken from a concrete case study (the security of the SWIM system in Air Traffic Management).

1 Introduction

> "...There are known unknowns: that is to say, there are things that we now know we don't know..."
>
> — Donald Rumsfeld, United States Secretary of Defense

In the domain of software, evolution refers to a process of continually updating software systems in accordance to changes in their working environments such as business requirements, regulations and standards. While some evolutions are unpredictable, many others can be predicted albeit with some uncertainty (e.g. a new standard does not appear overnight, but is the result of a long process).

The term *software evolution* has been introduced by Lehman in his work on laws of software evolution [17, 18], and was widely adopted since 90s. Recent studies in software evolutions attempt to understand causes, processes, and effects of the phenomenon [2,14,16,22]; or focus on the methods, tools that manage the effects of evolution [19,25,28]. The ultimate objective is to make the software systems more resilient to evolution.

* This work is supported by the European Commission under projects EU-FET-IP-SECURECHANGE.

Requirement evolution has also been the subject of significant research [12, 15, 24, 26, 31]. However, to our understanding, most of these works focus on the issue of management and consistency of requirements. Here, we tackle the more fundamental question of modeling uncertain evolving requirements in terms of evolution rules. Our ultimate goal is to support the decision maker in answering such a question "Given these anticipated evolutions, what is a solution to implement an evolution-resilient system?".

This motivates our research in modeling and reasoning on a requirement model of a system which might evolve sometime in the future. We assume that stakeholders will know the tentative possible evolutions of the system-to-be, but with some uncertainty. For example, the Federal Aviation Authority document of the forthcoming service middleware SWIM for air traffic management lists a number of potential alternatives subject to other high-level decisions (eg the existence of an organizational agreement for nation-wide identity management of SWIM users). Such organization-level agreements don't happen overnight (and may shipwreck at any time) and stakeholders with experience and high-level positions have a clear visibility of the likely alternatives, the possible but unlikely solutions, and the politically impossible alternatives.

Our objective is to model the evolution of requirements when it is known to be possible, but it is unknown whether it will happen: the *known unknown*.

1.1 The contributions of this paper

We set up a game-theoretic foundation for modeling and reasoning on evolutionary requirement models:

- A way to model requirement evolutions in term of two kinds of evolution rules: *controllable* and *observable* rules that are applicable to many requirement engineering models (from problem frames to goal models).
- A game-theoretic based explanation for probabilities that evolutions happen.
- Two quantitative metrics that assess the certainty of requirement models, which provide clue for designer to make decision on optimal things to implement in to system-to-be.

The intuition behind controllable rules is that they are design decisions under the control of the designer. In contrast, observable rules corresponds to potential evolution whose realization is outside the control of the designer but that can be somewhat estimated by the stakeholders.

The game-theoretic semantics of probability has been introduced since the frequentist semantics of probabilities makes little sense for software design: mandating this or that requirements is likely a unique event and there is no random variable to be sampled repeatedly, as one would find in a process of toin cossing.

In the rest of the paper, we describe a case study (§2) which we use throughout this work. Next, we discuss our approach (§3) in which we model requirement evolutions in terms of evolution rules. We then present our proposed quantitative metrics (§4) for reasoning on evolutionary requirement models. Next, we address the problems of handling large requirement models and iterative evolution (§5). Finally, we briefly review past work in the field and conclude our work (§6).

Table 1. High level requirements of SWIM Security Services.

ID	Requirement	Opt.
RE1	Manage keys and identities of system entities (human, software, devices,...)	
RE2	Support Single Sign-On (SSO)	•
RE3	Support a robust Identity and Key Management Infrastructure (IKMI) that can be scaled up to large number of applications and users.	•
RE4	Intrusion detection and response	
RB1	Less cross-program dependencies for External Boundary Protection System	
RB2	More robust and scalable common security solution	•
RB3	Simpler operation of External Boundary Protection System	•
RB4	Support overall security assessment	•

The Opt(ional) column determines whether a requirement is compulsory or not. Notice that this compulsory is only applicable to the time system is designed. Due to evolution, optional requirements may turn to be compulsory, and current compulsory requirements may no longer be needed or become obsolete in the future.

2 Case study

Throughout this work, to give a clearer understanding of the proposed approach we draw examples taken from the design architecture of System Wide Information Management (SWIM) [7, 23] in Air Traffic Management (ATM).

SWIM should provide a secure, overarching, net-centric data network, and introduces a Service-Oriented Architecture (SOA) paradigm to airspace management. The objective of SWIM is to "decouple producer of information from the possible consumer in such a way that the number and nature of consumers can evolve through time" [23, p.16]. This approach provides a practical, low-cost solution to integrate seamlessly legacy heterogeneous ATM systems, while facilitating the advantages of the next generation of ATM systems. This also means the adaptability to the evolution of ATM system is increased.

The US FAA [7] proposes a logical high level functional architecture of SWIM which consists of several function blocks. Our examples will focus on the Security Services block, one of the essential elements of SOA Core Services of SWIM. Basically, Security Services have two major functions: policy enforcement and monitoring. The former function consists of enforcing policies regarding access to services and data resources. The later consists of monitoring National Air Space (NAS) services to detect security breach or fraudulent use of resources.

At high level analysis of security capabilities, there are five areas of security. They are: *i)* NAS Enterprise Information Security System (ISS-ENT), *ii)* Boundary Protection ISS (ISS-BP), *iii)* SWIM Core ISS, *iv)* NAS End System ISS, and *v)* Registry control. To avoid a detailed discussion on the architecture of SWIM Secure Services, which are not main topic of this work, while providing enough information for illustrating our work we refine our scope of interest on two areas: ISS-ENT and ISS-BP. Below, we describe a sketch overview of these areas.

– ISS-ENT includes security requirements that are provided as part of an underlying IT/ISS infrastructure used by systems throughout the NAS.

Table 2. Design elements that support requirements listed in Table 1.

ID	Element Description	RE1	RE2	RE3	RE4	RB1	RB2	RB3	RB4
A	Simple IKMI	•							
B1	OpenLDAP based IKMI	•		•					
B2	Active Directory based IKMI	•	•	•					
B3	Oracle Identity Directory based IKMI	•	•	•			•		
C	Ad-hoc SSO		•						
D	Network Intrusion Detection System				•				
E	Common application gateway for External Boundary Protection System							•	•
F	Centralized Policy Decision Point (PDP)						•		
G	Application-based solution for External Boundary Protection System					•			

Each element in this table can support (or fulfill) requirements listed in columns. To prevent useless redundant, some elements are exclusive to due to functionality overlapping (*e.g.,* A, B1, B2 and B3 are mutual exclusive each other).

– ISS-BP includes requirements regarding control connections and information exchanges between internal NAS and external entities. These requirements refer to both network layer controls (*e.g.,* VPNs, firewalls) and application layer controls (gateway application that controls the connection between NAS and non-NAS systems, and ensures that the content of data packages are conformed predefined rules before allowing them to pass).

Table 1 divides requirements in two groups: first, the essential requirements at the time that the system is designed; second, the optional requirements that can be ignored at present, but might be critical sometime in the future. For convenience, each requirement has a corresponding identifier: two characters for the security area (RE - stand for ISS-ENT requirements, RB - stand for ISS-BP ones), and a sequence number.

Finally we show solutions to requirements of ISS-ENT and ISS-BP listed in the previous table. Each solution occupies a row in Table 2. Each solution has an IDentifier, a short description and checklist of requirements that it can fulfill.

3 Modeling Evolution Requirements

In this section, we describe how we model evolution, which essentially affects to any further analysis. We capture the evolution by classifying them into two groups: ***controllable*** and ***observable***. Furthermore, we include in this section the game-theoretic account for probability.

3.1 Evolution on requirement model: Controllable and Observable

Stakeholder requirements, mostly in a textual format, are stakeholder wishes of the system-to-be. Analysis on these requirements in such a format is difficult and not efficient. Designers then have to model requirements and design decision by

using various approaches (*e.g.,* model-based, process-based, or goal-based) and techniques (*e.g.,* DFD, UML).

In general, a requirement model basically is a set of elements and relationships, which varies depended on particular approach. For instance, according to Jackson and Zave [30], model elements are *Requirements*, *Domain assumptions*, *Specification*. In goal-based models (*e.g.,* i*), elements are goal, actor and so on.

Here we do not want to investigate on any specific requirement model (*e.g.,* goal-based model, UML models), or go to detail about how many kinds of element and relationship a model would have. The choice of a one's favorite model to represent these aspects can be as passionate as the choice of a one's religion or football team so it is out of scope. Instead, we treat elements at abstract meaning, and only interest in the satisfaction relationship among elements.

In our work, we define the satisfaction relationship in terms of usefulness. That an element set X is useful to another element set Y depends on the ability to satisfy (or fulfill) Y if X is satisfied. We define a predicate useful(X, Y) which returns true (1) if X can satisfy all elements of Y, otherwise return false (0). The implementation of useful is depended on the chosen specific model. For examples:

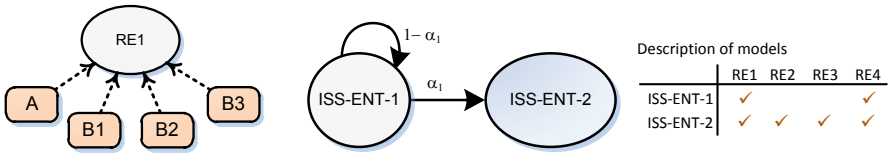
- Goal models [20]: the useful corresponds to Decomposition and Means-end relationships. The former denotes a goal can be achieved by satisfying its subgoals. The later refers to achieving a (end) goal by performing (means) tasks.
- Problem frames [13]: the useful corresponds to requirement references and domain interfaces relationships. Requirements are imposed by machines, which connect to problem world via domain interfaces. Problem world in turn connects to requirements via requirement references.

For evolutionary software systems which may evolve under some circumstances (*e.g.,* changes in requirements due to changes in business or regulations, wrong domain assumption), their requirement models should be able to express as much as possible information about known unknowns *i.e.* potential changes. These potential changes are analyzed with evolution assessment algorithms to contribute to decision making process, where a designer decides what are going to the next phase in the development process.

Based on a person who can decide these evolutions happen but those are not. We categorize requirement evolutions into two classes:

- *controllable evolutions*, in which the designer can decide which evolution to follow in order to meet some high level requirements from the stakeholder, with low level requirements for component.
- *observable evolutions* which are not under the control of the designer, but it can be somehow detected when it happened or whose future likelihood can be estimated with a certain confidence of the stakeholder.

Controllable evolutions, in other words, are designer's moves to identify different alternatives for implementing a system. The designer then can choose the most "optimal" one based on her experiment and some analyses on these alternatives. In this sense, controllable evolution is also known as design choice.



(a) Controllable rule (b) Observable rule

Fig. 1. Example of controllable rule (a), and observable rule (b).

Observable evolutions, in contrast, correspond to moves of reality to decide how the requirement model looks like in the future. Therefore, designer has to forecast the reality's choice with a level of certainty. The response of the designer can be forecasted and thus incorporated into the design (in order to have an unattended evolution) or can be deployed at run-time.

We capture the evolution in terms of evolution rule. We have *controllable* and *observable rule* corresponding to controllable and observable evolution.

Definition 1. *A controllable rule r_c is a set of tuples $\langle RM, RM_i \rangle$ that consists of an original model RM and its possible design alternative RM_i.*

$$r_c = \bigcup_{i}^{n} \left\{ RM \xrightarrow{*} RM_i \right\}$$

Definition 2. *An observable rule r_o is set of triple $\langle RM, p_i, RM_i \rangle$ that consists of an original model RM and its potential evolution RM_i. The probability that RM evolves to RM_i is p_i. All these probabilities should sum up to one.*

$$r_o = \bigcup_{i=1}^{n} \left\{ RM \xrightarrow{p_i} RM_i \right\}$$

Fig. 1 is a graphical representation of evolution rules taken from SWIM case study. Left, Fig. 1(a) describes a controllable rule that a requirements of IKMI (RE1) has four design choices: A, B1, B2, and B4 (see Table 1 and Table 2). Right, Fig. 1(b) shows that the initial requirement model ISS-ENT-1 (including RE1 and RE4) can involve from ISS-ENT-2 (including RE1 to RE4), or remain unchanged with the probabilities of α and $1 - \alpha$. The formulae of these rules are as follows:

$$r_c = \left\{ RE1 \xrightarrow{*} A, RE1 \xrightarrow{*} B1, RE1 \xrightarrow{*} B2, RE1 \xrightarrow{*} B3 \right\}$$

$$r_o = \left\{ \mathsf{ISS\text{-}ENT\text{-}1} \xrightarrow{\alpha_1} \mathsf{ISS\text{-}ENT\text{-}2}, \mathsf{ISS\text{-}ENT\text{-}1} \xrightarrow{1-\alpha_1} \mathsf{ISS\text{-}ENT\text{-}1} \right\}$$

3.2 Game-theoretic account of Probability

In this section, we discuss on probability interpretations and why and how we employ game-theoretic (or betting interpretation) to account for the probability of occurrence of observable rule.

As mentioned, each potential evolution in observable evolution has an associate probability; probabilities of all potential evolutions of an observable rule should sum to 1. However, who tells us these? And what is the semantic of probability? To answer the first question, we, as system Designers, agree that Stakeholder will be the one who tells us possible changes in a period of time. About the second question, we need an interpretation for semantic of probability.

Basically, there are two broad categories of probability interpretation, which can be called "physical" and "evidential" probabilities. Physical probabilities, in which frequentist is a representative, are associated with a random process. Evidential probability, also called Bayesian probability (or subjectivist probability), are considered to be degrees of belief, defined in terms of disposition to gamble at certain odds; no random process is involved in this interpretation.

To account for probability associated with an observable rule, we can use the Bayesian probability as an alternative to the frequentist because we have no event to be repeated, no random variables to be sampled, no issues about measurability (the system that the designer is going to build is often unique in some respects). However, we need a method to calculate the value of probability as well as to explain the semantic of the number. Since the value of probability is acquired from the requirement eliciting process involving the stakeholder, we propose using the game-theoretic method in which we treat probability as a price. This would be easier for stakeholder to reason in terms of price (or cost) rather than probability.

The game-theoretic approach, discussed by Shafer et al. [27] in Computational Finance, begins with a game of three players, *i.e.* Forecaster, Skeptic, and Reality. Forecaster offers prices for tickets (uncertain payoffs), and Skeptic decides a number of tickets to buy (even a fractional or negative number). Reality then announces real prices for tickets. In this sense, probability of an event E is the initial stake needed to get 1 if E happens, 0 if E does not happen. In other words, the mathematics of probability is done by finding betting strategies.

In this paper, we do not deal with stock market but the design of evolving software, *i.e.* we extend it for software design, we then need to change the rules of the game. Our proposed game has three players: *Stakeholder*, *Designer*, and *Reality*. For the sake of brevity we will use "he" for the Stakeholder, "she" for the Designer and "it" for Reality. The sketch of this game is denoted in protocol 1.

Protocol 1
Game has n round, each round plays on a software C_i
FOR $i = 1$ *to* n
    Stakeholder announces p_i
    Designer announces her decision d_i*: believe, don't believe*
    If Designer believes
        $K_i = K_{i-1} + M_i \times (r_i - p_i)$
    Designer does not believe
        $K_i = K_{i-1} + M_i \times (p_i - r_i)$
    Reality announces r_i

The game is about Stakeholder's desire of having a software C. He asks Designer to implement C, which has a cost of $M\$$. However, she does not have enough money to do this. So she has to borrow money from either Stakeholder or National Bank with the return of interest (ROI) p or r, respectively.

Stakeholder starts the game by announcing p which is his belief about the minimum ROI for investing $M\$$ on C. In other words, he claims that r would be greater than p. If M equals 1, p is the minimum amount of money one can receive for 1\$ of investment. Stakeholder shows his belief on p by a commitment that he is willing to buy C for price $(1+p)M$ if the designer does not believe him and borrows money from someone else.

If Designer believes Stakeholder, she will borrow M from Stakeholder. Later on, she can sell C to him for $M(1+r)$ and returns $M(1+p)$ to him. So, the final amount of money Designer can earn from playing the game is $M(r-p)$.

If Designer does not believe Stakeholder, she will borrow money from National Bank, and has to return $M(1+r)$. Then, Stakeholder is willing to buy C with $M(1+p)$. In this case, Designer can earn $M(p-r)$.

Suppose that Designer has an initial capital of K_0. After round i-*th* of the game, she can accumulate either $K_i = K_{i-1} + M(r-p)$ or $K_i = K_{i-1} + M(p-r)$, depend on whether she believes Stakeholder or not. Designer has a winning strategy if she can select the values under her control (the $M\$$) so that she always keeps her capital never decreasing, intuitively, $K_i >= K_{i-1}$ for all rounds.

The law of large numbers here corresponds to say that if unlikely events happen then Designer has a strategy to multiply her capital by a large amount. In other words, if Stakeholder estimates Reality correctly then Designer have a strategy to budget for costs that will not run over budget.

4 Making Decision: what are the best things to implement

One of the main objectives of modeling evolution is to provide a metric (or set of metrics) to indicate how well a system design can adapt with evolution. Together with other assessment metrics, designers have clues to decide what an "optimal" solution for a system-to-be is.

The major concern in assessment evolution is answering the question: "Whether or not a model element (or set of element) becomes useless after evolution?". Since the occurrence of evolution is uncertain, so the usefulness of an element set is evaluated in term of probability whose interpretation is discussed in § 3.2.

In this sense, this work proposes two metrics measuring the certainty of an element sets as follows.

Max Belief (MaxB): of an element set X is a function that measures the maximum belief supported by Stakeholder such that X is useful to a set of top requirements after evolution happens. This belief of usefulness for a set of model element is inspired from a game in which Stakeholder play a game together with Designer and Reality to decide which elements are going to implementation phase.

Residual Risk (RRisk): of an element set X is the complement of total belief supported by Stakeholder such that X is useful to set of top requirements after evolution happens. In other words, residual risk of X is the total belief that X is not useful to set of top requirements regard to evolution. Importantly, do not confuse this notion of residual risk with the one in risk analysis studies which are different in nature.

Given an evolutionary requirement $RM = \langle RM, \mathcal{R}_o, \mathcal{R}_c \rangle$ where $\mathcal{R}_o = \bigcup_i \left\{ RM \xrightarrow{p_i} RM_i \right\}$ is a one-element set of observable evolution rules, and $R_c = \bigcup_{ij} \left\{ RM_i \xrightarrow{*} RM_{ij} \right\}$ is a set controllable evolution rules applying to potential evolutions of r_o, the calculation of max belief and residual risk is illustrated in Eq. 1, Eq. 2 as follow.

$$MaxB(X) = \max_{RM \xrightarrow{p_i} RM_i \in \mathcal{S}} p_i \tag{1}$$

$$RRisk(X) = 1 - \sum_{RM \xrightarrow{p_i} RM_i \in \mathcal{S}} p_i \tag{2}$$

where $\mathcal{S}$ is set of potential evolutions in which X is useful.

$$\mathcal{S} = \left\{ RM \xrightarrow{p_i} RM_i | \exists (RM_i \xrightarrow{*} RM_{ij}) \in \mathcal{R}_c \ st. \mathsf{useful}(X, RM_{ij}) \right\}$$

One may argue about the rationale of these two metrics. Because he (or she) can intuitively measure the usefulness of an element set by calculating the *Total Belief* which is exactly the complement of our proposed *Residual Risk*. However, using only Total Belief (or *Residual Risk*) may mislead designers in case of a *long-tail* problem.

The long-tail problem, firstly coined by Anderson [1], describes a larger population rests within the tail of a normal distribution than observed. A long-tail example depicted in Fig. 2 where a requirement model RM might evolve to several potential evolutions with very low probabilities (say, eleven potential evolutions with 5% each), and another extra potential evolution with dominating probability (say, the twelfth one with 45%). Suppose that an element A appears in first eleven potential evolutions, and an element B appears in the last twelfth potential evolution. Apparently, A is better than B due to A's total belief is 55% which is greater than that of B, say 45%. However, at the end of the day, only

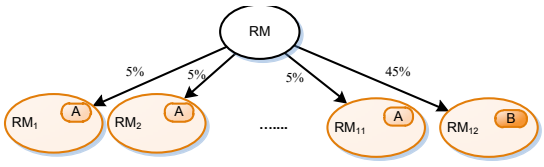


Fig. 2. The long-tail problem.

