Softw Syst Model (2010) 9:493–528
DOI 10.1007/s10270-009-0146-z

THEME SECTION

Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance

Andreas Seibel · Stefan Neumann · Holger Giese

Received: 21 January 2009 / Revised: 16 October 2009 / Accepted: 6 November 2009 / Published online: 27 December 2009

Abstract In the world of model-driven engineering (MDE) support for traceability and maintenance of traceability information is essential. On the one hand, classical traceability approaches for MDE address this need by supporting automated creation of traceability information on the model element level. On the other hand, global model management approaches manually capture traceability information on the model level. However, there is currently no approach that supports comprehensive traceability, comprising traceability information on both levels, and efficient maintenance of traceability information, which requires a high-degree of automation and scalability. In this article, we present a comprehensive traceability approach that combines classical traceability approaches for MDE and global model management in form of dynamic hierarchical mega models. We further integrate efficient maintenance of traceability information based on top of dynamic hierarchical mega models. The proposed approach is further outlined by using an industrial case study and by presenting an implementation of the concepts in form of a prototype.

Keywords Model-driven engineering · Traceability maintenance · Global model management · Mega model

Communicated by Prof. Richard Paige.

A. Seibel (✉) · S. Neumann · H. Giese
Hasso Plattner Institute, University of Potsdam,
Potsdam, Germany
e-mail: andreas.seibel@hpi.uni-potsdam.de

S. Neumann
e-mail: stefan.neumann@hpi.uni-potsdam.de

H. Giese
e-mail: holger.giese@hpi.uni-potsdam.de

1 Introduction

Due to the advances in hardware and network technologies, the complexity of software systems is rapidly increasing and developing such software systems by only using current code-centric implementation technologies has become impractical [20]. Model-driven engineering (MDE) addresses the complexity, which is inherent in today's software system development, by using a set of modeling artifacts representing parts of a system at different levels of abstraction and within different views. These modeling artifacts can be different types of models expressing specialized modeling languages or part of them. MDE further introduces the application of model-based operations to systematically transform modeling artifacts into concrete implementations. In MDE, modeling artifacts do not exist in isolation but have inherent dependencies between each other. Being unaware of these dependencies potentially lead to inconsistencies when it comes to changes to some modeling artifacts which does not lead to appropriate changes in dependent modeling artifacts. Neglecting these dependencies can lead to a decreased system quality and increased development time and costs, which endangers the success of developing software systems [3,6,11,40]. Thus, traceability approaches are essentially making these dependencies visible by means of *traceability links*.

A common and long-standing definition of *traceability* is given in the IEEE Standard Glossary of Software Engineering Terminology: "*...the degree to which a relationship having a predecessor-successor or master-subordinate relationship to one another...*" [28]. This definition of traceability is, however, too restricted for the MDE context since it requires that modeling artifacts are in a chronological relationship [34]. In MDE, models of different views are often employed in parallel rather than in sequence. A much broader

definition of traceability is given by Aizenbud-Reshef et al. [1]: "*...any relationship that exists between artifacts involved in the software engineering life-cycle...*", which is therefore used throughout this article.

A common foundation of traceability approaches are *traceability models*, which capture traceability links between modeling artifacts. In classical traceability approaches for MDE, *low-level traceability models* are applied, which consist of *low-level traceability links* between model elements only [2,3,5,16,30,32,42,44]. In contrast, early approaches addressing global model management primarily focus on *high-level traceability models*[1] that contain *high-level traceability links* between models only.

A comprehensive traceability approach has to combine high-level as well as low-level traceability models into a *combined traceability model* and not treat them as disjoint approaches. Thus, a combined traceability model should capture hierarchy information that denotes the affiliation of model elements and models as well as hierarchical dependencies between traceability links. These dependencies are essential when it comes to automated maintenance of *traceability information*[2] because low-level traceability links can depend on high-level traceability links and contrary.