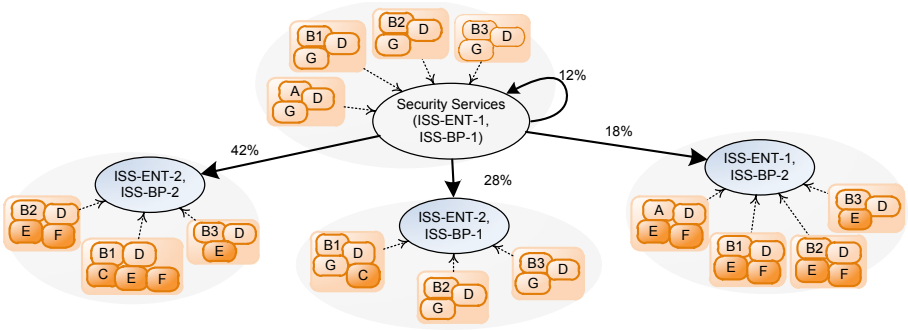


Fig. 3. Evolution of the SWIM Security Service.

one potential evolution becomes effective (i.e., is chosen by Reality) rather than 'several' potential evolutions are together chosen. If we thus consider every single potential evolution to be chosen, the twelfth one (45%) seems to be the most promising and *Max Belief* makes sense here. Arguing that A is better than B or versa is still highly debatable. Ones might put their support on the long tail [1], and ones do the other way round [5]. Therefore, we introduce both *Residual Risk* and *Max Belief* to avoid any misleading in the decision making process that can be caused when using only Total Belief.

For a better understanding of *Max Belief* and *Residual Risk*, we conclude this section by applying our proposed metrics on the evolution of SWIM Security Services discussed in previous section. In Fig. 3, here we have an initial requirement model RM0(ISS-ENT-1,ISS-BP-1) that will evolve to RM1(ISS-ENT-2,ISS-BP-1), RM2(ISS-ENT-1,ISS-BP-2), and RM3(ISS-ENT-2,ISS-BP-2) with probabilities of 28%, 18% and 42%, respectively. There are 12% that RM0 stays unchanged. Each requirement model is represented as a bubble in which there is a controllable rule with several design alternatives. Each design alternative is an element set represented as a rounded rectangle that contains elements (such as A, D, and G) to support (fulfill) requirements of that requirement model.

Table 3 shows some examples, where the first column displays element sets, and the two next columns show the values of max belief and residual risk. Notice that the max belief and residual risk in the first row, where the element set is $\{A, D\}$, are n/a which means that we are unable to find any potential evolution that $\{A, D\}$ can support all top requirements.

Table 3. Examples of Max Belief and Residual Risk

Element set	Max Belief	Residual Risk
{A, D,}	n/a	n/a
{A, E, D, G, F}	18%	70%
{B3, D, G}	28%	60%
{B1, D, G, C}	28%	60%
{B3, D, E, G}	42%	0%
{B2, D, E, F, G}	42%	0%

Look at the table, $\{B3, D, E, G\}$ and $\{B2, D, E, F, G\}$ seem to be the best choices, since they have a high max belief (42%) and low residual risk (0%). The zero residual risk means these element sets are surely still useful after evolution. If the cost of implementation is the second criteria, $\{B3, D, E, G\}$ seems to be better if the cost for each element is equal.

5 Handling complex evolution

If the model is too large and complex, instead of dealing with the evolution of the whole model, we can consider evolution in each subpart. If a subpart is still too large and complex, we can recursively divide it into smaller ones until we are able to deal with. Now, suppose that we have a big model divided into several pieces with evolution rules. We then need to combine these local rules together to produce the global evolution ones for the whole model. For simplicity, we assume that:

ASS-1: Independence of evolutions All observable rules are independent. It means that they do not influence each other. In other words, the probability that an evolution rule is applied does not affect to that of other rules.

ASS-2: Order of evolutions Controllable evolutions are only considered after observable evolutions.

As discussed, observable rules are analyzed on independent subparts. Prevailing paradigms of software development (*e.g.,* Object-Oriented, Service-Oriented) encourage encapsulation and loosely coupling. Evolutions applying to subparts, therefore, are often independent. Nevertheless, if there are two evolution rules which influent each other, we can combine them into a single evolution. We assume that dependent evolutions do happen, but not a common case. Hence manual combination of these rules is still doable.

The second assumption is the way we deal with controllable rules. If we apply controllable rules before observable ones, it means we look at design alternatives before observable evolutions happen. This makes the problem more complex since under the effect of evolution, some design alternatives are no longer valid, and some others new are introduced. Here, for simplicity, we look at design alternatives for evolved requirement models that will be stable at the end of their evolution process.

After all local evolutions at subparts are identified, we then combine these rules to a global evolution rule that applies to the whole model. The rationale of this combination is the effort to reuse the notion of Max Belief and Residual Risk (§4) without any extra treatment. In the following we discuss how to combine two independent observable evolution rules.
Given two observable rules:

$$r_{o1} = \bigcup_{i=1}^{n} \left\{ RM1 \xrightarrow{p1_i} RM1_i \right\} \text{ and } r_{o2} = \bigcup_{j=1}^{m} \left\{ RM2 \xrightarrow{p2_j} RM2_j \right\}$$

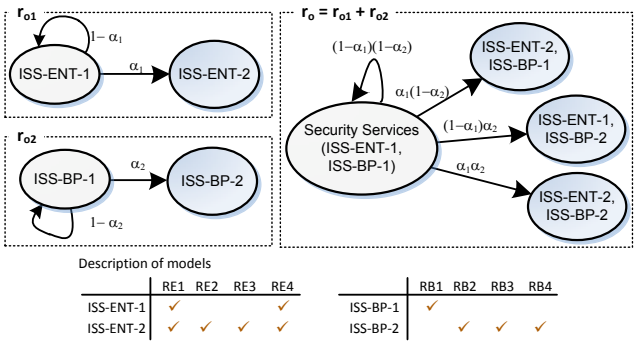


Fig. 4. Example of combining two observable evolution rules.

Let r_o is combined rule of r_{o1} and r_{o2}, we have:

$$r_o = \bigcup_{\substack{1 \le i \le n \\ 1 \le j \le m}} \left\{ RM1 \cup RM2 \xrightarrow{p1_i * p2_j} RM1_i \cup RM2_j \right\}$$

Fig. 4 illustrates an example of combining two observable rules into a single one. In this example, there are two subparts of SWIM Security Service: ISS-ENT and ISS-BP. The left hand side of the figure displays two rules for these parts, and in the right hand side, it is the combined rule.

In the general case we have multiple steps of evolution *i.e.* evolution happens for many times. For the ease of reading, *step 0* will be the first step of evolution, where no evolution is applied. We use RM_i^d to denote the i-*th* model in step d, and $r_{od,i}$ to denote the observable evolution rule that applies to a model RM_i^d , *i.e.* $r_{od,i}$ takes RM_i^d as its original model.

The multi-step evolution begins with an original model RM_1^0. This model can evolve to one of the potential evolutions RM_i^1. In the second step, each RM_i^1 then also evolves to one of many potential evolutions RM_j^2. The evolution stop after k steps of evolution. If we represent a model as a node, and connect a model to its potential evolutions as we have done as aforementioned, then we have a tree-like graph, called *evolution tree* with k-depth.

Fig. 5 illustrates a two-step evolution, in which observable rules are denoted as dotted boxes. The original model lays on top part of a box, and all potential evolutions are in sub boxes laid at the bottom. There are directed edges connecting the original model to potential evolutions. The label on each edge represents the probability such that original model evolves to the target model.

In Fig. 5, an initial requirement model RM_1^0 can evolve to either RM_1^1, RM_2^1 or RM_3^1. Likewise, RM_i^1 evolves to RM_j^2, where i=1..3 and j=1..9. Here, we have a ternary complete tree of depth 2. Generally, the evolution tree of a k-step consecutive evolution is a complete k-depth, m-ary tree.

We can always collapse a k-step evolution into an equivalent 1-step one in terms of probability by letting the original model evolve directly to the very

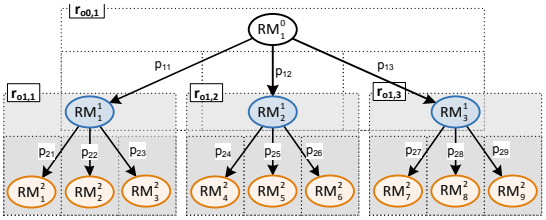


Fig. 5. Multiple steps (phases) evolving requirement model.

last alternatives with the probabilities that are multiplication of probabilities of intermediate steps.

Therefore, any k-step evolution has an equivalent 1-step evolution. Hence all analyses discussed in §4 are applicable without any modification.

6 Related Work and Conclusion

A majority of approaches to software evolution has focused on the evolution of architecture and source code level. However, in recent years, changes at the requirement level have been identified as one of the drivers of software evolution [4, 12, 31]. As a way to understand how requirements evolve, research in PROTEUS [24] classifies changing requirements (that of Harker et al [11]) into five types, which are related to the development environment, stakeholder, development processes, requirement understanding and requirement relation. Later, Lam and Loomes [15] presents the EVE framework for characterizing changes, but without providing specifics on the problem beyond a meta model.

Several approaches have been proposed for supporting requirements evolution. Zowgi and Offen [31] work at meta level logic to capture intuitive aspects of managing changes to requirement models. Their approach involves modeling requirement models as theories and reasoning changes by mapping changes between models. However, this approach has a limitation of overhead in encoding requirement models into logic.

Russo et al.'s [26] propose an analysis and revision approach to restructure requirements to detect inconsistency and manage changes. The main idea is to allow evolutionary changes to occur first and then verify their impact on requirement satisfaction in the next step. Also based on this idea, Garcez et al [4] aim at preserving goals and requirements during evolution. In the analysis, a specification is checked if it satisfies a given requirement. If it does not, a diagnosis information is generated to guide the modification of specification in order to satisfy the requirement. In the revision, the specification is changed according to diagnosis information generated. Similar to Garcez et al, Ghose's [9] framework is based on formal default reasoning and belief revision, aiming to address the problem of inconsistencies due to requirement evolution. This approach is supported by automated tools [10]. Also relating to inconsistencies, Fabrinni et al.'s [6] deals with requirement evolution expressed in natural language, which is challenging to capture precisely requirement changes. Their approach employs

formal concept analysis to enable a systematic and precise verification of consistency among different stages, hence, control requirement evolution.

Other notable approaches include Brier et al.'s [3] to capturing, analyzing, and understanding how software systems adapt to changing requirements in an organizational context; Felici et al [8] concern with the nature of requirements evolving in the early phase of the system; Stark et al [29] study the information on how change occurs in the software system and attempts to produce a prediction model of changes; Lormans et al [21] use a formal requirement management systems to motivate a more structural approach to requirement evolution.

In this work, we have presented a way to represent evolutions on requirement models in terms of evolution rules. We used probability to express the uncertainty of evolutions. We discussed why we cannot simply use Bayesian interpretation of probability here. Instead, we employed game-theoretic approach to give explanation for the semantic behind probabilities.

Furthermore, we introduced two notions of max belief and residual risk, which help to reason on evolutionary models. The analysis outcome, together with other criteria (*e.g.,* cost, risk analysis) provide a clue to decision makers to decide the optimal solution, which is more evolution-resilient for the system-to-be.

We based our work on a concrete case study of the System Wide Information Management for air traffic control. The examples not only help to explain better our idea, but also show the applicability of our approach, though it still needs more validation in future.

As a part of future work, we plan to instantiate our approach in specific modeling language such as goal-based language and validate the empirical applicability of goal-models to the SWIM case study.

References

1. C. Anderson. The long tail. *Wired*, October 2004.
2. A. Anton and C. Potts. Functional paleontology: The evolution of user-visible system services. *TSE*, 29(2):151–166, 2003.
3. J. Brier, L. Rapanotti, and J. Hall. Problem-based analysis of organisational change: a real-world example. In *Proc. of IWAAPF '06*. ACM, 2006.
4. A. d'Avila Garcez, A. Russo, B. Nuseibeh, and J. Kramer. Combining abductive reasoning and inductive learning to evolve requirements specifications. In *IEE Proceedings - Software*, volume 150(1), pages 25–38, 2003.
5. A. Elberse. Should you invest in the long tail? *Harvard Business Review*,Jul–Agu 2008.
6. F. Fabbrini, M. Fusani, S. Gnesi, and G. Lami. Controlling requirements evolution: a formal concept analysis-based approach. *ICSEA '07*, 2007.
7. FAA. System wide information management (swim) segment 2 technical review. Tech. report, 2009.
8. M. Felici. *Observational Models of Requirements Evolution*. PhD thesis, 2004.
9. A. Ghose. A formal basis for consistency, evolution and rationale management in requirements engineering. *ICTAI '99*, 1999.
10. A. Ghose. Formal tools for managing inconsistency and change in re. In *IWSSD '00*, Washington, DC, USA, 2000. IEEE Computer Society.

11. S. Harker, K. Eason, and J. Dobson. The change and evolution of requirements as a challenge to the practice of software engineering. In *RE '01*, 1993.
12. J. Hassine, J. Rilling, J. Hewitt, and R. Dssouli. Change impact analysis for requirement evolution using use case maps. In *IWPSE '05*, 2005.
13. M. Jackson. *Problem Frames: Analysing & Structuring Software Development Problems*. Addison-Wesley, 2001.
14. C. F. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *TSE*, 25(4):493–509, 1999.
15. W. Lam and M. Loomes. Requirements evolution in the midst of environmental change: a managed approach. In *CSMR '98*, 1998.
16. M. LaMantia, Y. Cai, A. MacCormack, and J. Rusnak. Analyzing the evolution of large-scale software systems using design structure matrices and design rule theory: Two exploratory cases. In *Proc. of WICSA '08*, pages 83–92, 2008.
17. M. Lehman. On understanding laws, evolution and conservation in the large program life cycle. *J. of Sys. and Soft.*, 1(3):213 –221, 1980.
18. M. Lehman. Programs, life cycles, and laws of software evolution. *Proc. IEEE 68*, 9:1060 –1076, September 1980.
19. L. Lin, S. Prowell, and J. Poore. The impact of requirements changes on specifications and state machines. *SPE*, 39(6):573–610, 2009.
20. L. Liu and E. Eric Yu. Designing information systems in social context: A goal and scenario modelling approach. *Info. Syst*, 29:187–203, 2003.
21. M. Lormans, H. van Dijk, A. van Deursen, E. Nocker, and A. de Zeeuw. Managing evolving requirements in an outsourcing context: an industrial experience report. In *IWPSE '04*, pages 149 –158, 2004.
22. T. Mens, J. Ramil, and M. Godfrey. Analyzing the evolution of large-scale software. *J. of Soft. Maintenance and Evolution: Research and Practice*, 16(6):363–365,2004.
23. Program SWIM-SUIT. D1.5.1 overall swim users requirements. Tech. report, 2008.
24. Project PROTEUS. Deliverable 1.3: Meeting the challenge of chainging requirements. Tech. report, Centre for Soft. Reliab., Univ. of Newcastle upon Tyne, 1996.
25. R. Ravichandar, J. Arthur, S. Bohner, and D. Tegarden. Improving change tolerance through capabilities-based design: an empirical analysis. *J. of Soft. Maintenance and Evolution: Research and Practice*, 20(2):135–170, 2008.
26. A. Russo, B. Nuseibeh, and J. Kramer. Restructuring requirements specifications. In *IEE Proceedings: Software*, volume 146, pages 44 – 53, 1999.
27. G. Shafer, V. Vovk, and R. Chychyla. How to base probability theory on perfect-information games. *BEATCS*, 100:115 – 148, February 2010.
28. P. Soffer. Scope analysis: identifying the impact of changes in business process models. *J. of Soft. Process: Improvement and Practice*, 10(4):393–402, 2005.
29. G. Stark, A. Skillicorn, and R. Ameele. An examination of the effects of requirements changes on software releases. *Crosstalk: Journal of Defence Software Engineering*, pages 11–16, December 1998.
30. P. Zave and M. Jackson. Four dark corners of req. eng. *TSEM*, 6(1):1–30, 1997.
31. D. Zowghi and R. Offen. A logical framework for modeling and reasoning about the evolution of requirements. *ICRE '97*, 1997.

APPENDIX C. Change- Driven Model Transformations: Change (in) the Rule to Rule the Change

Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Change-Driven Model Transformations

Change (in) the Rule to Rule the Change

Gábor Bergmann[1], István Ráth[1], Gergely Varró[2], Dániel Varró[1]

[1] Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
H-1117 Magyar tudósok krt. 2, Budapest, Hungary
[2] Technische Universität Darmstadt,
Real-Time Systems Lab,
DE-64283 Merckstr. 25, Darmstadt, Germany

Received: date / Revised version: date

Abstract In the paper, we investigate *change-driven model transformations*, a novel class of transformations, which are directly triggered by complex model changes carried out by arbitrary transactions on the model (e.g. editing operation, transformation, etc). After a classification of relevant change scenarios, we identify challenges for change-driven transformations. As the main technical contribution of the current paper, we define an expressive, high-level language for specifying change-driven transformations as an extension of graph patterns and graph transformation rules. This language generalizes previous results on live model transformations by offering trigger events for arbitrarily complex model changes, and dedicated reactions for specific kinds of changes, making this way the concept of change to be a first-class citizen of the transformation language. We discuss how the underlying transformation engine needs to be adapted in order to use the same language uniformly for different change scenarios. The technicalities of our approach will be discussed on a (1) model synchronization case study with non-materialized target models and (2) a case study on detecting the violation of evolutionary (temporal) constraints in the security requirements engineering domain.

Key words incremental model transformation – change models – change-driven transformations

Acknowledgements We would like to thank the anonymous reviewers for their insightful advice. This work was partially supported by EU projects SENSORIA (IST-3-016004) and SecureChange (ICT-FET-231101). The third author was partially supported by the Postdoctoral Research Fellowship of the Alexander von Humboldt Foundation.

1 Introduction

The changes of models are a central issue in model-driven software engineering (MDSE). For instance, software engineers continuously change and improve their design models using manual refinement steps or semi-automated model refactoring transformations. However, the change of one model may easily introduce inconsistencies with other models developed by other designers. In case of model-based simulation, the simulator needs to efficiently detect the changes of the underlying model (e.g. to highlight steps which can be executed at this stage).

Unfortunately, the actual notion and representation of change can be very different in practical tools and scenarios. While modern domain-specific modeling environments offer an explicit set of operations, which can be executed on the model by the user in graphical editors, in many other cases, the engineers have no control over the many ways a model may change: any kind of model manipulations are possible in any sequence.

In modern modeling environments (like the Eclipse Modeling Framework, EMF), elementary model changes are reported on-the-fly by some *live notification* mechanisms to support undo/redo operations. Model versioning frameworks persist the *change history* of models (history-aware log of model changes, which records causal dependency / timeliness between such changes) in the form of an external *change document*. Many complex refactoring operations frequently require the user to preview the requested changes (expressed as *change commands*) prior to actually executing them to the model. A common concept for all these cases is the *change delta*, which uniformly captures (the aggregated effect of) a transaction which caused the model to evolve from a previous *pre-state* to a *post-state*.

Furthermore, in certain complex tool integration scenarios, we receive absolutely no information about changes, i.e., changes just happen without any trace (notification or change log). Other scenarios include those with observable (but not controllable) changes.

Model transformations, which consume (as input) or produce (as output) change models, in addition to the underlying models themselves, are called *change-driven transformations (CDT)* [1]. The practically relevant change scenarios impose additional challenges for CDTs. First, change models are typically restricted to contain elementary model changes (i.e. the creation and deletion of certain model elements); change-driven transformations, however, are frequently triggered by *complex (aggregated) model changes* carried out by a transaction on the model (e.g. user edit, refactoring transformation, etc), which is a key challenge. Moreover, some models can be *non-materialized*, i.e. when only an external interface is available for query and manipulation, which traditionally requires the development of complex adapters in the modeling environment. In addition, *traceability information* can also be limited and externalized (i.e. cannot be stored in the host models), which imposes further challenges. Finally, well-formedness restrictions over a trajectory of changing models can express *evolutionary constraints*.

Despite the large variety of existing model transformation languages and tools, the concept of change is not a first-class citizen in them. In the paper, we argue that in many application scenarios, *making the change as part of transformations rules* is a promising approach. As a result, appropriate reactions can be based upon observing the current snaphot of the model and also *the way how the model evolved*.

The execution mechanism of change-driven transformations also differs from traditional batch transformations. First, uncontrolled model changes (which are reported by a model management framework or detected by the CDT engine itself) happen in a transaction. Then based upon the observed changes, certain change-driven transformation rules may be triggered to manipulate the models in a controlled way. Naturally, model changes induced by these rules are transactions themselves, which may thus trigger further reactions.

In this paper, (1) we propose a high-level *change-driven transformation language* which allows to unambiguously and succinctly capture the nature of change (as integral part of the language) and to specify the appropriate reactions. (2) Furthermore, *implementation techniques of CDTs* will be discussed which allow to use a single CDT language uniformly and independently from the actual change scenario and change representation including (A) internal representations (notifications of the modeling environment) vs. externalized change models, (B) forward vs. backward deltas. The concrete syntax of the language is introduced as an extension to the transformation language of the VIATRA2 framework, and it generalizes previous results on live transformations [2,3].

The technicalities of change-driven transformations will be presented using two motivating case studies. First, in *a model synchronization scenario* incremental synchronization is carried out on a non-materialized target model (i.e. when only target object identifiers and a model manipulation interface of the target model are known, and the rest of the target model does not exist as an in-memory model within the transformation framework) with weak traceability links. Then, *a case study from the security requirements engineering domain* will demonstrate how evolutionary (temporal) requirements can be specified and incrementally evaluated using change-driven transformations.

The rest of the paper is structured as follows. Section 2 provides a categorization of changes to be addressed in the paper, and details the main challenges of change-driven transformations. In Section 3, a motivating model synchronization case study is introduced as a running example for our paper. Section 4 describes the novel language for change-driven transformations. Section 5 outlines a prototype system architecture for the various cases of change scenarios. Section 6 exemplifies the use of our CDT language in the model synchronization scenario. Specifying and checking evolutionary constraints will be the goal of another case study presented in 7. Section 8 provides a detailed discussion on the advantages and limitations of our approach. Finally, Section 9 summarizes related work and Section 10 concludes our paper.

2 Overview of the approach

Changes are inherent to modeling. In model-driven engineering, models are rarely static, in fact, they are evolving continuously. Most of this evolution is driven by user input in modeling environments and editors. In other cases, changes are automatically introduced by batch model manipulations such as model import, transformation and export.

Change is considered to be the transition of a model from a *pre-state*, to a *post-state*, and the difference between the two is called the *change delta* (or *model delta*). This terminology is independent of the granularity and the abstraction level; it applies for changes that are just elementary model manipulation operations as well as for batch transactions or even for complex business decisions.

A model-driven design setup requires these changes to be propagated along a chain of tools into derived models or generated source code. The workflow may also involve the merging of models, back-annotating the results of an analysis performed on a transformation's target model to the source model, or identifying interesting or erroneous parts within models. Thus there is a need for capturing the changes precisely.

In this paper, we propose a novel model transformation technology designed to address this problem by operating on *changes of models* as first-class citizens. We first propose (in Section 2.1) a classification scheme for changes that we aim to handle uniformly with change-driven transformations. We introduce a taxonomy that will be useful to describe which cases our change-driven transformation approach aims to deal with, and what its advantages are. Section 2.2 explains the challenges of change-driven transformations, and Section 2.3 outlines how the rest of the paper will address them.

2.1 Aspects of change

We define four perspectives (control, observability, information source, delta representation), distinguishing several different ways to perceive changes to a model. An overview is shown in Fig. 1.

2.1.1 The controllability perspective There are scenarios where changes are *controllable*, meaning only an explicitly defined set of changes is permitted at each state of the model. A common example is when models are required to be edited exclusively using dedicated editors that only allow a limited set of high-level domain-specific model manipulations. Such modeling languages are often described by generating graph grammars [4, 5], where the grammar rules coincide with the editing rules.

However, in a wider range of scenarios, the transformation designer has no control over the possible ways a model may change during its lifecycle. It can happen through manual editing in a visual tool, batch refactoring, model transformation, model merging, etc. Any type of model manipulation is possible: creation/deletion of entities and relations of arbitrary type, modifying attribute values or element names, in any arbitrary sequence, in unforeseeable ways. Furthermore, it is even possible that models temporarily violate certain domain-specific well-formedness constraints during the changes. In this case, we need to handle *non-controllable* changes.

2.1.2 The observability perspective After the transformation iscompleted and any derived model(s) are created, it is possible that the target models are changed without any model management support (e.g. when the generated source code is changed in model-to-text scenarios). When the transformation is invoked next time, it can only access the current updated version (post-state), without having any additional information sources revealing how the models were changed since the last transformation execution. In this case, the change is *invisible*.

However, with support from a model management environment, there may be ways to trace the changes made to a model, such as change logs. When the transformation system has to determine the appropriate reactions to execute, it can take advantage of such information sources. We consider the change *observable* if it can be deduced what the pre-state was, what change delta has been applied to it, and what the resulting post-state is.

2.1.3 The source of information perspective If the change is observable, further distinction is possible based on what kinds of information sources are available. As previously mentioned, a change consists of a pre-state, a post-state and a change delta between them. The change is observable if and only if at least two of these three information sources are directly available, since the third one can be derived. Although this derivation is possible, it might not always be efficient in an actual implementation. Therefore we distinguish three scenarios based on which two of these three information sources are available, acknowledging that each scenario offers a different kind of support for implementing change propagating transformations. A similar categorization is presented in [6].

Some model management systems may preserve a previous version of the model from the last execution of the transformation, in addition to the current version. This can be the case if version control is enabled in the model repository. When the pre-state and the post-state are directly observable, we call it the *snapshot scenario* (state-based in the terminology of [6]).

In other situations, a description of the change may be available before it was applied on the model. An example of such a situation would be applying a patch onto the model, that consists of changes performed on a remote copy of the model. This is also the case when change requests have to be analyzed in a change management system, before the changes are actually carried out. If the pre-state and the delta are directly available, we call it the *command scenario* (forward delta in the terminology of [6]), and the delta can also be called a *change command*.

Finally in the *history scenario* (called backward delta in [6]), the post-state is directly available along with the delta (which can be called a *change history*). A typical example would be manually editing a model in an editor environment, which produces notifications of the editing operations after they have been carried out, or saves transaction logs (e.g. redo stack) together with the updated version of the model.

It is a rare but possible case that all three information sources are directly available (this is called the change-based case in [6]). For example, an editor may save change logs, while the model repository captures the pre-state and the post-state as well. In this case any of the implementation strategies proposed for the above three scenarios is applicable, and the choice can be made on the basis of efficiency.

		pre-state	change delta	post-state	Documented	Live
Observable	History (backward	?	+	+	redo stack, transaction log	continuous notification after modifications (e.g EMF)
	Command (forward delta)	+	+	?	change request, patch	continuous pre-submit / pre-commit monitoring of changes
	Snapshot (state-based)	+	?	+	version control of models with "black box" modifications in-between	
	Invisible	?	?	+	arbitrary modifications between transformation runs	

Fig. 1 Change scenarios (ignoring controllability)

2.1.4 The delta representation perspective In the history and command scenarios, the change delta is available as an information source. In this case, we define a fourth perspective that indicates how the change delta is perceived by the model transformation environment.

In the *documented change scenario*, the delta is available as a data structure called the *delta document*, that specifies exactly how the pre-state and the post-state differs. One example (history scenario) is a model editor maintaining a redo log during editing, that may be retained when the model is saved. The previously mentioned change management system with change requests can be thought of an example in the command scenario.

In the *live change scenario*, the change is experienced on-the-fly, as it happens by continuously receiving run-time notifications on the change. The notifications (e.g. method calls) can be issued before or after the actual change (command or history). The notification granularity (frequency) can range from the level of elementary model manipulations to aggregated effects of longer transactions, smoothly transitioning into the documented case. A live scenario frequently happens in model editing environments and centralized model management solutions. As a great advantage of this scenario, changes to a source model can be on-the-fly reflected in the target model, and other kinds of live transformation can be performed efficiently, facilitating valuable feedback [3,7].

2.2 Transformations of change

Change driven model transformations are model transformations which consume changes of the host model M as input (see Fig. 2), and turn these changes into model manipulation operations, native operations (such as asynchronous messages, or external API calls), or traceability records for persistent storage of changes.

Essentially, a change driven transformation rule is *enabled by some changes in the host model*. The actual change representation can be of different nature (in accordance with Fig. 1), e.g. a sequence of model manipulation operations or a change delta.

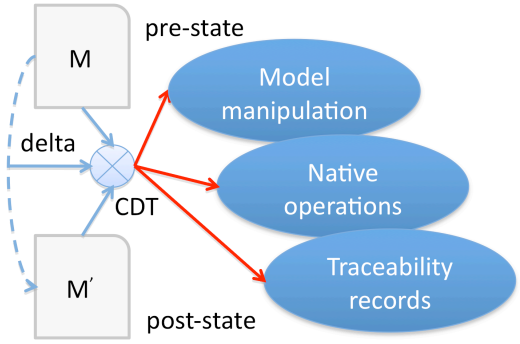


Fig. 2 Change-driven transformations

2.2.1 Challenges for change-driven transformations This interpretation of change-driven transformations needs to be refined in many practical application scenarios with different model handling characteristics, which are discussed in the following.

– **Unified handling of complex changes in all change processing scenarios**. Analogously to high-level formalisms of model and graph transformations, change-driven transformations should support a declarative, high-level specification of changes that can be seamlessly integrated into a "host" model or graph transformation language. Moreover, this formalism (and the underlying execution semantics) should support a uniform specification and execution model for all change processing scenarios discussed previously, in order to relieve the transformation developer from a significant amount of manual coding (notification, adapters etc.), especially in the case of non-controllable changes. As an additional benefit, this independence will make a transformation portable across different change scenarios without modifying its code.
The language can then be used as (i) a complete stand-alone formalism for handling model transformation scenarios such as incremental model synchronization, model simulation (animation) in discrete systems, and on-the-fly well-formedness constraint evaluation. Additionally, (ii) it is also useful as an in-

termediate formalism bridging the gap between the technical challanges of the different change scenarios and high-level languages tailored for certain uses of model transformations (e.g. QVT Relations for model synchronization, or other GT-based languages for behavioral simulation).

– **Ability to handle traditional model transformation scenarios.**
Ideally, the change-driven rule formalism should support traditional execution semantics as well (based on an empty pre-state), so that the rules can be used without additional changes e.g. to perform the "first" transformation phase in model synchronization scenarios.
– **Handling both materialized and non-materialized models**.
A typical assumption of most model transformation approaches is that the host model M is available as a materialized model in a common model store (e.g. as in-memory EMF models inside the MT framework). However, in some model transformation scenarios, this may not be technically feasible (e.g. for performance reasons – the model may be too large to fit in memory, or not trivial to import and convert). Practically, this means that only an external interface of its native environment is available for querying and manipulating M, but still using some model transformation approach is desirable to incrementally synchronize the model (e.g. for maintaining consistent views).
– **Traceability models** are used universally in many MT scenarios for correspondence mapping and also to preserve some information on the execution state of the transformation itself. This information (along with negative application conditions) is mostly used to help the specification of incremental rules that only operate on changed parts of the model (e.g. incremental change propagation in model synchronization). In controllable, non-controllable and invisible change processing scenarios, our change-driven tranformation technology will automatically maintain a cache containing the (historical) information about the pre-state. As a result, traceability models (as well as rule preconditions) can be simplified significantly: they are only used for correspondence mapping between source and target models, but not for storing the past. For instance, old values of attributes would no longer be required to be stored as part of the traceability model to support attribute changes, which is typically the case for existing transformation technology.
– **Checking properties over evolving models** can also be a specification challenge for change-driven transformations. Here certain constraints can be evolutionary in the sense that they need to be evaluated over a sequence of model evolution steps and not over a single snapshot of the model. Traditional constraint languages (like OCL) can only handle these properties by encoding the trajectory as part of the models, which may blow up models significantly.

In addition to providing support for these traditional traceability use-cases, change-driven transformations also allow the changes themselves to be represented as models (attached to the host model on which they are evaluated). Moreover, the model-based representation should be completely equivalent to the in-memory representation of live changes so that both the "documented" and "live" change processing scenarios can be handled uniformly.

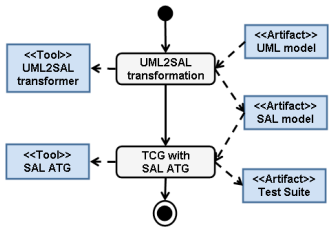
2.3 Contributions of the paper

As a summary, *change-driven transformations* take change information as inputs and produce change information as output. Taking this abstract view of CDTs, we first propose a language and execution semantics (Section 4) for capturing change-driven transformations in a uniform way. Afterwards, Section 5 shows an implementation architecture to support executing the same language in different change scenarios. Finally, we demonstrate (in Section 6) how change-driven transformations can automate a model synchronization problem in a tool integration context.

3 Case study: synchronization for deployed workflow models

Our motivating scenario is based on an actual tool integration environment developed for the SENSORIA and MOGENTES EU research projects. Here high-level workflow models (with control and data flow links, artefact management and role-based access control; the concrete syntax is illustrated in Fig. 3(a)) are used to define complex development processes which are executed automatically by the JBoss jBPM workflow engine in a distributed environment consisting of Eclipse client workstations and Rational Jazz tool servers. The process workflows are designed in a domain-specific language, which is automatically mapped to an annotated version of the jPDL execution language of the workflow engine. jPDL is an XML-based language (see Fig. 3(b) for the example that corresponds to Fig. 3(a)), which is converted to an XML-DOM representation once the process has been deployed to the workflow engine.

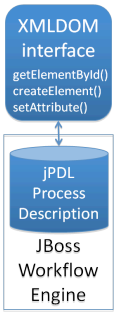
A major design goal was to allow the process designer to edit the process model and make changes without the need for re-deployment. To achieve this, we implemented an *asynchronous incremental code synchronizing model transformation*. This means that (i) while the user is editing the source process model, the changes are recorded. Then (ii) these changes can be mapped incrementally to the target jPDL XML model without regenerating it from scratch. Additionally, (iii) the changes



(a) Domain-specific workflow model

```
<node name="Interface Type Mapping">
  <action name="UML2SAL Transformation"
          class = "…">
    <toolId>UML2SAL transformer</toolId>
    <functionId>UML2SAL</functionId>
    <returnType>SALModel</returnType>
    <params>
      <entry>
        <key>umlModel</key><value>...<value>
      </entry>
    </params>
  </action>
  <transition from="initial_node"
              to="TCG-with-SAL-ATG" />
</node>
```

(b) JPDL XML Document



(c) jPDL interface

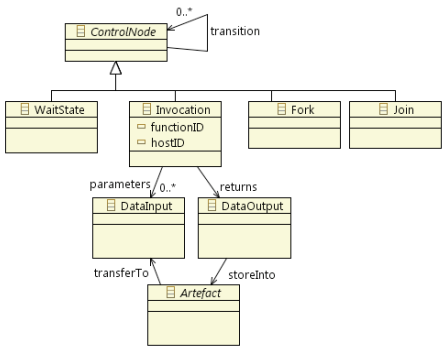
Fig. 3 Artefacts of the motivating scenario

can be applied directly on the deployed XML-DOM representation through jBPM's process manipulation DOM programming interface (Fig. 3(c)), but, (iv) in order to allow the changes to be applied to the remote workflow server, the actual XML-DOM manipulation is executed on a remote host asynchronously to the operations of the process designer.

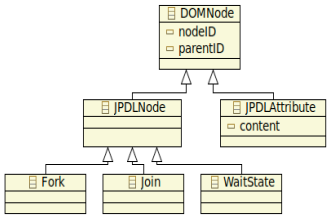
3.1 Metamodels of the case study example

The transformation scenario features the following models (Fig. 4):

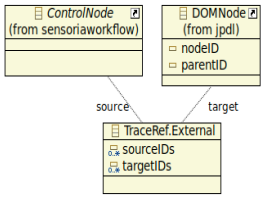
- *The domain-specific workflow language models* are materialized in the modeling environment of the editor, which is integrated with the VIATRA2 transformation engine. These models conform to the metamodel shown in Fig. 4(a). This language features workflows comprised of *Control nodes* of various types (such as *Invocation* which corresponds to an invocation of a (remote) tool service function, and *WaitState* which corresponds to a wait state of the execution process).
- *The jPDL models* are not stored in the transformation engine, but directly in the jBPM execution environment, in an XML-DOM-style format. However, to ease the understandability of further examples, we present the relevant fragment of the jPDL metamodel (Fig. 4(b)) in an ECore-like syntax. In this representation, the jPDL process graph is comprised of hierarchically embedded (through *parentID* references) *JPDLNodes* that can have various *JPDLAttributes* storing user-definable information (we use them to store domain-specific information contained in the source model, such as invocation parameters, tool service function names etc.).
- Finally, *traceability or correspondence models* are stored in the transformation engine, conforming to the simple metamodel shown in Fig. 4(c). These *external traceability* nodes store only "weak" links, which means that ID references point to domain-specific elements as well as to jPDL DOM elements. This way, external models can also be referenced.



(a) Domain-specific workflow metamodel



(b) jPDL DOM metamodel fragment



(c) External traceability metamodel

Fig. 4 Metamodels of the example

3.2 Change example

Fig. 5 shows a sample workflow instance model, a sequence of model manipulation steps, and the resulting modified state of the model. On the left side, two snapshots of the workflow model are shown, with the differences (the delta) highlighted (the jPDL target model is omitted for clarity).

The delta in-between could correspond e.g. to a GUI operation that inserts a new invocation between two neighbors in the control flow. If this workflow model had been transformed into a jPDL process before, the two models have to be re-synchronized after the change. The transformation has to react to this change, i.e. create a new JPDLNode that corresponds to Invocation $i3$, insert it into the sequence, and map the input and output parameters.

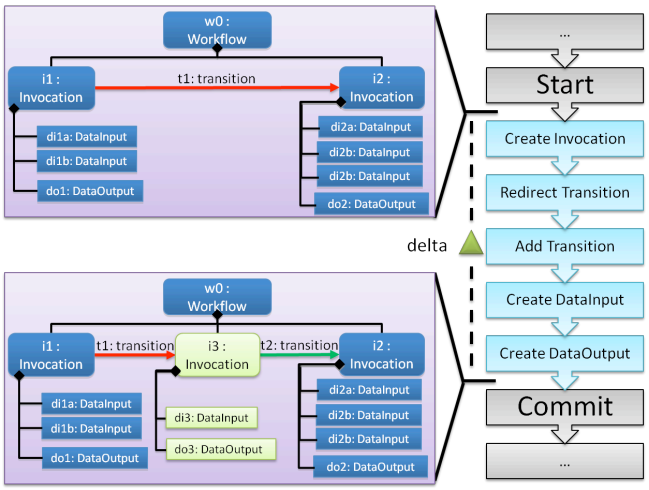


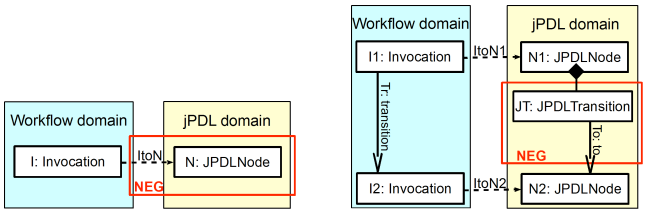
Fig. 5 Example model change

3.3 Constraints and Transformations with Graph Transformation

While model transformations may be implemented in any generic programming language, specialized model transformation tools offer declarative, rule-based transformation languages to support the easy and concise definition and efficient execution of transformations. Several tools use *Graph Transformation* (GT) [8] as theoretical underpinnings. The approach of graph transformation captures models as (typed, attributed) graphs. Parts of the graph can be described by declarative queries called *Graph Patterns* (GP) [9,10]. A pattern match is a conformant subgraph of the model; the match maps each pattern variable (nodes, edges) of the pattern to a concrete model element. The graph pattern matcher module is responsible for finding matches for patterns. A graph pattern may have a *Negative Application Condition* (NAC) to exclude certain cases. The model manipulation steps are defined by *Graph Transformation Rules* (GTR), using a GP called left-hand-side (LHS) to specify the condition when and where the rule is applicable, and another GP called right-hand-side (RHS) to declaratively indicate how the graph should be changed at the matches of the LHS (e.g. by specifying elements to be added, removed, or updated).

For readers unfamiliar with the basic concepts of graph transformation, Appendix A provides the definitions in the same way as they will be used in the paper, along with explanations and examples. The textual concrete syntax featured in listings extends the the GT parts of the transformation language [11] of the VIATRA2 framework.