Due to the increasing complexity of software systems, and therefore software models, maintenance of traceability information needs to be automated whenever possible to make traceability approaches still attractive to end-users. Classical traceability approaches for MDE rather focus on automated creation of low-level traceability links [3,5,16,30,42]. In the context of global model management, the approaches [4,9,10,18] provide no support for automatic creation or deletion of high-level traceability links. In [7] it is shown how to automatically create low-level traceability models as a whole based on already existing high-level traceability links in mega models. The approaches in [38,39] are pretty similar but they use another technology. They automatically create low-level traceability models within the context of macro models. However, none of the considered approaches can automatically create *and* delete *individual* low-level *and* high-level traceability information in a combined traceability model.

Furthermore, modeling artifacts are not static but rather subject to continuous changes, which may render the intention of existing traceability links suspect. In literature, the semantic of traceability links is usually interpreted as their intention (what is the intended use) of the traceability links [2, 31]. The context of traceability links are modeling artifacts, which are related to traceability links. Thus, an approach is required to efficiently maintain suspect traceability links. Nevertheless, completely deleting and subsequently re-creating traceability information as soon as the intention of traceability links become suspect may lead to performance issues. Thus, to efficiently maintain traceability information, we require further maintenance support to avoid completely deleting and subsequently re-creating traceability information as soon as the intention of traceability links become suspect, which enables, to some extent, *incremental* maintenance of traceability information where needed.

Thus, efficient maintenance of traceability information should also comprise (re-)establishing the *validity* of traceability links in a separate maintenance process, which goes beyond initial creation. The validity of traceability links is defined as the satisfaction of the intention of traceability links. So, to (re-)establish the validity of traceability links, some labor is required that is responsible for re-satisfying the intention of traceability links. The labor can be any complex operation or even manual tasks. Furthermore, separating maintenance of traceability information into automatic creation/deletion and subsequent (re-)establishment of the validity of traceability links, in cases of contextual changes, is even beneficial for reasons other than performance. In certain cases, the process of (re-)establishing the validity is not required at certain points in time or can even be foolish [19,41]. In these cases, the existence of traceability links is preferred over knowing their validity instantaneously. Thus, (re-)establishing the validity of traceability links should be delayed independently from the initial creation. Furthermore, because reasoning about the existence of traceability links is potentially more *lightweight*, it could be applied in shorter cycles than (re-)establishing their validity, which may imply more *heavyweight* operations or even manual tasks. To the best of our knowledge, there is no approach that solves all the aforementioned issues by providing comprehensive traceability and further providing efficient maintenance of traceability information.

In this article, we provide a comprehensive traceability approach by means of a combination of high-level traceability models (mega models) and low-level traceability models. It further defines hierarchical dependencies between high-level and low-level modeling artifacts and between traceability links at different levels to glue both traceability models into a combined traceability model. Thus, our approach is an integration of global model management and classical traceability approaches in the context of MDE. Additionally, our approach provides efficient maintenance of traceability information based on our combined traceability model. The maintenance process is split into two processes called *localization* and *execution*. Localization is the process of comprising the creation and deletion of traceability information whereas execution is the process of (re-)establishing

[1] These models are not called traceability models but, e.g., mega models in [4,9,10,18] and macro models in [38,39].

[2] Sometimes traceability links alone are not sufficient to specify traceability. Thus, additional traceability-related information is defined under the umbrella of traceability information.

the validity of already existing traceability links. The overall approach is further called *dynamic hierarchical mega models*.

Summarized, the novel contributions of this article are:

1. a comprehensive traceability approach based on dynamic hierarchical mega models, which is a combined traceability model;
2. efficient maintenance of traceability information by introducing a separated localization and execution process based on a dynamic hierarchical mega model;
   - localization uses endogenous model transformation rules to automatically create and delete traceability information, which further guarantees termination and uniqueness;
   - execution is used to subsequently (re-)establish the validity of already existing traceability links;
3. a proof of concept in the form of a prototype implementation of the approach.