Example. Figure 6(a) shows a graph pattern that matches invocation nodes in the workflow model that have not been mapped to any jPDL nodes yet. This mapping information is preserved by traceability links, represented graphically by dashed lines. The rectangle marked by `NEG` encloses a subpattern defining a negative application condition; the two involved domains are highlighted for clarity. Figure 6(b) reveals a more complex pattern, that finds a transition edge connecting two such invocations in the workflow model, that have already been mapped to jPDL nodes, but the corresponding jPDL transition element does not exist yet. Taking a closer look at these patterns, we can identify the variables and pattern constraints of both the positive patterns and the NACs; they are summarized by the table in Figure 6.



(a) Unmapped invocation (b) Unmapped transition

Variables	
I	$I1$, $N1$, $I2$, $N2$, $ItoN1$, $ItoN2$, Tr
Constraints	
I:Invocation	$I1$:Invocation $I2$:Invocation $N1$:JPDLNode $N2$:JPDLNode $ItoN1$:traceability, $I1$ to $N1$ $ItoN2$:traceability, $I2$ to $N2$ Tr: transition, $I1$ to $I2$
NAC Variables	
I, N, $ItoN$	$N1$, $N2$, JT, To
NAC Constraints	
N:JPDLNode $ItoN$:traceability, I to N	JT:JPDLTransition JT in $N1$ To:JPDLTransition.to, JT to $N2$

Fig. 6 Textual Representation of the Graph Pattern of Fig. 6(b)

```
pattern noJPDLTr(I1, N1, I2, N2) =
{
 Invocation(I1); // entity constraint
 traceability(ItoN1, I1, N1); // relation constraint
 JPDLNode(N1);
 neg pattern connected(N1, N2) = { // NAC definition
  JPDLTransition(JT) in N1;
  JPDLTransition.to(To, JT, N2);
 }
 JPDLNode(N2);
 traceability(ItoN2, I2, N2);
 Invocation(I2);
 Invocation.transition(Tr, I1, I2);
}
```

Listing 1 Example Graph Pattern

Listing 1 displays the same pattern as Figure 6(b). The bodies of patterns mainly consist of constraints expressed as predicates on variables. Entity constraints (e.g. type restrictions) are represented by unary predicates, while relation constraints (capturing the structural connectivity of the underlying graph model) are expressed by ternary predicates (edge variable, source,

target). The name of the predicate is the node or edge type name, respectively; the list of variables affected by the constraint follows in parentheses. The pattern and its NAC subpatterns have a set of interface variables that are visible (exported) from outside.

4 A Language for Change-driven Rules

4.1 Requirements and motivation for change-driven rules

For many transformation engineers, declarative, rule-based techniques offer an easy-to-understand way to specify model transformations. Consequently, we propose such a high-level change-driven rule formalism where transformation rules are augmented with a *guard*. The guard is evaluated in context of the changes that the graph model has undergone to determine whether the rule is an appropriate reaction to the change. In rule-based expert systems, this idea of *change* as a distinguished representation of information has been used for decades; for instance, in the well-known terminology of Event-Condition-Action (ECA) systems [12], our guards correspond to the notions of "triggering event" and the contextual condition of rules. As a complete adaptation of these techniques to model transformation technology (which is able to handle all relevant change processing scenarios using a unified, high-level formalism) does not yet exist to our best knowledge, we believe that such a language – architected as an *extension* to an existing graph transformation language – will serve practical applications well, in a number of application scenarios (e.g. model synchronization [1], on-the-fly constraint validation [3] and model animation [13]).

There are a number of requirements that such a language needs to fulfill:

- **reactivity** to be able to specify dynamic model changes as events that activate a rule
- **conciseness** to result in compact specifications for change-driven transformations
- **high-level specification** to be able to abstract from irrelevant details
- **intuitiveness** so that rules can be easily understood by those who are familiar with other model transformation languages
- **expressiveness** in order to be able to specify a large class of change-driven transformations using this language.

In this section, we define the concepts of change-driven transformations by proposing a language as an extension of the VIATRA2 transformation language. A quick overview of the language concepts is presented in Fig. 7, which will be gradually discussed in the section: Section 4.2 defines change patterns, while Section 4.3 specify change driven transformation rules on the foundations of change patterns.

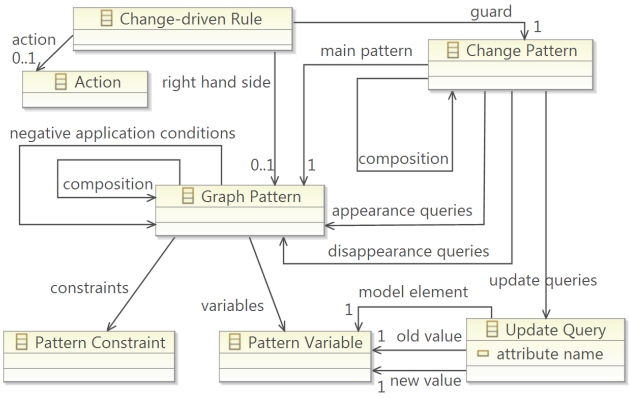


Fig. 7 Simplified metamodel of the proposed transformation language

Throughout the definitions, we heavily rely on some well-known concepts of Graph Transformation; for a more detailed introduction see Appendix A. These concepts include Graph Model G, Graph Pattern P, Graph Pattern with negative application conditions (NAC) $PN = \langle P, N^* \rangle$, attributed Graph Models and Graph Patterns, and finally Graph Pattern matching, with $G, m \models P$ meaning that m is a match for pattern P in graph G. Additionally, we also use the concept of Graph Transformation Rules consisting of a left-hand-side and a right-hand-side graph pattern (LHS and RHS), and the notion of applying the rule on a match of LHS, replacing it with an image of the RHS.

4.2 Change Patterns

We define high-level guards for change-driven rules in the form of Change Patterns. In addition to conventional graph patterns matched against the post-state, guards should also contain constructs for expressing the difference between the pre-state and post-state in the form of *change queries*. An *appearance query* indicates a graph pattern with a new match in the post-state, while the *disappearance query* indicates that a match of a given graph pattern is invalidated by the change. An *update query* captures that an attribute changes from an old value to a new one, i.e. it detects if an old value of an attribute disappeared, or a new value appeared.

The benefit of using graph patterns instead of elementary changes as appearance/disappearance queries is that a change pattern will match regardless of the order of elementary model manipulations that ultimately satisfied the appearance / disappearance / update queries. Thus it is irrelevant what the last operation was that e.g. completed the pattern of the appearance query. As a result, a single change pattern compactly captures a large set of different change sequences.

Definition 1 (Change Pattern) *Change Patterns (CP) can be defined as a tuple* $CP = \langle PN, P^*_+, P^*_-, U^*_{:=} \rangle$, *where*

- $PN = \langle P, N^* \rangle$ *is the* ***main graph pattern*** *with positive pattern* P *and negative application conditions* N^*.
- P^*_+ *is a set of graph patterns* $\{P_i = \langle V_i, C_i \rangle\}$ *called* ***appearance queries****. Each appearance query* P_i *with variables (pattern elements)* V_i *and their constraints* C_i *represents that a certain graph pattern appears due to the change.* P_i *is allowed to share variables with* P.
- P^*_- *is a set of graph patterns* $\{P_j = \langle V_j, C_j \rangle\}$ *called* ***disappearance queries****. Each disappearance query* P_j *with variables (pattern elements)* V_j *and their constraints* C_j *represents that a certain graph pattern disappears due to the change.* P_j *is allowed to share variables with* P.
- $U^*_{:=}$ *is a set of tuples* $\{U_h = \langle v_h^{Mod}, attr_h, v_h^{pre}, v_h^{post} \rangle\}$ *called* ***update queries****. Each update query represents that a certain model element has one of its attributes changed, where* $v_h^{Mod} \in V(P)$ *is a variable of* P *that represents the model element,* $attr_h$ *is the attribute name, and the (optional) variables* $v_h^{pre}, v_h^{post} \in V(P)$ *represent the pre-state and post-state values of the attribute, respectively.*
- *Appearance, disappearance and update queries altogether are called* ***change queries****.*
- *The set of common variables of a change query and the main pattern is called its* ***interface****.* $I_i = V_i \bigcap V(P)$, $I_j = V_j \bigcap V(P)$ *and* $I_h = \{v_h^{Mod}, v_h^{pre}, v_h^{post}\}$.
- *The* ***pre-state pattern*** $P_{pre}(CP) = \bigcup_{P_i \in P^*_-} P_i \bigcup P$ *summarizes disappearance queries and the main positive pattern, i.e. all patterns representing existence in the pre-state.*
- *The* ***post-state pattern*** $P_{post}(CP) = \bigcup_{P_i \in P^*_+} P_i \bigcup P$ *summarizes appearance queries and the main positive pattern, i.e. all patterns representing existence in the post-state.*

The match of change patterns (Fig. 8) is defined against a pair of graphs G_{pre} and G_{post}, such that G_{post} is derived from G_{pre} by some (maybe only observable, but not controllable) model manipulation. Thus the sets of model entities (Ent_{pre} and Ent_{post}) and relations (Rel_{pre} and Rel_{post}) may intersect on elements that were preserved by the step from G_{pre} to G_{post}. By definition, *Dom* (the immutable set of attribute values including all integer values, strings, etc.) is the same in both cases. Here G_{pre} and G_{post} represent the pre-state and post-state respectively, but their presence in the definition does not imply that the concept of change patterns is restricted to the snapshot scenario (see Section 2.1) – only to unify the semantic discussion.

All variables of the change pattern that represent attributes are required to be "bound", in order to avoid unintended challenges such as equation solving. There are multiple ways to bind an attribute variable: either by declaring it as the attribute value of a model element, or by participating in the interface (header parameter) of a change query, or by being the result of a function (e.g. addition) on bound attribute variables. Similar restrictions hold for the graph patterns in NACs and change queries.

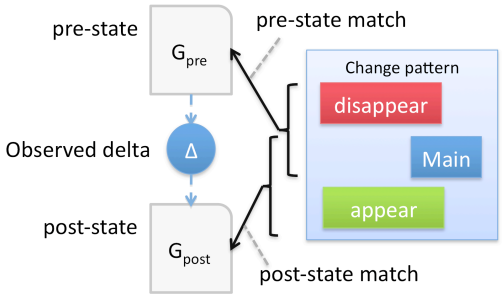


Fig. 8 Change Pattern concepts

Definition 2 (Match of Change Pattern) *A match of the Change Pattern* $CP = \langle PN, P^*_+, P^*_-, U^*_{:=} \rangle$ *in* $\langle G_{pre}, G_{post} \rangle$ *is the mapping* $m = \langle m_P, m^*_+, m^*_- \rangle : CP \to \langle G_{pre}, G_{post} \rangle$, *where*

- $m_P : PN \to G_{post}$ *is a match of* PN, *in the post-state* G_{post}*:* $G_{post}, m_P \models PN$.
- *For each* $P_i \in P^*_+$, *the set* m^*_+ *contains a mapping* $m_i : P_i \to G_{post}$, *such that*
  - $G_{post}, m_i \models P_i$, *i.e.* m_i *a match of pattern* P_i *in graph* G_{post},
  - $m_i(v) = m_P(v)$ *for interface variables* $v \in I_i$, *i.e.* m_i *interfaces with the match of the main pattern, and*
  - $G_{pre}, m_i \not\models P_i$, *i.e. the same* m_i *is not a match in the pre-state.*
- *For each* $P_j \in P^*_-$, *the set* m^*_- *contains a mapping* $m_j : P_j \to G_{pre}$, *such that*
  - $G_{pre}, m_j \models P_j$, *i.e.* m_j *a match of pattern* P_j *in graph* G_{pre},
  - $m_j(v) = m_P(v)$ *for interface variables* $v \in I_j$, *i.e.* m_j *interfaces with the match of the main pattern, and*
  - $G_{post}, m_j \not\models P_j$, *i.e. the same* m_j *is not a match in the post-state.*
- *For each* $U_h = \langle v_h^{Mod}, attr_h, v_h^{pre}, v_h^{post} \rangle \in U^*_{:=}$ *update query,*
  - $G_{pre} \models m_P(v_h^{Mod}).attr_h = m_P(v_h^{pre})$, *i.e. the pre-state value of attribute* $attr_h$ *of* v_h^{Mod} *was* v_h^{pre}, *and*
  - $G_{post} \models m_P(v_h^{Mod}).attr_h = m_P(v_h^{post})$, *i.e. the post-state value of attribute* $attr_h$ *of* v_h^{Mod} *is* v_h^{post}, *and*
  - $m_P(v_h^{pre}) \neq m_P(v_h^{post})$, *i.e. there was a change between two different values, and*

– *if* v_h^{pre} *or* v_h^{post} *is omitted from the update query, its value is considered to be existentially quantified.*

For a match $m = \langle m_P, m_+^*, m_-^* \rangle : CP \to \langle G_{pre}, G_{post} \rangle$,

– *the* ***pre-state match*** *is defined as* $m_{pre} = \bigcup_{P_j \in P_-^*} m_j \bigcup m_P$*, i.e. the unification of the match components corresponding to the pre-state pattern* $P_{pre}(CP)$*; consequently* m_{pre} *is a match of the pre-state pattern in* G_{pre}*, i.e.* $G_{pre}, m_{pre} \models P_{pre}(CP)$*;*
– *the* ***post-state match*** *is defined as* $m_{post} = \bigcup_{P_i \in P_+^*} m_i \bigcup m_P$*, i.e. the unification of the match components corresponding to the post-state pattern* $P_{post}(CP)$*; consequently* m_{post} *is a match of the post-state pattern in* G_{post}*, i.e.* $G_{post}, m_{post} \models P_{post}(CP)$*.*

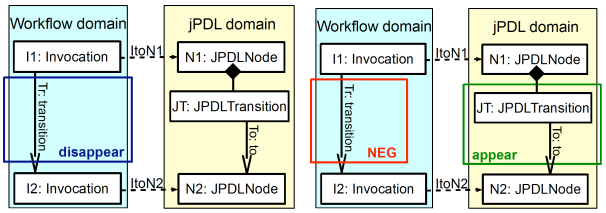
Note that this definition is deliberately asymmetric for G_{pre} and G_{post}, as the main pattern PN is interpreted on G_{post} only. The same holds for P_+^* and P_-^*.

It is also worth noticing that the language feature of update queries is a syntactic sugar. While it helps to concisely define change patterns in common cases and also potentially to increase CP matching efficiently, nevertheless appearance and disappearance queries alone provide enough expressive power. In particular, update query $U_h = \langle v_h^{Mod}, attr_h, v_h^{pre}, v_h^{post} \rangle \in U_{:=}^*$ is equivalent to the disappearance of the value assignment (for attribute name $attr_h$) between v_h^{Mod} and v_h^{pre}, and the appearance of the assignment for the same attribute name between v_h^{Mod} and v_h^{post}. While not indicated before, the definitions for pre- and post-state patterns $P_{pre}(CP)$ and $P_{post}(CP)$ are actually intended to take this representation of update queries into account.

4.2.1 Extensions Although not presented in the formal definition to provide better focus on the core contribution, there is a wide range of straightforward extensions to the presented version of the CP formalism (which is actually part of our language), including negative application conditions in the patterns used as change queries, nested NACs, change queries attached to a NAC of PN (or even a nested NAC) instead of P, etc. Without including a proof, it is worth pointing out that if these features are available, then the expressiveness of CPs becomes equivalent to first-order formulae over the set of predicates describing the pre-state and the post-state.

This suggests that the CP formalism is powerful enough justifying the choice to be used to trigger change-driven rules. There are also some extensions which we will use in later examples of the paper: for convenience, both graph patterns and CPs can be written as the composition of smaller patterns (even facilitating reuse) using the *find* keyword; this can be thought of dependencies between (change) patterns.

4.2.2 Example. Figures 9(a) and 9(b) show the CPs that detect deleted workflow transitions (in order to delete the jPDL transition), and newly created jPDL transitions (to be mapped back into the workflow domain), respectively. NACs are often visually represented as special sub-patterns (enclosed in a "NEG" box), the rectangles marked by `appear` or `disappear` indicate that the enclosed pattern is an appearance or disappearance query, respectively. As said earlier, the CP of Figure 9(b) is insensitive to the last operation that caused the jPDL transition to appear, it can be the creation of the transition, redirecting, moving under a different JPDLNode, etc. Listing 2 displays the same CP with a textual syntax. As an extension to the graph pattern language, change queries are available as a (sub)pattern definition with *appear* or *disappear* prefixes, also having a set of exported or visible variables (including the interface variables) listed in parentheses.



(a) Detecting deleted workflow transition (b) Detecting created jPDL transition

Fig. 9 Example Change Patterns

```
change pattern newJPDLTr(I1, N1, I2, N2, JT) =
{
 Invocation(I1);
 traceability(ItoN1, I1, N1);
 JPDLNode(N1);
 appear pattern (N1, JT, N2) = {
  JPDLTransition(JT) in N1;
  JPDLTransition.to(To, JT, N2);
 }
 JPDLNode(N2);
 traceability(ItoN2, I2, N2);
 Invocation(I2);
 neg pattern (I1, I2) = {
  Invocation.transition(Tr, I1, I2);
 }
}
```

Listing 2 Example Change Pattern

For demonstration purposes, these CPs are matched against the transaction depicted in Figure 5. The pre- and post-states are shown in Figures 10 and 11, respectively. The workflow model is indicated alongside its counterpart in the jPDL domain, but the traceability links and other details such as attributes are not shown. Traceability links are present between $w0$ and $p0$, $i1$ and $n1$, $i2$ and $n2$, $di1a$ and $pei1a$, etc. We assume that the workflow model was changed (in Fig. 11), namely a new $i3$ invocation was created in the workflow, $di3$ input and $do3$ output specification were created within the invocation, a new $t3$ transition edge was created from the new invocation to $i2$, and the old transition $t1$ was retargeted to $i3$. The jPDL model, however, was not changed; trans-

formation rules will have to detect the discrepancy and propagate the changes to the target model.

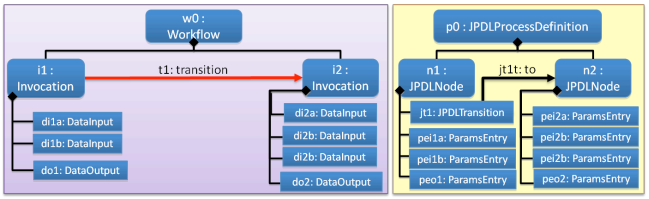


Fig. 10 Pre-State of Example Transaction

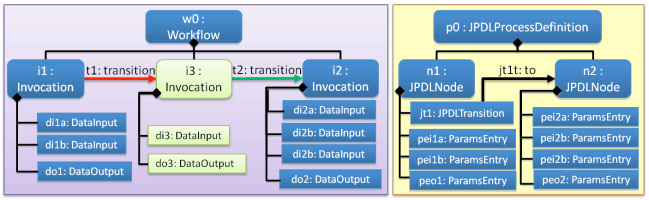


Fig. 11 Post-State of Example Transaction

The CP in Figure 9(b) will not match against this change, as it contains an appearance query in the jPDL domain, but the jPDL target model was not changed. However, the CP in Figure 9(a), has a match. The current snapshot contains the two invocations and their corresponding JPDLNodes, but there was the $t1$ transition relation between $i1$ and $i2$ which is not present anymore (it was not actually deleted, just redirected). As the model manipulation did not change the jPDL part, the jPDL representation of this transition, namely $jt1$, is still present in the current model, so this is a match of the CP. More precisely, the disappearing variable Tr in the change pattern will be substituted for transition edge $t1$, $I1$ in the pattern is mapped to $i1$ in the workflow, $I2$ is mapped to $i2$, $N1$ to $n1$ in the jPDL domain, $N2$ to $n2$, JT to $jt1$, To to $jt1t$, finally $ItoN1$ and $ItoN2$ to the traceability links that are not shown. The occurrence of the CP will trigger the rule, that will be responsible for removing $jt1$.

4.3 Change-Driven Rules

Using our Change Pattern formalism, we can now introduce change-driven transformation rules. GT-style rules consisting of a CP as a LHS/guard (instead of a conventional LHS pattern) and a graph pattern as RHS are *Change-Driven Rules* (CDR). A CDR specifies a reaction to the CP used as its guard. As explained on Fig. 12, the reaction is a controlled change transforming G_{post} into an even newer state G_{new}. The transformation substitutes the match of the guard (more precisely, the match of the post-state pattern $P_{post}(CP)$) with the image of the RHS pattern, with the same semantics as a GT rule application. In fact, the application of the CDR will be defined by a reduction to an application of a GT rule.