The rest of the article is organized as follows: we start by introducing the approach in more detail in Sect. 2 by presenting a case study, outlining the detailed challenges, and giving a first conceptual overview about the proposed approach of dynamic hierarchical mega models. The syntax of our combined traceability model is outlined in Sect. 3 by means of a meta model of dynamic hierarchical mega models and further examples from our case study. The localization process is described in Sect. 4 explaining the technology that is used to automatically create/delete traceability information and an appropriate algorithm for efficient application. The execution process follows in Sect. 5, which shows how we derive a proper ordering of operations to efficiently (re-)establish the validity of traceability links. The efforts necessary to setup an MDE environment with our approach, the supported usage scenarios, and a prototype demonstrating the feasibility of the presented concepts, are outlined in Sect. 6. An evaluation of the prototype and the possible usage scenarios follows in Sect. 7. Finally, we discuss related work in Sect. 8 and close the article with a final conclusion and an outlook on planned future work.

2 Approach

To introduce our approach, we start by introducing our industrial case study which is used throughout this article to motivate the underlying needs and explain our approach. We then highlight the open challenges for our approach and motivate them by examples taken from the case study. Finally, we give a conceptual overview of our approach outlining how the raised open challenges are addressed.

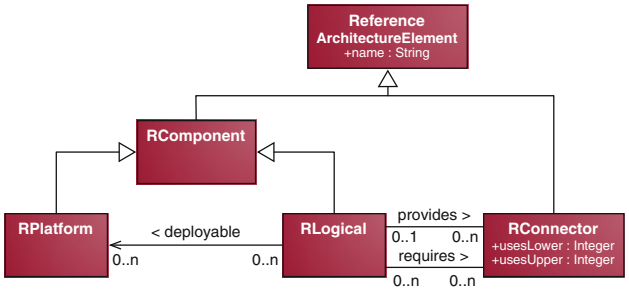

Fig. 1 RA meta model

2.1 Case study

In a research project with CA Labs,[3] we considered model-based support for the development of *software product configurations and deployments*. Therefore, we developed three domain-specific languages (DSL), which are presented in a simplified manner in the following. Software products and their configuration variability's are defined in Reference Architectures (RAs), which are developed by in-house R&Ds[4] of CA Inc. Configuration and deployment decisions are specified in Solution Architectures (SAs), which are tailored in cooperation with customers to reflect customers' needs. The products that are configured in an SA are deployed on physical machines, which are specified in separated IT Infrastructures (ITIs). An ITI reflects the concrete physical infrastructure of a customer requesting an SA. In the end, the SA is a solution that provides a certain service required by the customer.

2.1.1 Reference architectures

The RA has two possible applications. On the one hand, it is used for developing a software product. Thus, it describes a high-level architecture, which is made of components at a high-level of abstraction. On the other hand, it is used to capture all valid configurations of a software product comprising structural issues as well as component attributions, which are important for the subsequent deployment of the software product. In this article, we use a severely reduced meta model of the RA, which is sufficient to explain the examples in the follow-up of this article. The meta model is shown in Fig. 1.

An RA is made of ReferenceArchitectureElements. A ReferenceArchitectureElement is the overall type and can be a component of type RComponent or a connector of type RConnector. RComponent is either refined by a platform of type RPlatform or a logical component of type RLogical. RLogical is a representation of a software product that is deployable to RPlatform components, which is specified by the deployable

[3] CA Labs is the research department of CA Inc.; http://www.ca.com.

[4] R&D denotes Research and Development.