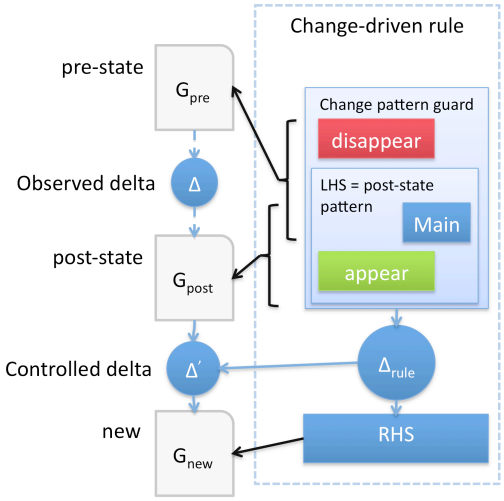


Fig. 12 Change-driven rule concepts

Definition 3 (Change-driven Rule) *Change-driven rules* $CDR = \langle CP, RHS \rangle$ *are specified by a* ***guard*** *change pattern* $CP = \langle PN, P^*_+, P^*_-, U^*_{:=} \rangle$ *defining the applicability of the rule, and a* ***postcondition*** *(or right-hand side) positive pattern* RHS *which declaratively specifies the result model after rule application. The post-state pattern* $P_{post}(CP)$ *and* RHS *are allowed to share variables.*

Obviously, RHS may only use/delete elements that are not already deleted in the CP, hence the usage of the post-state pattern. P_{post} and its match m_{post} in G_{post} will also be used to define the application of the rule. CDR application is the replacement of the post-state pattern with the RHS, or equivalently, the application of a conventional GT rule obtained from the change-driven rule with P_{post} substituted for LHS.

Definition 4 (Elision of a Change-driven Rule) *The* ***elision*** *of a Change-driven Rule* $CDR = \langle CP, RHS \rangle$ *is the Graph Transformation Rule* $R_{CDR} = \langle P_{post}(CP), RHS \rangle$*, whose left-hand-side is the post-state pattern of the guard Change Pattern* CP*, and the right-hand-side is shared with* CDR*.*

Definition 5 (Application of Change-driven Rule) *A change-driven rule* $CDR = \langle CP, RHS \rangle$ *can be applied on a guard match* $m = \langle m_P, m^*_+, m^*_- \rangle : CP \to \langle G_{pre}, G_{post} \rangle$ *after a change from pre-state* G_{pre} *to post-state* G_{post}*. The application of the CDR results in a new graph model* G_{new} *derived from* G_{post}*, where the transition from* G_{post} *to* G_{new} *is identical to the application of the elision GT rule* $R_{CDR} = \langle P_{post}(CP), RHS \rangle$ *on post-state match* m_{post}*. If* CP *has no matches in* $\langle G_{pre}, G_{post} \rangle$*, then* CDR *is not applicable.*

4.3.1 Extensions While the declarative specification of GT rules and CDRs can be very concise in some cases (especially with pattern reuse through composition), in some cases it is more practical to also associate imperative actions to the rule that should be executed on the match of the guard. Examples include logging or debugging, chaining related rules, performing nontrivial computation, etc. Therefore the transformation language used in our examples contains an extension to the core CDR formalism, so that an action sequence can be attached to the rules using the *action* keyword. This technique provides a complete imperative alternative to using the declarative RHS formalism.

Change-driven rules vs. GT rules. It is worth pointing out that both traditional GT rules and an earlier event-driven rule formalism (graph triggers in [3]) can be thought of as special cases of the more expressive CDR formalism. CDR rules reduce to GT rules in case there are no change queries, while CDR rules are equivalent to graph triggers in the case of an empty main pattern (graph triggers use the appearance/disappearance of the entire precondition pattern as guard condition).

4.3.2 Example Figure 13 and Listing 3 show the CDR that propagates transitions created in the jPDL domain back to the workflow model. The guard CP, identical to Figure 9(a), activates whenever a transition disappears between two Invocations, which is still mapped to a transition element between the corresponding JPDLNodes. The RHS does not contain the JPDL transition element anymore, therefore it will be deleted when the rule is applied.

For example, as already discussed, the CP guard will have a match on the pair of states depicted in Figures 10 and 11; the rule will be applied as a reaction, resulting in the deletion of the JPDL transition $jt1$ (and thus the connecting edge $jt1t$) that previously corresponded to the workflow transition.

4.4 Validation

In the following, we summarize our arguments on why this transformation language extension answers the challenges of Section 2.2.1 and satisfies the requirements given in Section 4.1:

- **reactivity**: using change patterns as guards for transformation rules, the transformation can react to changes in the model.
- **conciseness**: change queries capture the relevant information in the delta without the need for individually addressing possible sequences of elementary changes.
- **high-level specification**: model changes can be abstractly captured as appearance and disappearance of graph pattern matches; and change-driven transformations are also independent from the source of the triggering changes.
- **intuitiveness**: our language extends declarative static model queries and model manipulation (as provided by graph patterns and graph transformation rules) in a natural way by introducing change patterns (which are guards that specify elements which must appear or disappear) and change driven rules (which describe reaction to changes).
- **expressiveness**: change patterns allows the transformation designer to specify rules which can distinguish between identical post-states of the model based on the *modification trajectories* which led to that state, without (i) having to encode these modifications into complex traceability models and (ii) bloating transformation rules with them. In other words, change-driven rules extend the expressive power of graph transformation rules by high-level queries corresponding to the *changes* exhibited by the graph.

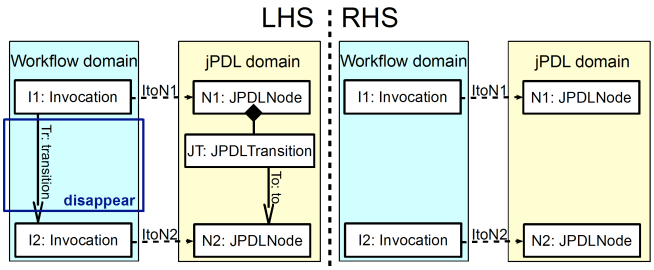


Fig. 13 Propagate deletion of Transition

```
cdrule propagateDelWFTr(I1, N1, I2, N2) =
{
 guard find delWFTr(I1, N1, I2, N2, JT);
 postcondition pattern noJT(I1, N1, I2, N2, JT) =
 {
  Invocation(I1);
  traceability(ItoN1, I1, N1);
  JPDLNode(N1);
  Invocation(I2);
  traceability(ItoN2, I2, N2);
  JPDLNode(N2);
 }
}
```

Listing 3 Example Change Driven Rule

5 Implementation Architecture

The following sections outline a system architecture that implements change-driven transformation. Solutions to the following task items are required:

- (positive and negative) graph pattern matching of the CP's main pattern in the post-state
- evaluating and matching appearance and disappearance queries
- evaluating update queries

– matching change patterns, using the solutions of the above three tasks
– applying change driven rules on matches of the guard change pattern

The first three of these tasks require different implementation techniques in different change scenarios (see Section 2.1), to take advantage of the benefits and avoid unnecessary operations that may degrade performance. First, Section 5.1 discusses our proposed solutions to the first three tasks in all change scenarios except for the live case. Next, Section 5.2 focuses on the live scenario with its unique execution model. Finally, the last two task items are addressed in Section 5.3.

5.1 Change query evaluation in documented or invisible change scenarios

Documented history and command scenarios In the documented change scenarios, either the pre-state or the post-state of the model is available along with a delta document recording the changes. In order to match appearance and disappearance queries, existing graph pattern matcher algorithms have to be slightly modified to operate on a graph that contains the elements of the available snapshot and also the elements that only occur in the delta document. This graph should distinguish unchanged, deleted and created elements.

A match of a positive graph pattern is only considered valid in the post-state, iff it contains no deleted elements. A pattern with NACs has a valid match in the post-state iff it is a valid post-state match of the positive pattern, and all NAC matches (if any) are disappearing (see later). A pattern match is considered appearing iff it is valid in the post-state (as defined above), and contains at least one created element, or has at least one NAC match (which is – as stated above - disappearing). Finally, a pattern match is considered disappearing if it contains no created elements, all of its NAC matches (if any) are appearing, and there is either at least one deleted element of the positive match, or a NAC match (which must be appearing).

Using these rules, the pattern matcher can determine the match set of the main pattern in the post-state, as well as that of the appearance and disappearance queries. Attribute updates are straightforward to be evaluated based on the delta document.

Snapshot scenario In the snapshot scenario, both the pre-state and the post-state are directly available, therefore change query evaluation is reduced to fairly simple steps. An appearance query is satisfied if the pattern is matched in the post-state, but the same match is invalid in the pre-state; and vice versa for disappearance queries. Therefore evaluating these queries require a tailored graph pattern matcher that is similar to algorithms dealing with negative patterns. Finally, whether and how a certain attribute was updated can be detected using referenced value comparison.

However, one of our assumptions here was that if a model element exists in both states, it is trivial to recognize that they are in fact the same element (this is actually required in the definition of Change Pattern). This is possible if model elements have a unique identifier that is preserved across different versions. Unfortunately, in some modeling environments this is not guaranteed; for example, generic EMF objects are not identifiable by default in a way that is valid across snapshots (but fortunately EMF provides both live notifications and redo stacks instead). In this case, the two versions of the model have to be reconciled against each other (by either a generic heuristic or a domain-specific way) before the changes can be computed and CDR can be applied; see the related literature on model comparison [16].

Invisible scenario As only the post-state is available, post-state matching of the main pattern is trivial in this case, but evaluating change queries is not. The common solution to this problem has significant time and space overhead: the transformation creates a *shadow copy* of the model each time it is invoked. On the next transformation run, the model itself represents the post-state, but the shadow copy preserves the pre-state, therefore the change queries can be evaluated. Of course, there is no need to replicate the entire model; it is sufficient to store the match sets of patterns used as appearance and disappearance queries, and to preserve the attribute values corresponding to element types and attribute names involved in update queries. The appearance and disappearance queries can be evaluated by matching the patterns against the post-state and comparing the match set against the one preserved in the shadow copy. Likewise, update queries can be evaluated by comparing the current attribute value against its shadow copy.

To prevent inconsistencies, the shadow copy should be inaccessible to normal model editing operations, which can be achieved either storing it separately (e.g. in a different file), or by using special model element types, markers, etc., that visual editors and other transformations ignore. If it is stored separately, the problem of preserving model element identity has to be dealt with, similarly to the snapshot scenario.

A widespread practice [15,17] is to use the traceability model (sometimes called reference model or correspondence model) in a way that it preserves the LHS (or a significant subset thereof) of all executed rules. Thus the traceability connections essentially store a copy of the source model, thereby providing a shadow copy functionality. In those model transformation approaches where this is not handled automatically, significant manual effort is required for maintaining this shadow copy. With change-driven transformations, however, the platform can provide change queries as a service, hiding im-

plementation details. The hidden implementation will involve an automatic shadow copy mechanism in the invisible change scenario (and less resource intensive solutions in the other change scenarios). This allows a much simpler maintenance of traceability in many cases (especially bidirectional synchronization), sometimes as simple as using the same name for a source and target element, as there is no need to manually preserve the entire LHS.

5.2 Change query evaluation in live change scenarios

Challenge of live scenarios While all techniques for the documented scenarios are functionally correct in the live scenario as well, there may be an additional important requirement in this case. Live notifications can be used to perform live transformation, where change-driven rules can be executed on-the-fly. Since live notification is received about changes that are in progress, and reactions are triggered during an interactive session, pattern matching is required to be responsive and efficient. We propose an architecture capable of efficiently matching change patterns and applying change-driven rules with live monitoring of the model as it evolves.

The entire architecture is illustrated in Fig. 14. The rest of Section 5.2 discusses how change queries are evaluated efficiently in live scenarios.

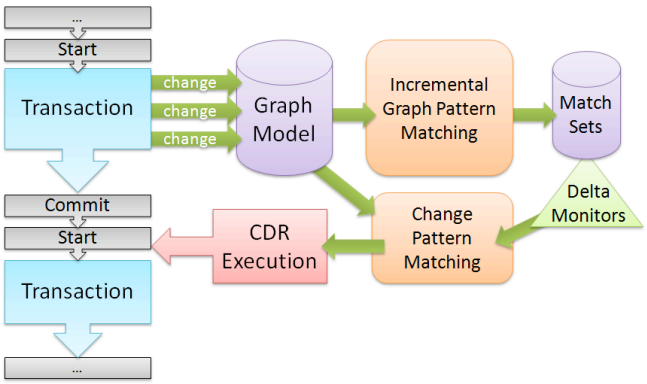


Fig. 14 Implementation architecture for change-driven rules in the live scenario

Incremental Pattern Matching The aim to execute transformations without the costly re-evaluation of unchanged parts of the evolving (source) model is called *source incrementality*. Source incrementality can be achieved by employing incremental pattern matching techniques; for example, the RETE algorithm [18] was used in [19]. The key idea of incremental pattern matching is that matches of a pattern are cached to be readily available at any time, and this cache is incrementally updated whenever notifications are received about changes in the underlying model. Obviously, such a technique results in increased memory consumption in order to store match sets. Furthermore, these stored result sets have to be continuously maintained whenever the model is changed, causing an overhead on model manipulation. Nevertheless, benchmarks [20] indicate that incremental pattern matching can improve performance or scalability by several orders of magnitude in certain scenarios. Moreover, incremental pattern matching leads to easy discovery of appearing or disappearing pattern matches, thus it can be used to efficiently implement the change query feature of change patterns by incremental calculation of matching set differences.

Having received change notifications, the incremental pattern matcher shows an up-to-date picture of the post-state. This is true even in the command scenario where these changes might not have been applied to the model itself, still retaining the pre-state. Therefore the post-state graph pattern matching of the main pattern can be performed by the incremental pattern matcher in both history and command scenarios. In the history scenario, the model itself reflects the post-state, therefore a regular (local search) pattern matcher is also applicable for this task, if it is preferrable for performance reasons.

Delta Monitoring A *delta monitor* is a component that can be attached to a match set cache of the incremental pattern matcher at any time, and it will start to record changes affecting the match set from that time on. At any point in the future, the delta monitor will be able to report about which new matches appeared and which matches disappeared since it was initialized. This is efficiently achieved by hooking into the internal notification/update mechanism of the incremental pattern matcher. Changes of a single match (e.g. the same match appearing and then disappearing later) may invalidate each other, therefore the delta monitor really reflects (a projection of) the delta between the two states, and not just recorded history. A slightly modified delta monitor can be used to remember attribute updates.

Change query evaluation Change queries can be efficiently evaluated using delta monitors and incremental pattern matching. Before the change is performed (or notifications are received), a delta monitor is to be attached (or reinitialized, if already attached) onto the incrementally maintained match set of each graph pattern that occurs as a change query within a CP. After the change, the contents of the delta monitors will reflect the graph pattern matches that have appeared or disappeared. This complements the post-state reflected by the incremental pattern matcher (or alternatively, in the history scenario, the model itself), to provide all necessary information for matching change patterns.

5.3 Implementing Change-Driven Rules

Matching Change Patterns Change patterns are equivalent to an extended graph pattern formalism, where the set of admissible pattern constraints contains *change sets* of pattern matches as hyperedge constraints [21]. In the end, the match set of a change pattern can be determined from the match set of the change queries, the main positive pattern and the negative patterns; and such a pattern matcher architecture is conceptually similar to existing ones dealing with negative application conditions. Therefore graph pattern matching mechanisms can be used to evaluate change patterns, based on the partial solutions (change queries and post-state pattern matching of the main pattern) obtained differently in each change scenario.

Rule execution The sequence of elementary model manipulation operations executed by any transformation unit, GUI-based manipulation, model merge or other job can be arbitrarily segmented into *transactions*, that are assumed to result in a consistent state of the affected model. The transaction is the unit of change that CDRs will react to; the starting and the end points of the transaction will be considered the pre-state and the post-state, respectively. In documented change scenarios, the whole change process between the given pre- and post-states can be considered a single transaction. In live scenarios, as notifications may be continuously sent, it is a nontrivial question how to segment transactions; it helps if there is some support for explicitly defining transaction boundaries and commit points. A typical transaction can be e.g. the execution of single functionality through the UI, corresponding to multiple elementary operations.

Upon the end of each transaction, the change patterns are evaluated to determine which change-driven rules are applicable. If there are any such CDRs, they are applied on the model, using algorithms that are identical to regular GT rule application. As this rule application phase modifies the model, it can be considered a change transaction itself, with its effects wrapped into a separate transaction. At the end of this second transaction, the effects of executed CDRs can be reacted to as well, as long as there are triggered CDRs. This follow-up loop is actually a live scenario, regardless of the circumstances of the original triggering change. This queue-based execution schema has been previously elaborated in details in [3].

6 Elaboration of the case study

6.1 Overview of the approach

In this section, we demonstrate the concept and the application of change-driven transformations (Sec. 2), relying on the novel change-driven transformation formalism of Sec. 4, by the elaboration of the motivating scenario described in Section 3.

6.1.1 Challenges In the current paper, we investigate a model synchronization scenario where the goal is to *asynchronously* propagate changes in the source model M_A to the target model M_B (Fig. 15). This means that changes in the source model are not mapped on-the-fly to the target model, but the synchronization may take place at any later time. However, it is important to point out that the synchronization is still *incremental*, i.e. the target model is not re-generated from scratch, but updated according to the changes in the source model.

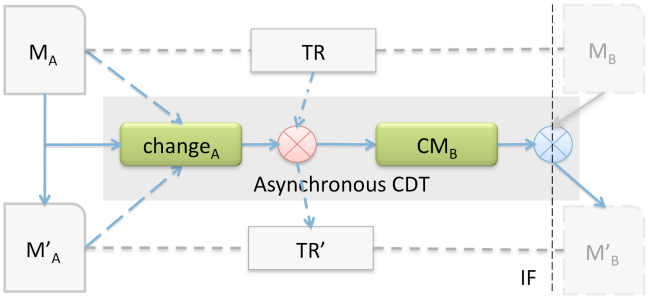


Fig. 15 Model synchronization driven by change models

Moreover, our scenario also requires that M_B is not materialized in the model transformation framework, but accessed and manipulated directly through an external interface IF of its native environment. This is a significant difference to traditional model transformation environments, where the system relies on model import and export facilities to connect to modeling and model processing tools in the toolchain. Here, we only assume the existence of very simple (untyped) traceability links. These links need not be actual persisted references; it is sufficient to establish "virtual" connections defined by using the same ID or name for the two elements in the two models. Moreover, we will also discuss various operations for processing change models:

- **Processing live historical changes.** Based upon the notification mechanism of the underlying model management framework, we can process changes on-the-fly as they are generated by the user or a model transformation. In our case study example (see Section 6.3), these changes (of the source model) will be used in a model synchronization scenario to produce change commands (that can be applied to the target model).
- **Processing documented command changes.** Change commands represented as models themselves can be automatically applied to models. More precisely, one can combine the current snapshot of the target model M (representing the initial state) and a change model CM to create the final snapshot M'. An example for applying a change model will be presented in Section 6.4.

6.1.2 Technology As the source model M_A is in the model space of the transformation engine, model changes are directly observable through a notification mechanism. These changes are processed by change-driven rules ($change_A$ in Fig. 15). However, as the target model M_B can only be manipulated asynchronously, we design the transformation in a way that instead of direct target model manipulations, we encode the changes of the target models as *change command model* instances CM_B. These models conform to a change command metamodel (presented in Fig. 16 of Sec. 6.2), and represent parametrized model manipulation operations.

Hence, the actual model transformation itself is implemented as a mapping between the (in-memory) *changes* of the source language A to change models corresponding to change commands that can be applied in language B (see middle part of Fig. 15). This transformation is described later in details in Sec. 6.3.

As change command models represent a trace of model evolution, they may be automatically applied to models (see right part of Fig. 15). More precisely, such a transformation combines a snapshot of the model M_B (representing the initial state) and a change command model CM_B (representing a sequence of operations applicable starting from the initial state) to create the final snapshot M'_B. In other words, the change command model CM_B represents an "operational difference" between M'_B and M_B, with the order of operations preserved as they were actually performed on M_B.

As change-driven transformations can transparently process both observable and persisted changes (Sec. 2), we use our rules to map change command models into native function calls that directly manipulate the external target model (Sec. 6.4).

6.2 Change models

First, we briefly overview how to persist changes as change models. For this elaboration, we only concern observable (and non-controllable) changes (the representation of controllable changes - e.g. as graph transformation rules - has been discussed extensively in literature [22,23]).

To clarify and capture the notions of domain-specific model changes precisely, we present a simplified classification system for jPDL model changes based on a metamodel (Fig. 16).

By the terminology of Section 2.1, both *historical* and *command*-type changes represent a difference (*delta*) between a pre-state and a post-state of the model. They differ only in their interpretation: histories are valid with respect to a post-state, while commands may be applied to a pre-state. Hence, in our model-based taxonomy, both notions are represented by the *jPDLCommand* type. The *next* relation enables the representation of operation sequences (transactions). Change commands contain *ordered* elementary change operations.

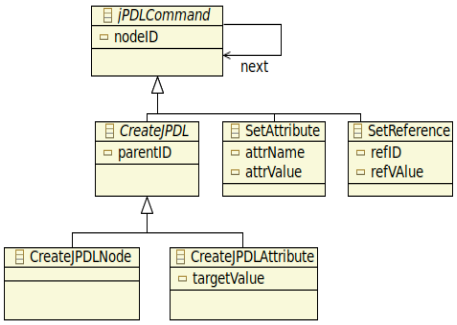


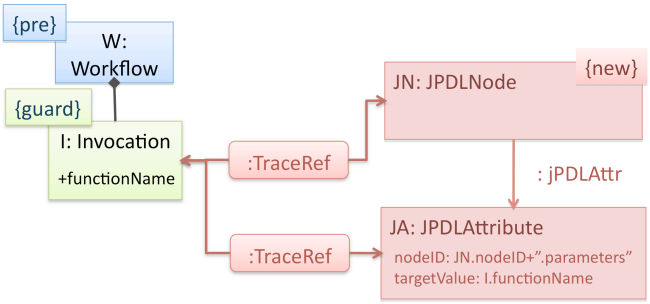
Fig. 16 Change command metamodel for jPDL

This metamodel uses unique IDs to refer to (non-materialized) model elements (as defined in the jPDL standard); since jPDL documents also follow a strict containment hierarchy, creation operations (as depicted in Fig. 16) refer to a $parentID$ in which an element is to be created. In the follow-up examples of our case study, we will make use of *CreateJPDLNode* and *CreateJPDLAttribute* to illustrate the usage of this domain-specific change history metamodel.

6.3 Propagating changes by change-driven transformations

In this section, we describe a sample transformation rule where the creation of an *Invocation* in the domain-specific workflow language is mapped to the creation of a corresponding jPDL Node and its attribute (Fig. 17). This is a practical elaboration of the "live historical" scenario described in Section 2.1.

We use a change-driven graph transformation rule, which defines a guard condition that triggers when a new *Invocation* node is added to the domain-specific workflow model. The guard consists of a context definition (the *common pattern* referring to the already existing *Workflow* node) and the change pattern (referring to the newly created *Invocation* node). In the postcondition part, the corresponding jPDL-specific change command model elements, along with the traceability model are declared. As *Invocation*s are represented by jPDL Nodes with an attribute node, the target change model will consist of two "create"-type elements, chained together by the *jPDLCommand.next* relation. The simple traceability model consists of two mapping nodes that connect the *Invocation* node to its counterparts in the target model (the JPDLNode and the JPDLAttribute). In this example, we make use of the fact that both source and target models have a strict containment hierarchy (all elements have *parent*s), which is used to map corresponding elements to each other:




```
transformation mapInvocation() = {
 /* This is a common static part of the pattern,
  * matched for all change patterns
  * and postconditions.
  */
 common pattern()= {
  Workflow(WF);
 }
 /* This cdrule processes the creation of
  * an invocation node
  */
 cdrule newInvocation() = {
 guard appear pattern (I) = {
   Invocation(I);
   Workflow.nodes(_,WF,I);
 }
 postcondition pattern()={
 /* create a JPDL Node and its corresponding
  * attribute node in the JPDL-DOM */
  CreateJPDLNode(JN) {
    nodeID -> I.ID;
    parentID -> WF.ID;
  }
  CreateJPDLAttribute(JA) {
    nodeID -> I.ID+".function";
    parentID -> I.ID;
    targetValue -> I.functionName;
  }
  jPDLCommand.next(_,JN,JA);
  // traceability models
  // one for connecting the invocation node I1
  // to the JPDL node
  TraceRef.External {
    sourceIDs -> [I.ID];
    targetIDs -> [JN.ID];
  }
  // and one to its attribute
  TraceRef.External {
    sourceIDs -> [I.ID];
    targetIDs -> [JA.ID];
}}}
```

Fig. 17 Example change-driven transformation rule

– Based on the new node I in the source model, we calculate the target parent's ID *parentID* as *WF.ID*.
– Similarly, the target jPDL node's ID is *I.ID*
– The jPDL Attribute node's ID will be calculated as *'I.ID'.function* to place the target node under the target parent.
– Finally, the attribute *functionName* designates a particular function on a remote interface which is invoked when the workflow engine interprets an *Invocation* workflow node. It is represented by a separate node in the jPDL XML-DOM tree. The *targetValue* attribute of the additional *CreateJPDLAttribute* element is derived from the appropriate attribute value of *Invocation* node in source model.

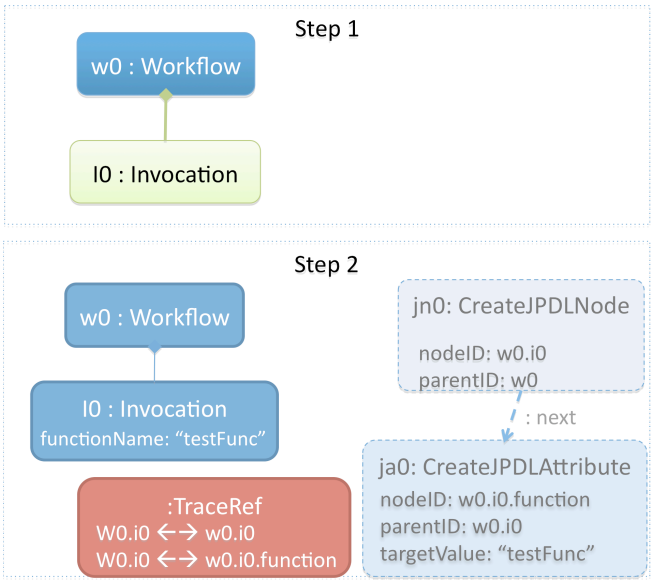


Fig. 18 Example execution sequence

Sample execution sequence Fig. 18 shows an example execution sequence of this rule. The sequence starts with a model consisting only of a top-level container node $w0$ of type *Workflow*. In Step 1, the user creates a new *Invocation* node $i0$ inside $w0$. The change-driven transformation engine triggers the execution of `mapInvocation()` only if the subgraph $w0$ – $i0$ is complete. In Step 2, `mapInvocation()` is fired, and the appropriate jPDL change command model instances are created.

6.3.1 Handling attribute changes Updates to attribute values in the source model can be easily processed by change-driven transformation rules. Fig. 19 illustrates a case where the *functionName* attribute of *Invocation* nodes is being observed; this rule maps a change in the attribute value to a Change Command that can later be executed on the target model that will update the jPDL-DOM attribute value accordingly. The target node in the jPDL model ($jPDL\_ID$) is identified by using the (external) traceability model element referenced in the *common pattern* of the rule.

Note that for bidirectional synchronization with attributes, traditional GT-based formalisms would require storing the attribute value in the traceability model in order to be able to detrmine the direction of change.

6.3.2 Mapping complex operations Certain (simple) changes of the source model may require to be mapped to complex operations on the target model. This is typically the case when the mapping between source and target languages is not a one-to-one correspondence, but a more complex abstraction. All such complex operations can be performed even when using change commands instead of direct model manipulation, as the example in Fig.20 illustrates.

```
/* This cdrule handles the change
 * of the function's name
 */
 cdrule changeFunctionName() = {
  common pattern(I1,jPDL_ID) = {
   TraceRef.External {
   sourceIDs -> [I1.ID];
   targetIDs -> [jPDL_ID];
  }
  guard change pattern() = {
   Invocation(I1);
   update I1.functionName from _OldName to NewName;
  }
  postcondition pattern()={
   setAttribute {
    nodeID -> jPDL_ID.function;
    attributeName -> 'targetValue';
    attributeValue -> NewName;
}}}
```

Fig. 19 Attribute update processing by CDTs

```
/* This cdrule handles deletion of an
 * invocation node in a complex way.
 */
 cdrule mapDeleteInvocation() = {
   common pattern() = {
    Workflow(WF);
   }
   guard change pattern jpdlNodeOrphaned(JPDL_ID) = {
    disappear pattern invocationNode(I2) = {
      Workflow.nodes(_,WF,I2);
      Invocation(I2);
     }
    TraceRef.External {
      sourceIDs -> [I2.ID];
      targetIDs -> [JPDL_ID];
    }
   }
   postcondition pattern()={
/* do not delete corresponding JPDL node (if existent)
 * but first  (1) mark it as deleted
 * and second (2) remove it from the process graph
 * by replacing it with a wait state
 */
     setAttribute(SA) {
       nodeID -> JPDL_ID;
       attributeName -> 'isDeleted';
       attributeValue -> 'true';
     }
     CreateJPDLWaitState(CJWS) {
       nodeID -> JDPL_ID;
       parentID -> WF.ID;
     }
     jPDLCommand.next(_SA,CJWS);
   }
   action {
// connect all preceding nodes to the wait state
     let PreviousCommand = undef in
     forall PN with find precedingNode(PN,I2) do seq {
       new setReference(SR) {
         nodeID -> PN.ID;
         referenceID -> 'transition';
         referenceValue -> CJWS.nodeID;
       }
       new jPDLCommand.next(_,previousCommand,SR);
       update previousCommand = SR;
     }
}}}}
```

Fig. 20 Mapping changes to complex operations

In Fig. 20, the change-driven rule *mapDeleteInvocation()* is shown. This rule triggers if a previously mapped *Invocation* node is deleted (indicated by the combination of the *TraceRef.External* model element and a change pattern that triggers when an *Invocation* disappears from the graph). Upon the registration of this event, this rule will formulate the following commands to be performed on the target model:

1. First, the corresponding jPDL node will be located and marked as deleted (by setting the *isDeleted* attribute to true with a *setAttribute*-type jPDLCommand). Next, a new *JPDLWaitState* will be created to replace the deleted *JPDLNode*.
2. Finally, a more complex modification sequence is defined that disconnects the *JPDLNode* from the jPDL process graph, by overwriting the "transition" references of all its preceding nodes to point to the newly created *WaitState*.

Note that the newly created jPDLCommand elements are connected sequentially by using the *PreviousCommand* helper variable through the forAll cycle in Fig. 20.

6.4 Applying change models to non-materialized models

In this section, we elaborate a technique of processing "documented historical" changes (in the terminology of Section 2.1). On the macro level, change models are represented as chains of parametrized elementary model manipulation operations. As such, they can be processed linearly, proceeding along the chain until the final element is reached (thus modeling the execution of a transaction). The consumption of a change model element is an interpretable step with corresponding actions performed in the context defined by the change model's references.

As Fig. 15 shows, we apply change models to manipulate non-materialized models through an interface. The speciality of this scenario is that instead of working on directly accessible in-memory models, the transformation engine calls interface functions which only allow basic queries (based on ids) and elementary manipulation operations. In this case, change models are very useful since they allow *incremental* updates, as they encode directly applicable operation sequences.

In order to apply change commands to external models, the transformation engine is augmented with a "native model access interface" component. This interface allows for query and manipulation operations as follows: for the jPDL models of the motivating scenario, we mapped the XML-DOM process model manipulation programming interface to VIATRA2's *native function* API. The following native functions are used:

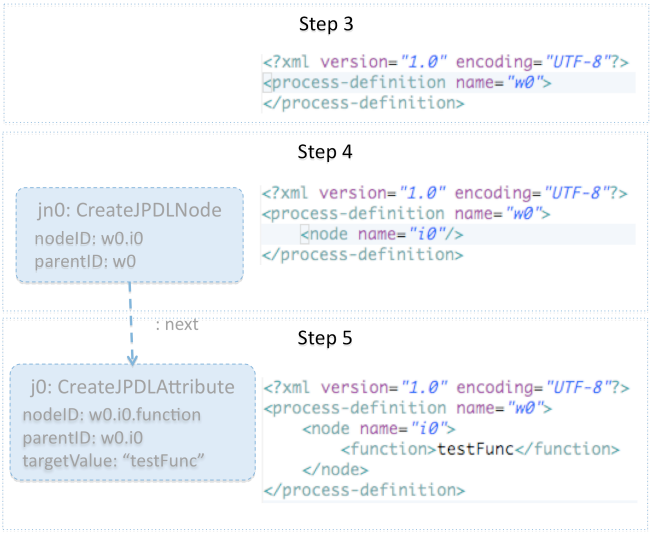
- *getElementById(ID)*: retrieves a jPDL element identified by its unique ID.
- *createElement(parentRef,targetID)*: creates a new jPDL DOM element as a child of its parent (identified by *parentRef*), with a given unique ID (*targetID*).
- *addElement(DocID,elementRef)*: adds the element *elementRef* to the jPDL DOM identified by *DocID*.

– *addElementToParent(parentRef,elementRef)*: adds the element *elementRef* to the jPDL DOM's node identified by *parentRef*.
– *setContents(elementRef,text)*: sets the textual content of the given DOM element (*elementRef*) to *text*.

In this scenario, the mapping rules and the simple traceability implementation (direct id mapping) allow for the changes to be mapped and applied in a straightforward way, without complex navigation and queries on the target model.

Example transformation rule In this case study example, we define a change-driven application rule based on domain-specific change commands for the jPDL XML-DOM model (Fig. 16). Fig. 21 shows the `newCompoundJPDLNode()` rule, which is used to interpret a subsequence of change model chains for the jPDL domain. More precisely, this rule's guard matches for a condition that describes that a new jPDL node, along with its "function" attribute is to be created. In the change command model representation, this corresponds to the pair of *CreateJPDLNode* and *CreateJPDLAttribute* change model fragments. The rule uses native functions `createElement`, `addElement` to instantiate new jPDL XML elements directly in the deployed process model on the workflow server; and `setContent` is used to overwrite the attribute node's textual content.

The upper part of Fig. 21 shows the final three steps of our running example. In Step 3, the initial state of the deployed workflow model, the process definition corresponding to *Workflow* $w0$ is still empty. During the rule execution, first, the jPDL Node $i0$ is created (Step 4), and then in Step 5, the attribute node is added with the appropriate textual content. The entire algorithm which applies change models follows the linear sequence of operations along the relations with type *jPDLCommand.next*; the first operation in a transaction can be determined by looking for a change model fragment without an incoming *jPDLCommand.next* edge.




```
/*
 * This CD rule maps JPDL nodes to the
 * external JPDL-DOM tree.
 */
transformation mapNode() = {
 /*
  * This cdrule handles the case when a new JPDL node
  * is added together with its "function" attribute.
  * This is checked by giving an attribute condition
  * on the ID suffix.
  */
 cdrule newNode() = {
  guard change pattern() = {
    appear pattern () = {
      JPDLNode(JN) {
        JPDLAttribute(JA);
        check(JA.nodeID == JN.nodeID+".function");
      }
    }
  }
 action {
 // create JPDL Node - Step 4
 // we manipulate external models here
 call DOM.addElement(JPDLRoot,
       DOM.createElement(JN.parentID,JN.nodeID));
 // create JPDL Attribute - Step 5
 let DOM_ANode=DOM.createElement(JN.nodeID,JA.nodeID);
 DOM.setContent(DOM_ANode, JA.value);
 call DOM.addElementToParent(JN.nodeID,DOM_ANode);
}}}
```

Fig. 21 Applying change models through the jPDL XML-DOM API

7 Case study on evolutionary constraints

A second motivating scenario is from the domain of security requirements engineering, using the Air Traffic Management case study of the SecureChange European FP7 FET project. A requirements model assists security engineers to capture security-related aspects of the system, to analyze the security needs of the stakeholders, and to argue about potential security threats. The concepts of a security requirements modeling language such as SecureTropos [24] typically include actors and their goals, security-critical information assets and other resources, actions to fulfill goals, trust relationships, delegation of responsibility over assets, goals or actions, etc. An important role of security requirement models is to support reasoning about security properties by formal [25] or informal [26] argumentation techniques in an early stage of development. These arguments may support or refute the satisfaction of security requirements, and are built upon assumptions and ground facts. Some of these facts originate from the security model, and are called *evidence*.

7.1 Metamodel for security requirements

To introduce this case study, we use a simplified security metamodel shown in Fig. 22. The model contains *Arguments*, that record a formal or informal reasoning process conducted in the past. An Argument is linked to requirement model elements that support it as evidence.

The requirements model presented here is a simplified actor model based loosely on i* [27].

An *Actor* is a stakeholder or an autonomous part of the system. *Task*s are performed by Actors to support fulfilling requirements. Actors can provide *Resources* that are valuable data assets. *Communication* elements represent that a sender Actor makes some data assets available to a receiver Actor. Several important element types such as *Goal*s or *Trust* are missing, as they are not relevant to the example scenario; *Communication* here replaces a more general class of *Delegation* relationships for the sake of simplicity; attributes are also omitted.

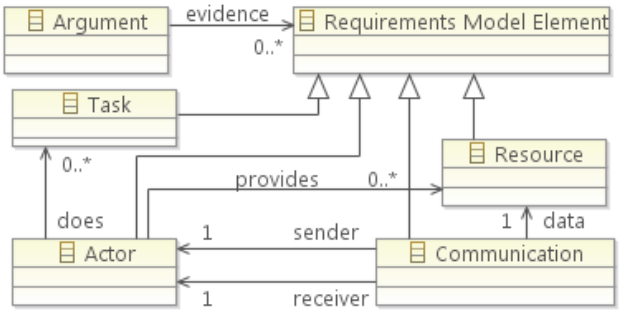


Fig. 22 The security metamodel used in the example

7.2 Sample model

Fig. 23 shows a security model for air traffic communication systems. The three actors are Air Traffic Controller (*ATCO*), Airlines (*AL*) and Catering Services (*CS*). Actor *ATCO* provides resource *RAS* (runway assignment), and communicates it to *AL*. Actor *AL* provides resource *MO* (meal orders), and communicates it to *CS*.

The integrity of data asset *RAS* is security-critical, because if terrorists were able to change *RAS*, they could make planes crash. To ensure the integrity of *RAS*, *ATCO* carries out the following Tasks:

- T_1 "Use data security and secure communication technologies", to make sure the Air Traffic Controller is in control of *RAS*.
- T_2 "Conduct a yearly IT security training of the whole staff", to reduce the likelihood of social engineering attacks.
- T_3 "Enforce policy that every manual decision has to be approved by a second member of the controller staff", to reduce the impact of human error or malice.
- T_4 "Perform a quarterly security screening of employees", monitoring whether an employee has big debts, or can be blackmailed into helping criminals, or has befriended terrorists, etc. to reduce the likelihood of malice.

To decide whether the integrity of *RAS* will be maintained, security experts conduct an informal argumentation analysis *ARG*. Based on the model and their background knowledge, they judge that they are confident in this security requirement. Tasks T_{1-4} and Resource *RAS* are used in their argument, so they are recorded in the model as the evidences for *ARG*.

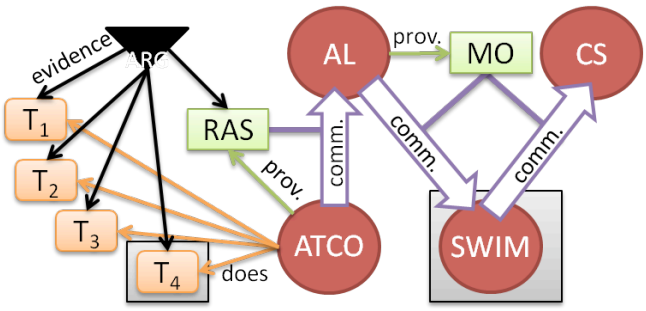


Fig. 23 An example security model and two evolutions

7.3 Example evolutions

An evolution triggering re-evaluation A possible evolution of the model is the following: to cut back costs, *ATCO* plans to reduce the frequency of security screenings, so T_4 will be modified to "Perform a yearly security screening of employees". Thus change is inflicted on an evidence for *ARG*. Since we have no formal way to determine whether the modified Task can fulfill the security needs, the argumentation experts are alerted to revisit argument *ARG*. They decide e.g. that the security requirement is still met with the weakened guarantees, based on regulation, previous experience and a risk analysis conducted by Risk Engineers.

An evolution not requiring re-evaluation Another possible evolution is that the communication between *AL* and *CS* is now routed through a new Actor *SWIM* (System-Wide Information Management). This change can have a wide influence on the system, but it does not invalidate the argument *ARG*, as no evidence of *ARG* was involved in the change. Therefore this time there is no need for the argumentation experts to exert further manual effort.