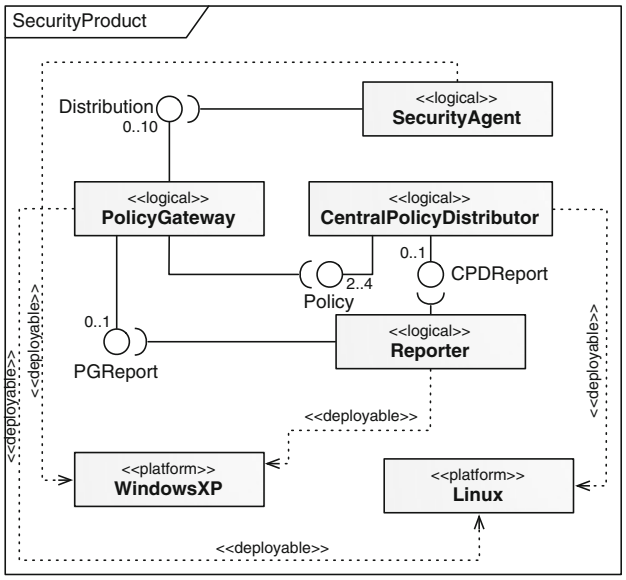

Fig. 2 Example instance of an RA using a concrete syntax

association. RConnector is a communication interface between logical components of type RComponent. An RComponent can provide connectors as well as require them, expressed by the associations provides and requires. An RConnector has two integer attributes usesLower and usesUpper, which represent lower and upper bounds for the number of RComponent instances that can use an instance of RConnector. As an example, we employ a simple RA that describes a security product by means of four logical components, which is shown in Fig. 2.[5]

The RA named SecurityProduct in Fig. 2 consists four logical components (⟨⟨logical⟩⟩), which are instances of RLogical, two platform components (⟨⟨platform⟩⟩), which are instances of RPlatform, and four connectors (denoted as circles), which are instances of RConnector.

SecurityAgents check for security breaches by validating if all security policies[6] are satisfied on the platform the SecurityAgents are deployed to. Security policies are specified within a CentralPolicyDistributor and further distributed to all indirectly connected SecurityAgents via the Policy connector. Therefore, all security policies are first distributed to all PolicyGateways using the Policy connector. PolicyGateways are used for workload purposes when distributing security policies to SecurityAgents. If the number of SecurityAgents increases, new PolicyGateways need to be connected to the CentralPolicyDistributor via the Policy connector to keep the number of SecurityAgents per PolicyGateway low. The

[5] All elements are instances of the according meta model but are shown by means of a concrete syntax to improve readability. However, all examples are further shown as abstract syntax because we want to highlight the integration into dynamic hierarchical mega models.

[6] Security policies are not explicitly modeled here.

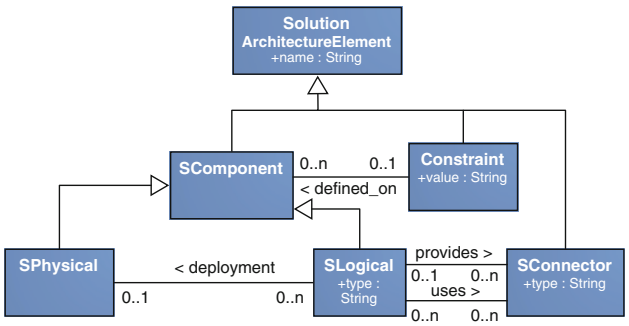

Fig. 3 SA meta model

Reporter is used to filter logging information, which are sent from all SecurityAgents to the PolicyGateway and subsequently to the CentralPolicyDistributor via PGReport and CPDReport connectors, respectively.

The RA shows even more information. The numbers next to connectors are the attributes usesLower and usesUpper the related RConnector class in the RA meta model. For example, the Policy connector requires at least two and at most four PolicyGateway components to be connected to it. The stereotyped connections are references of the type deployable between the classes RLogical and RPlatform. For example, the Reporter component is deployable to platforms of type WindowsXP.

2.1.2 Solution architectures

An SA is a configuration and deployment specification for specific software products that are defined by RAs. Since an RA is used as a blueprint for supporting the configuration and deployment of these software products, there are implicit dependencies between SAs and RAs because software products in SAs are concrete configurations of the related software products in RAs. A severely reduced version of the SA meta model is shown in Fig. 3.