7.4 Evolutionary constraints

Security requirement models have their set of well-formedness constraints, ensuring that the model is meaningful and consistent. We discussed graph pattern-based on-the-fly validation of well-formedness constraints in a preceding work [3]. These constraints are *static* in the sense that they only restrict the current state of the model. However, there are cases where *evolutionary*

constraints are needed, that can take into account the change that is applied to the model.

In the example evolution outlined in Section 7.3, invalidating informal arguments was such a problem. Formal and informal argumentation is carried out using the requirements model, to determine which security requirements are met. This argumentation is a laborious and costly process requiring significant human expertise. In evolving security-critical applications, it is important that the argumentation is only carried out for those security requirements that are influenced by the change of the model. Thanks to the traceability from Argument to evidence, there is enough information to determine which arguments need to be re-evaluated. The argument has to be invalidated if one of its evidences is involved in a change. This, however, cannot directly be determined by using only the present (post-state) of the security model.

The challenge is to provide a straightforward and efficiently evaluated declarative language for this purpose. We propose on-the-fly, incremental evaluation for a wide range of evolutionary constraints that can be implemented as an efficient reactive mechanism by using Change Patterns and Change-Driven Rules.

7.5 Change-Driven Transformation for Evolutionary Constraints

We propose establishing a set of CDRs that would flag invalidated arguments as invalid and request argumentation analysis. The key difference between each of these rules is the guard Change Pattern. The recommended strategy is to identify types of changes that warrant a re-evaluation of the argument, and define a rule for each of them.

To aid in building CPs and CDRs, some helper Graph Patterns are defined first. Graph Pattern $validArgument(A)$ captures an Argument A that has not been invalidated, while GP $evidenceOfArgument(A, E)$ captures an argument A and a model element E which it references as an evidence.

The CDR *invalidateUponEvidenceUpdate*() is activated when an attribute of an evidence element is updated. The rule is guarded by a CP that contains an update query linked to a match of $evidenceOfArgument(A, E)$ and $validArgument(A)$. The following code sample shows an initial version of this rule in a simplified syntax:

```
change pattern evidenceUpdated(A,E) {
        // static condition
        find validArgument(A);
        find evidenceOfArgument(A,E);
        // event: element updated!
        // attr.name and values are ignored
        update E._ from _ to _;
}
cdrule invalidateUponEvidenceUpdate(A,E) {
        guard evidenceDeleted(A,F,M);
        action {
                call flag_as_invalid(A);
        }
}
```

As an example for the application of this CDR, suppose that the evolution described in Section 7.3 is carried out. This means that Argument ARG is a match of $validArgument(A)$; at the same time, ARG and T_4 ("Perform a quarterly security screening of employees") constitute a match of $evidenceOfArgument(A, E)$. Whenever an evolution updates an attribute of this Task (e.g. downscale to yearly screening to cut costs, as in the example), the update query will detect this, making (ARG, T_4) a match of the change pattern $evidenceUpdated(A, E)$ and activating the CDR. The rule will flag the argument ARG for re-evaluation; argumentation experts will be alerted to revisit the argumentation and decide whether the looser policy is enough to maintain security needs.

Similarly, a second CP could capture arguments whose evidence is deleted. Due to the flexibility of the CP formalism, additional similar rules can be created depending on system-specific policies; for instance the argument should be invalidated if a model element is of a certain type, and edges of certain types are connected to (or disconnected from) it.

8 Discussion

Now we discuss the potential advantages and limitations of change-driven transformation over traditional model transformation techniques.

8.1 Theoretical discussion

In this theoretical discussion, we primarily focus on graph transformation based approaches, which provide the closest correspondence to our techniques, moreover, they have a sound, well-established underlying theory.

Theoretical expressiveness of change driven transformations While not formally proven in the current paper for space considerations, it is worth pointing out that for each change-driven transformation system (CDTS) a corresponding graph transformation system (GTS) can be derived, which simulates the CDTS. As a result, *our CDT formalism is not more expressive in a pure theoretical point of view, but it is still Turing complete.*

The construction essentially builds upon storing explicitly the pre-state as a dedicated part of the model. Then a separate set of GT rules would be responsible for (1) the detection of change and management of pre-states and post-states (based on the difference between G_{pre} and G_{post}) and (2) simulating the effect of a CDTS (based on the difference between G_{post} and RHS). As

a consequence, both the underlying model and the rule set would explode.

Without this special encoding, the GT rules are less expressive as they take only the post-state into account to determine which action to take while CD rules can refer to the pre-state as well.

As a side remark, a CDTS can be constructed even if the model transformation problem itself is nondeterministic by its nature, thus it is not a function but a relation between the source and target language (e.g. a tree-based priority model needs to be flattened to an arbitrary sequence respecting priority relations).

Analysis of change-driven transformations The main practical relevance of this simulation property from CDTS and GTS is that it enables to investigate traditional semantic properties like termination or determinism using the rich theory of graph transformation systems. For instance, if the simulating GTS can be proved to be terminating, then the original CDTS is terminating as well. As a consequence, *many existing analysis techniques of GTS are reusable for CDTSs.*

Constraint detection by change-driven transformations Declarative specifications (like graph patterns, graph transformation rules, OCL constraints) are frequently used for detecting the violation of well-formedness constraints in domain-specific models. However, constraints related to the temporal behavior or the evolution of models are very hard to specify and detect as it requires to explicitly encode the sequence of model snapshots as part of the model, and thus also part of the constraint. *Change patterns provide a direct and succinct way to detect a class of constraints related to the trajectory of model evolution* (see the case study of Sec.7).

8.2 Expressiveness wrt. model synchronization languages

As CDT rules are unidirectional, CDT specifications can be more complex than that of bidirectional approaches (like TGG or QVT Relations) as two separate rule sets are required. However, CDT specifications are also not as complex as they appear to be at first sight. First, the change pattern of a CDT rule compacts many different change trajectories, and only triggers for reaction once a complex (aggregated) change has been detected (disregarding the order of elementary changes). Furthermore, different changes requiring the same reaction can be grouped together in one CDT rule.

Change patterns vs. elementary changes as guards. A naive approach would be to use elementary change events (e.g. element creation / deletion) as guards [14]. It is not a priori known which kind of elementary model manipulation operation will eventually trigger a transformation step. Therefore this low-level formalism forces us to define a separate copy of the transformation rule (or very complex disjunctive preconditions) for each possible triggering elementary change, and augment each rule with a check to see whether the elementary change really triggers the reaction. The high-level formalism of change patterns can trigger reaction when a compound event occurs, thus it significantly compacts the specification of guards.

Change driven rules vs. triple graph grammars (TGGs). While being probably more complex than TGGs, change driven rules are also not as complex as they may seem. First, change patterns correspond to many different change trajectories, and only trigger for reaction once a complex (aggregated) change has been detected (disregarding the order of elementary changes). Consequently, traceability representation can be significantly simplified with respect to TGGs and QVT, since traceability models are not required to contain complex information about change trajectories (in contrast to e.g. [15]). Finally, triple graph grammars handle deletion of source elements by fully revoking the effects of the corresponding synchronization rules. As a result, the dependency between TGG rules has a significant effect on which parts of the target model need to be removed as a consequent undo action. Change-driven transformations allow a more fine-grained and explicit control for delete and move operations in source models to significantly reduce the amount of undos in the target model (by allowing temporal inconsistencies, for instance).

Causality and dependency between CDT rules First, causality and dependency between CDT rules can be handled implicitly using some traceability links in change patterns, which is conceptually similar to the TGG approach, and it does not require the additional use of *when* and *where* clauses as in QVT Relations. However, *dependency and reusability are offered on the (change-)pattern level*, thus complex main patterns, appearance and disappearance patterns can be assembled *using pattern composition*. Furthermore, we may (imperatively) call an arbitrarily complex batch transformation at any time as a reaction to a specific aggregated change.

In the future, we will further investigate how existing MT languages can be translated into CDTs to further reduce the complexity of CDT specifications in case of model synchronization.

8.3 Practical discussion

From a practice-oriented viewpoint, we will investigate (1) the traceability representation between source and target models, (2) the representation of the pre-states for model synchronization scenarios, (3) the handling of non-materialized models.

Traceability information The most apparent advantage of change driven transformations compared to TGG or QVT Relations is that they impose significantly weaker assumptions on the nature of traceability models required during the transformation. Both TGGs and QVT require a real mapping (correspondence) model to interconnect source and target models with typed traceability links which need to be persisted either in a model store (in case of TGG) or in the transformation context (in case of QVT). In case of CDTs, traceability links can be untyped, they can be stored in an external repository (independently of the model store or the transformation context), which may only persist the unique identifiers of source and target model elements. As the extreme case, traceability can also be provided on-the-fly by a function (e.g. a naming convention or identifier map) between the source and target models without persisting traceability information to a dedicated store. As a result, in case of CDTs, source and target models can be almost fully detached from each other in case of model synchronization scenarios using *very simple traceability links or on-the-fly, non-persisted traceability information (traceability function).*

Information about past In case of traditional declarative model transformation approaches (like TGGs or QVT) model synchronization is driven by the traceability information between the source and target models. For instance, if a source (or target) element is freshly created, it is detected by the lack of corresponding traceability element. Alternatively, the deletion of a source (resp. target) element can be observed by a dangling traceability element, which is only linked to a target (resp. source) element. This means that a large amount of information about the past is stored explicitly as part of the traceability model in the model store (or transformation context) in case of traditional model synchronization approaches. In case of CDTs, the effects of transactions are propagated incrementally to the change patterns and change-driven rules, and instead of storing information about the past, it is the change in the match sets of patterns and rules which can be observed to trigger synchronization. Depending on the actual change scenarios, *CDTs significantly reduce what information needs to be stored about the past in model synchronization problem.*

Non-materialized models Traditional model transformation approaches typically assume that both source and target models are materialized within a unified model store (e.g. both models are EMF models). However, in many practical tool integration scenarios, some models can only be accessed in their own environment using its dedicated API when duplicating such external models in an generic model store is not a viable option. *Change driven transformations can easily handle non-materialized models based upon the caching of match sets and the incremental processing of change information provided by transaction logs.*

9 Related Work

Now an overview is given on various approaches showing similarity to our proposal.

Event-driven techniques Event-driven techniques, which are the technological basis of change-driven model transformations, have been used in many fields. In relational database management systems (RDBMS), even the concept of triggers [29] can be considered as simple operations whose execution is initiated by events. Later, event-condition-action (ECA) rules [12] were introduced for active database systems as a generalization of triggers, and the same idea was adopted in rule engines [30] as well. However, ECA-based approaches lack the support for triggering by complex graph patterns, which is an essential scenario in model-driven development.

High-level transformation specification. Event-driven transformation specification can be avoided by using very high-level transformation specification formalisms. OMG's Query/View/Transformation (QVT) specification [31], in particular the Relations part, aims at declaratively defining a relation between corresponding source and target models; it is up to an execution platform to exert event-driven behavior in order to maintain this model. While pointing out the advantages of such a solution, [32] highlights issues with the ambiguity of interpretation and implementation of QVT, in the context of bidirectionality.

Inconsistency management Inconsistency management systems aim at ensuring the consistency of multiple views of a software, which is designed by several engineers using tightly or loosely integrated tools. Views can be formulated on different levels of abstraction, and a bidirectional consistency of views is maintained by inconsistency detection and resolution.

Since these systems should typically support informal (e.g., natural language-based) descriptions as views, inconsistency resolution can never be fully automated, and manual user interaction in certain scenarios is unavoidably required, in contrast to our approach, which *automatically propagates and transforms change descriptions in a well-defined, rule-based way to the target domain* to avoid the appearance of inconsistencies in the target model.

[33] presents a characteristic representative of inconsistency management systems, which records modification histories in the form of (model-based) change description objects just like our approach. In contrast to our solution, [33] additionally saves and stores the detected inconsistencies for their possible resolution at a

later time. The so-called grouping of inconsistencies in this approach would possibly allow for reaching a goal that is similar to the aim of pattern matching in the current paper, however, in [33] grouping is only used for presentation purposes, i.e., to create change and inconsistency lists for users to interact with.

[34] provides a conceptual architecture and prototype for supporting traceability and inconsistency management between software requirements descriptions, UML-style use case models and black-box test plans. Relationships between high-level software artefacts are represented by traceability links, which can be defined manually or in a semi-automated way. In contrast to our solution, this approach supports change notifications on a low abstraction level, it can transform only simple modifications automatically, while other changes still need developer intervention.

In graph transformation, [23] presents an approach for consistency management between abstract and concrete syntax representations of visual modeling languages. By their approach, the commands executed through the user interface are explicitly materialized as special command model elements and then processed by triple graph grammar (TGG) rules. This approach is a prime example for the "controlled" change processing scenario, where all possible editing operations are a-priori known; in contrast, our technique primarily targets non-controllable change processing (while we also cover controllable change scenarios through domain-specific command models).

[17] deals with consistency maintenance in UML models. This paper proposes target incremental techniques to efficiently detect inconsistencies and derive proposed corrections; recommended changes are represented explicitly (such as "DoesExist" and "ShouldExist"). This approach is based on storing very detailed traceability information about rule execution in order to determine when and how a rule should be re-executed for fixing inconsistences; in contrast, our approach is focused on reducing the amount of necessary information persisted in models.

[15] presents a unidirectional, target incremental batch transformation language for model synchronization. Between two synchronization runs, the user may modify the source as well as target models, and the system will then propagate the changes incrementally, leaving manual target modifications intact. This technique again relies on massive amounts of information cached in traceability models, by copying certain parts of the source model intro traceability models.

Software evolution approaches Software evolution approaches, which focus on the temporal development of system (meta)models, can be considered as a possible application area of our approach, which could generate deltas (for different modeling domains) as inputs for the merging process required in software evolution. However, note that our approach does not further support the merge conflict resolution subtask in any sense.

[6] lays down a wide-range terminology used in software evolution. According to this framework, snapshot, command, and history scenarios of Section 2.1 directly correspond to state-based, forward and backward delta approaches, respectively. Moreover, our solution can be categorized as an operation and intensional change-based approach as model changes are explicitly expressed as transformations, and they are independent from the versions to which they are applied.

The FAMOOS project [35] whose aim was to build a framework to support the evolution and reengineering of object-oriented software systems used languages FAMIX [36] and Hismo [37] for modeling purposes. More specifically, FAMIX is a language independent model of object-oriented systems, which can be used for exchanging information between reengineering tools. FAMIX can be considered as a simplified metamodel for class diagrams without any support for describing changes. Hismo [37] extends metamodels by adding a time layer on top of the structural information, and it provides a common infrastructure for expressing and combining evolution and structural analyses. The additional time layer enables Hismo to support version control and to calculate changes of models, and in this sense, it could serve as a source of input for our approach, but Hismo has no metamodel for describing changes on a high abstraction level.

Visualization tools in the FAMOOS framework use side effect free OCL-based queries, which can even involve constructs from the time layer, but these queries are imperative from the viewpoint of structural constraint navigation, and they have been used for quantitative structural measurements (e.g., for counting the number of changed methods), in contrast to our approach, which provides declarative graph patterns, which are used to drive and initiate the transformation of change descriptions. Additionally, the Goose tool in FAMOOS uses Prolog rules to search for violations of certain design guidelines. Prolog rules show similarity to our graph patterns in their structure, however, our approach requires no conversion of underlying models, in contrast to Goose, which can operate only on Prolog facts that have to be extracted in advance from FAMIX models.

[38] applies graph transformation for metamodel evolution in domain-specific languages. In this approach, GT rules evolve models in a metamodel compliance preserving way. More specifically, they describe the changes themselves inside a single modeling domain, but not the transformation of changes between different domains as in our solution. Moreover, [38] lacks live transformation support.

Calculation of model correspondence and differences. Frameworks such as AMW [39] allow discovering and

representing hierarchical correspondences and differences between models. The approach presented by [40] operates on a hierarchical traceability model to maintain high- and low-level correspondence between models, and outlines a mechanism for incrementally and efficiently maintaining traceability relationships. This technology can also be used to create transformations that incrementally propagate changes to target models. The key challenge of these approaches is establishing this correspondence, using heuristics if necessary.

Calculating differences (deltas) of models has been widely studied due to its important role in the process of model editing, which requires undo and redo operations to be supported. In [41], metamodel independent algorithms are proposed for calculating directed (backward and forward) deltas, which can later be merged with the initial model to produce the resulting model. Unfortunately, the algorithms proposed by [41] for difference and merge calculation may only operate on a single model, and they are not specified by model transformation. In [42], a metamodel independent approach is presented for visualizing backward and forward directed deltas between consecutive versions of models. Differences (i.e., change history models) have a model-based representation (similarly to [43]), and calculations are driven by (higher order) transformations in both [42] and our approach. However, in contrast to [42] and [43], our current proposal is applicable even in an exogeneous transformation context to propagate change descriptions from source to target models.

Incremental synchronization for exogeneous model transformations. Incremental synchronization approaches already exist in model-to-model transformation context (e.g., [44]). One representative direction is to use triple graph grammars [45] for maintaining the consistency of source and target models in a rule-based manner. The proposal of [46] relies on various heuristics of the correspondence structure. Dependencies between correspondence nodes are stored explicitly, which drives the incremental engine to undo an applied transformation rule in case of inconsistencies. Other triple graph grammar approaches for model synchronization (e.g., [47]) do not address incrementality. Triple graph grammar techniques are also used in [48] for tool integration based on UML models. The aim of the approach is to provide support for change synchronization between various languages in several development phases. Based on an integration algorithm, the system merges changed models on user request. In this sense, contrarily to our solution, none of these approaches performs live transformation, but such a technique could possibly be easily integrated into these tools as well.

The approach of [49] shows the largest similarity to our proposal as both (i) focus on change propagation in the context of model-to-model transformation, (ii) describe changes in a model-based and metamodel independent way, and (iii) use rule-driven algorithms for propagating changes of source models to the target side. In the proposal of [49] target model must be materialized and they can also be manually modified, which results in a complex merge operation to be performed to get the derived model. In contrast, our algorithms can be used on non-materialized target models, and the derived models are computed automatically on the target side.

10 Conclusion and Future Work

In the paper, we discussed change-driven transformations, which is a novel class of model transformations aiming to process or derive changes as their input or output. We presented a novel language for specifying change-driven transformations extending a well-established graph transformation language. We also outlined how the same language can be executed in different change scenarios by adapting incremental graph pattern matching engines.

Note that the scenarios discussed in the paper are important but not the only potential application fields of change-driven transformations. In addition to the model synchronization and evolutionary constraints case studies, we have an extended set of applications using CDTs as underlying formalism for the semantic back-annotation of model transformations driven by execution traces [50].

A primary focus for future work is to elaborate how change-driven transformations can serve as an intermediate language for the efficient execution of existing model transformation used for bidirectional model synchronization languages (like TGGs or QVT relations, and maybe also ATL). Moreover, it is also worth investigating if our change patterns language can be extended to the action part to allow the implicit derivation of change models as output of change-driven rules. Furthermore, we also intend to investigate the correctness and consistency checking of change-driven transformations. Finally, we also aim at using change models for model merging.

References

1. Ráth, I., Varró, G., Varró, D.: Change-driven model transformations. In: Proc. of MODELS'09, ACM/IEEE 12th International Conference On Model Driven Engineering Languages And Systems. (2009)
2. Hearnden, D., Lawley, M., Raymond, K.: Incremental model transformation for the evolution of model-driven systems. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems. Volume 4199 of LNCS., Genova, Italy, Springer (October 2006) 321–335

3. Ráth, I., Bergmann, G., Ökrös, A., Varró, D.: Live model transformations driven by incremental pattern matching. In: Theory and Practice of Model Transformations. Volume 5063/2008 of LNCS., Springer Berlin / Heidelberg (2008) 107–121
4. Köth, O., Minas, M.: Generating diagram editors providing free-hand editing as well as syntax-directed editing. In Ehrig, H., Taentzer, G., eds.: GRATRA 2000 Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems, Berlin, Germany (March 25–27 2000) 32–39
5. de Lara, J., Vangheluwe, H.: AToM^3: A Tool for Multi-formalism and Meta-modelling. In Kutsche, R.D., Weber, H., eds.: 5th International Conference, FASE 2002: Fundamental Approaches to Software Engineering, Grenoble, France, April 8-12, 2002, Proceedings. Volume 2306 of LNCS., Springer (2002) 174–188
6. Mens, T.: A state-of-the-art survey on software merging. IEEE Transactions on Software Engineering **28** (2002) 449–462
7. Ráth, I., Ökrös, A., Varró, D.: Synchronization of abstract and concrete syntax in domain-specific modeling languages. Journal of Software and Systems Modeling (2009) Accepted.
8. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook on Graph Grammars and Computing by Graph Transformation. Volume 2: Applications, Languages and Tools. World Scientific (1999)
9. Rensink, A.: Representing first-order logic using graphs. In Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G., eds.: Proc. 2nd International Conference on Graph Transformation (ICGT 2004), Rome, Italy. Volume 3256 of LNCS., Springer (2004) 319–335
10. Orejas, F., Ehrig, H., Prange, U.: A logic of graph constraints. In: FASE'08/ETAPS'08: Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering, Berlin, Heidelberg, Springer-Verlag (2008) 179–198
11. Varró, D., Balogh, A.: The Model Transformation Language of the VIATRA2 Framework. Science of Computer Programming **68**(3) (October 2007) 214–234
12. Dittrich, K.R., Gatziu, S., Geppert, A.: The active database management system manifesto: A rulebase of ADBMS features. In Sellis, T., ed.: Proc. 2nd International Workshop on Rules in Database Systems. Volume 985 of LNCS., Springer (September 1995) 1–17
13. Ráth, I., Vágó, D., Varró, D.: 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)
14. Egyed, A.: Instant consistency checking for the UML. In: Proceedings of the 28th international conference on Software engineering, New York, NY, USA, ACM (2006) 381–390
15. Tratt, L.: A change propagating model transformation language. Journal of Object Technology **7**(3) (March-April 2008) 107–126
16. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. International Journal of Web Information Systems (IJWIS) **5**(3) (2009) 271–304
17. Egyed, A., Letier, E., Finkelstein, A.: Generating and evaluating choices for fixing inconsistencies in UML design models. In: ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, Washington, DC, USA, IEEE Computer Society (2008) 99–108
18. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence **19**(1) (September 1982) 17–37
19. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the VIATRA transformation system. In: GRaMoT'08, 3rd International Workshop on Graph and Model Transformation, 30th International Conference on Software Engineering (2008)
20. Bergmann, G., Ákos Horváth, Ráth, I., Varró, D.: A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation. In: ICGT2008, The 4th International Conference on Graph Transformation
21. Horváth, Á., Varró, G., Varró, D.: Generic search plans for matching advanced graph patterns. In: Proc. of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007), Braga, Portugal, Electornic Communications of the EASST (March 31- Apr. 1 2007) 57–68
22. Guerra, E., de Lara, J.: Event-driven grammars: Towards the integration of meta-modelling and graph transformation. In Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G., eds.: Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings. Volume 3256 of Lecture Notes in Computer Science., Springer (2004) 54–69
23. Guerra, E., de Lara, J.: Event-driven grammars: Relating abstract and concrete levels of visual languages. Software and Systems Modeling **6**(3) (2007) 317–347
24. Mouratidis, H., Giorgini, P., Manson, G., Philp, I.: A natural extension of tropos methodology for modelling security. In: Agent Oriented Methodologies Workshop. Object Oriented Programming, Systems, Languages (OOPSLA), SEATTLE-USA, ACM (2002)
25. Tun, T.T., Yu, Y., Haley, C., Nuseibeh, B.: Model-based argument analysis for evolving security requirements. Secure System Integration and Reliability Improvement **0** (2010) 88–97
26. Haley, C.B., Laney, R.C., Nuseibeh, B., Hall, W.: Validating security requirements using structured toulmin-style argumentation (2005)
27. Yu, E.S.K.: Towards modelling and reasoning support for early-phase requirements engineering. In: Proceedings of the 3rd IEEE Int. Symp. on Requirements Engineering. (1997) 226–235
28. Shanahan, M.: Artificial intelligence today. Springer-Verlag, Berlin, Heidelberg (1999) 409–430
29. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book. Prentice Hall (2001)
30. Seiriö, M., Berndtsson, M.: Design and implementation of an ECA rule markup language. In Adi, A., Stoutenburg, S., Tabet, S., eds.: Proc. of the 1st International Conference on Rules and Rule Markup Languages for the Semantic Web. Volume 3791 of LNCS., Galway, Ireland, Springer (October 2005) 98–112
31. OMG: MOF Query View Transformation Specification. Object Management Group. (April 2008)
32. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. Software & Systems Modeling **9**(1) (January 2010) 7–20

33. Grundy, J., Hosking, J., Mugridge, W.B.: Inconsistency management for multiple-view software development environments. IEEE Transactions on Software Engineering **24**(11) (1998) 960–981
34. Olsson, T., Grundy, J.: Supporting traceability and inconsistency management between software artefacts. In Hamza, M.H., ed.: Proceedings of the 2002 IASTED International Conference on Software Engineering and Applications, Cambridge, USA (November 2002)
35. Ducasse, S., Demeyer, S.: The FAMOOS Object-Oriented Reengineering Handbook. (October 1999) `http://scg.unibe.ch/archive/famoos/handbook/4handbook.pdf`.
36. Tichelaar, S., Ducasse, S., Demeyer, S.: FAMIX and XMI. In: Proceedings of the Seventh Working Conference on Reverse Engineering, Brisbane, Australia, IEEE Computer Society (November 2000) 296–298
37. Gîrba, T., Ducasse, S.: Modeling history to analyze software evolution. Journal of Software Maintenance and Evolution: Research and Practice **18**(3) (May 2006) 207–236
38. Narayanan, A., Levendovszky, T., Balasubramanian, D., Karsai, G.: Automatic domain model migration to manage metamodel evolution. In Schürr, A., Selic, B., eds.: Proc. of the 12th International Conference on Model Driven Engineering Languages and Systems. Volume 5795 of Lecture Notes in Computer Science., Denver, Colorado, USA, Springer (October 2009) 706–711
39. Fabro, M.D.D., Bezivin, J., Jouault, F., Breton, E., Gueltas, G.: AMW: a generic model weaver. In: Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05). (2005)
40. Seibel, A., Neumann, S., Giese, H.: Dynamic hierarchical mega models: Comprehensive traceability and its efficient maintenance. Software and System Modeling **009**(s10270) (0 2009)
41. Alanen, M., Porres, I.: Difference and union of models. In Stevens, P., Whittle, J., Booch, G., eds.: Proc. of the 6th International Conference on the Unified Modeling Language, Modeling Languages and Applications (UML 2003). Volume 2863 of LNCS., San Francisco, California, USA, Springer (October 2003) 2–17
42. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: A metamodel independent approach to difference representation. Journal of Object Technology **6**(9) (October 2007) 165–185
43. Gruschko, B., Kolovos, D.S., Paige, R.F.: Towards synchronizing models with evolving metamodels. In: Proc. Int. Workshop on Model-Driven Software Evolution held with the ECSMR. (2007)
44. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. (2007) 164–173
45. Schürr, A.: Specification of graph translators with triple graph grammars. Technical report, RWTH Aachen, Fachgruppe Informatik, Germany (1994)
46. Giese, H., Wagner, R.: Incremental model synchronization with triple graph grammars. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Proc. of 9th International Conference on Model Driven Engineering Languages and Systems, (MoDELS 2006). Volume 4199 of LNCS., Springer (2006) 543–557
47. Klar, F., Königs, A., Schürr, A.: Model transformation in the large. In: ESEC-FSE '07: Proceedings of European Software Engineering Conference, New York, NY, USA, ACM (2007) 285–294
48. Becker, S.M., Haase, T., Westfechtel, B.: Model-based aposteriori integration of engineering tools for incremental development processes. Software and Systems Modeling **4**(2) (May 2005) 123–140
49. Jimenez, A.M.: Change propagation in the MDA: A model merging approach. Master's thesis, The University of Queensland (June 2005)
50. Hegedüs, Á., Ráth, I., Varró, D.: Back-annotation of Simulation Traces with Change-Driven Model Transformations. In: Proceedings of the Eigth International Conference on Software Engineering and Formal Methods. (2010) Accepted.
51. Rensink, A.: Representing first-order logic using graphs. In: International Conference on Graph Transformations (ICGT), volume 3256 of Lecture Notes in Computer Science, Springer (2004) 319–335

A Basics Concepts of Graph Transformation

A.1 Graph Patterns

The central concept of GT is the notion of graph patterns, which are basically small graphs. *Pattern matching* is the (computationally complex) process of identifying subgraphs in the graph model G that correspond to the pattern. More formally, a pattern $P\langle V, C\rangle$ contains a set V of *pattern variables* with some graph constraints C attached to them; a pattern *match* is a mapping $m : V \to G$ of all pattern variables to model elements so that the image of the variables observes all constraints. The most important constraints are entity constraints stating that a variable is a node of a certain type, and relation constraints stating that a variable is an edge of a certain type, connecting two given variables.

Definition 6 (Graph Model) *A graph model over a type system Type is a structure* $G = \langle Ent, Rel, src, trg, typ\rangle$ *where Ent is a set of entities (graph nodes), Rel is a set of relations (graph edges);* $src, trg : Rel \to Ent$ *maps the relations to their source and target entities, respectively; and the typing of elements is* $typ : GE \to Type$ *where GE is an abbreviation for the set of graph elements* $Ent \bigcup Rel$.

Our graph model assumes that each entity and relation takes its type from a type system which is simplified here to a set of predefined types. The notion of type compatibility is beyond the scope of this paper. Various other model features such as containment are omitted here for brevity. It is also possible to represent hypergraphs, where there are more than two incidence maps instead of just src and trg.

Definition 7 (Graph Pattern) *A graph pattern* $P = \langle V, C \rangle$ *over a type system* $Type$ *contains a set of* pattern variables V*, and a set of graph constraints* $C = C^{ent} \bigcup C^{rel}$ *attached to them.* V *is partitioned into entity variables* V^{ent} *and relation variables* V^{rel}*. Entity constraints* $C^{ent} \subseteq V^{ent} \times Type$ *state that a variable is a node of a certain type. Relation constraints* $C^{rel} \subseteq V \times V^{rel} \times V \times Type$ *state that a variable is an edge of a certain type, connecting two given variables representing the source and the target of the edge. To identify the variables and constraints of a specific pattern* P*, we use* $V(P)$ *and* $C(P)$*, respectively.*

The pattern language [11] of the VIATRA2 tool also permits additional constraints such as containment, equality and inequality, or *pattern composition*, which are not detailed here.

Definition 8 (Graph Pattern Match) *A* substitution $s : P \to G$ *of a graph pattern* $P = \langle V, C \rangle$ *in a graph model* $G = \langle Ent, Rel, src, trg, typ \rangle$ *over a type system* $Type$ *is a set of variable assignments* $asgn \in V \times GE$*, one for each variable* $v \in V$*. Let* $s(v) \in GE$ *denote the model element assigned by* s *to the variable* $v \in V$*.*

A substitution satisfies an entity constraint $c = \langle v, t \rangle \in C^{ent}$ *iff* $typ(s(v))$ *is compatible with* t*. A substitution satisfies a relation constraint* $c = \langle a, v, b, t \rangle \in C^{rel}$ *iff* $src(s(v)) = s(a)$ *and* $trg(s(v)) = s(b)$ *and* $typ(s(v))$ *is compatible with* t*.*

A match $m : P \to G$ *is a substitution that satisfies all constraints* $c \in C$ *of* P*, which will be denoted by* $G, m \models P$*.*

Remark: from now on, we assume that a single type system $Type$ is given, and will not include it in each further definition.

A *negative application condition* (NAC, indicated by the *neg* keyword) prescribes contextual conditions that, if satisfiable, invalidate a match of the pattern.

Definition 9 (Graph Pattern with Negative Application Condition) *A pattern with NAC is* $PN = \langle P, N^* \rangle$ *where* $P = \langle V, C \rangle$ *is a (positive) graph pattern, and* N^* *is a set of negative application conditions* $N_i = \langle V_i, C_i \rangle$*, each being a well-formed graph pattern, such that* $P \subseteq N_i$ *meaning that* $V \subseteq V_i$ *and* $C \subseteq C_i$*.*

Commonly, only the *subpattern* $\hat{N}_i = N_i \setminus P$ is explicitly indicated and depicted in figures and code extracts, which is defined as $\hat{N}_i = \langle \hat{V}_i, \hat{C}_i \rangle$, where $\hat{C}_i = C_i \setminus C$ and $\hat{V}_i \subseteq V_i$ is the set of variables involved in $\hat{C}_i$.

Definition 10 (Match of Graph Pattern with NAC) *A match* $m : PN \to G$ *of* $PN = \langle P, N^* \rangle$ *in graph model* G *is a match of the positive pattern* $G, m \models P$*, where there is no* $N_i \in N^*$ *and match* $m_i : N_i \to G$ *such that* $m \subseteq m_i$ *(meaning that* $m_i(v) = m(v)$ *for all* v *variables of* P*).*

Some systems even permit NACs to have NACs of their own; if there is no limit on the number of negations that can be nested within each other, graph patterns (without attribute constraints) become expressively equivalent to first order formulae over the predicates describing the graph model [51].

A.2 Attributed Systems

Definition 11 (Attributed Graph Model) *An attributed graph over a domain* Dom *of attribute values is a graph model* $G = \langle Ent, Rel, src, trg, typ \rangle$ *where* Ent *is partitioned into* $Ent_{Mod} \cup Dom$*.* Dom *is the immutable and infinite set of all attribute values (integers, strings, etc.), having special attribute types.* Ent_{Mod} *is called the set of model entities.* Rel *is partitioned into three sets.* $Rel_{Mod} = \{\ r \in Rel | src(r), trg(r) \in Ent_{Mod}\ \}$ *is the set of model relations.* $Rel_{Dom} = \{\ r \in Rel | src(r), trg(r) \in Dom\ \}$ *is the immutable and infinite set of relations between domain values, having types such as ordering, substring etc. Operators (multiplication, concatenation, etc.) can also be represented in hypergraphs, or with auxiliary nodes.* $Rel_{Val} = Rel \setminus (Rel_{Mod} \bigcup Rel_{Dom})$ *is the set of value assignment relations that assign attribute values to model entities; their types correspond to different attribute names. Assignment relations point from the model entity to the domain value, and have a many-to-one multiplicity for each attribute name.*

Obviously, implementations will only manifest a subset of Dom, such as $trg(Rel_{Val})$. Similarly Rel_{Dom} arcs are typically not stored, but it is assumed that the existence of a domain relation with given nodes and type is easily decidable. Some types of Rel_{Dom} relations might be *function-like*, meaning that one (or more) of their incident nodes can be efficiently deduced from the others (e.g. the value of a product is derivable from the value of its factors).

The fact that $r \in Rel_{Val}(G)$ where $src(r) = obj$, $trg(r) = val$ and $typ(r) = attr$ can also be denoted as $G \models obj.attr = val$.

Definition 12 (Graph Pattern with Attributes) *In attributed systems,* V^{ent} *of a graph pattern is further partitioned into model and domain entity variables (*V^{ent}_{Mod} *and* V^{ent}_{Dom}*), Their corresponding constraints are* $C^{ent}_{Mod} \subseteq V^{ent}_{Mod} \times Type$ *expressing that a variable represents a model node of a certain type and* $C^{ent}_{Dom} \subseteq V^{ent}_{Dom} \times Type$ *asserting attribute types such as integer. Similarly* V^{rel} *is partitioned into three sets: model relations* V^{rel}_{Mod} *between model elements, value assignments* V^{rel}_{Val} *that connect model elements to their attribute values, and domain relations* V^{rel}_{Dom} *between attribute values, each sort with a unique type of relation constraint.* $C^{rel}_{Mod} \subseteq V_{Mod} \times V^{rel}_{Mod} \times V_{Mod} \times Type$ *expresses that the variable represent a model edge of a certain type.* $C^{rel}_{Val} \subseteq V_{Mod} \times V^{rel}_{Val} \times V_{Dom} \times Type$ *means that a certain value assignment, associated with an attribute name (taken from*

Type), links a model variable to a domain variable as its attribute value. $C^{rel}_{Dom} \subseteq V_{Dom} \times V^{rel}_{Dom} \times V_{Dom} \times Type$ *basically means an attribute constraint check among the variables corresponding to attribute values.*

As graph pattern matchers are not required to be equation solvers or constraint engines, and the entirety of *Dom* cannot be manifested, some systems require attributed patterns to be formalized in such a way that matches can actually be computed from the model. A graph pattern is *matchable* iff each domain entity variable $v \in V^{ent}_{Dom}$ is bound either as the attribute value of a model element variable, or as the result of functions (e.g. addition) on bound variables; and its NACs are also matchable.

A.3 Graph Transformation

The mathematical formalism of Graph Transformation (GT) [8] provides a high-level rule and pattern-based manipulation language for graph models.

Definition 13 (Graph Transformation Rule) *Graph transformation rules* $GTR = \langle LHS, RHS \rangle$ *are specified by two graph patterns: a* precondition *(or left-hand side) pattern (with NAC)* LHS *defining the applicability of the rule, and a* postcondition *(or right-hand side) positive pattern* RHS *which declaratively specifies the result model after rule application. The variable sets of* LHS *and* RHS *are allowed to intersect.*

When the rule is applied on a match of the LHS, elements that are present only in (the image of) the LHS are deleted, elements that are present only in the RHS are created, and other model elements remain unchanged.

Definition 14 (Application of Graph Transformation Rule) *A graph transformation rule* $GTR = \langle LHS, RHS \rangle$ *can be applied on a match* $m : LHS \to G$ *in a graph model* G *as follows:*

- ***Deletion.*** *For each* $v \in V_{LHS} \setminus V_{RHS}$*, the model element* $m(v)$ *is deleted.*
- ***Creation.*** *For each* $v \in V_{RHS} \setminus V_{LHS}$*, a new model element is created and assigned to* v *in a way that* RHS *becomes satisfied (i.e. the new element is created in a type-conforming and structurally consistent way).*

If LHS *has no matches in* G*, then* GTR *is not applicable.*

For conciseness, certain effects (varying within GT tools) such as deletion of dangling edges, edge redirecting, etc. were omitted from the formal definition. Note that this core formalism of GT rules does not define how to react to changes, which is the subject of Section 4. In attributed graphs, domain entities and relations obviously cannot be modified, but the creation or deletion of value assignment edges is interpreted as updating the value of attributes. Once again, this is a simplified overview and is by far not universal to GT systems.

Example. As an illustration, Figure 24 shows a GT rule that is applied on invocation nodes that have not yet been mapped to a jPDL node (see LHS, same as Figure 6(a)), and creates the jPDL counterpart when applied (see RHS). Similarly, a GT rule responsible for mapping transitions is demonstrated by Figure 25, as well as Listing 4 (reusing the previously shown pattern as LHS). These two rules together transform the sequence of invocations in the workflow. Further rules, omitted in this paper, are required to deal with the attributes of invocations, and transform the rest of workflow elements.

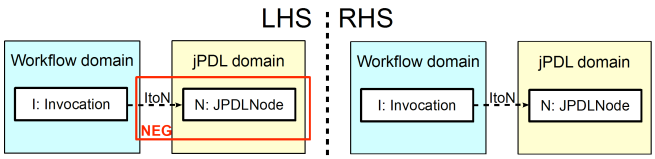


Fig. 24 Example GT Rule for mapping invocations

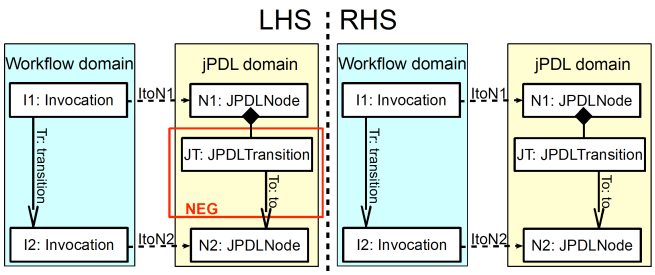


Fig. 25 Example GT Rule for mapping transitions

```
gtrule mapTrfromWFtoJPDL(I1, N1, I2, N2, JT) =
{
 precondition find noJPDLTr(I1, N1, I2, N2);
 postcondition pattern mapped(I1, N1, I2, N2, JT) =
 {
  Invocation(I1);
  traceability(ItoN1, I1, N1);
  JPDLNode(N1);
  JPDLTransition(JT) in N1;
  JPDLTransition.to(To, JT, N2);
  JPDLNode(N2);
  traceability(ItoN2, I2, N2);
  Invocation(I2);
  Invocation.transition(Tr, I1, I2);
 }
}
```

Listing 4 Example Graph Transformation Rule