An SA is made of SolutionArchitectureElements. A SolutionArchitectureElement is the overall type and can be a component of type SComponent, a connector of type SConnector or a constraint of type Constraint. RComponent is either refined by a representation of a physical machine of type SPhysical or a logical component of type SLogical. SLogical is a representation of a configured software product that should be deployed to an SPhysical, which is specified by the deployable association. An SPhysical is just a placeholder for a concrete physical machine that is available on-site at the customer. SLogical can further provide SConnectors, denoted by the provides association, and also use different SConnectors, denoted by the uses association. The attribute type is used to relate SLogicals and SConnectors to RLogicals and RConnectors, respectively. Thus, whenever the attribute type of an instance of an SLogical is equal to the attribute name of an instance of an RLogical is

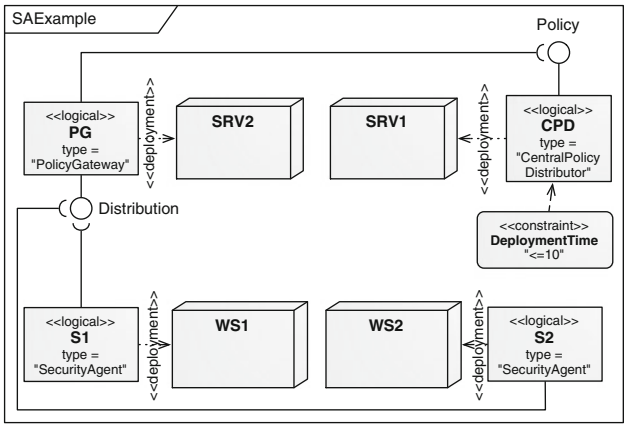

Fig. 4 Example of an SA using a concrete syntax

given, an implicit dependency exists. Constraints are defined on SComponents and have an attribute value that contains the constraint that needs to hold on referenced SComponents. Figure 4 shows a simple SA that conforms to the meta model of Fig. 3.

The SA named SAExample in Fig. 4 consists of four logical components (⟨⟨logical⟩⟩), which are instances of SLogical, four representations of physical machines (denoted as 3D rectangles), which are instances of SPhysical, two connectors (denoted as circles), which are instances of SConnector, and a constraint (⟨⟨constraint⟩⟩), which is an instance of Constraint.

This example shows a partial configuration of the software products defined in the RA of Fig. 2. It configures a PolicyGateway (PG), two SecurityAgents (S1 and S2), and a CentralPolicyDistributor (CPD). The CPD provides a connector called Policy that is used by PG, which further provides the connector Distribution that is used by S1 and S2. The SecurityAgents S1 and S2 should observe the physical machines called WS1 and WS2, respectively, where they should be deployed to. S1 and S2 are getting their security policies from PG through the connector Distribution. PG itself receives its security policies from CPD via the connector Policy. PG and CPD should be deployed on different physical machines called SRV2 and SRV1. The constraint DeploymentTime is related to CPD, which means that deploying all security policies to S1 and S2 must be conducted within 10 min.[7]

2.1.3 IT infrastructures

An ITI is a concrete physical infrastructure reflecting the physical IT of a customer. It is used as the target of SAs as a deployment target. For this purpose we have defined a meta model, which is shown in a severely reduced version in Fig. 5.

[7] The time unit is omitted in this figure.

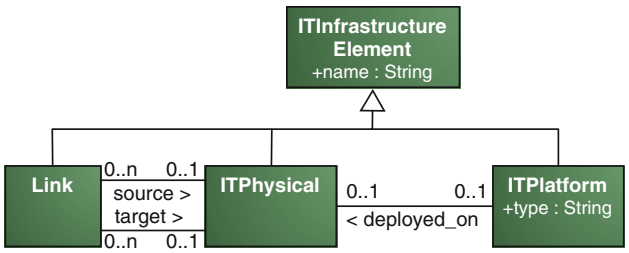

Fig. 5 ITI meta model

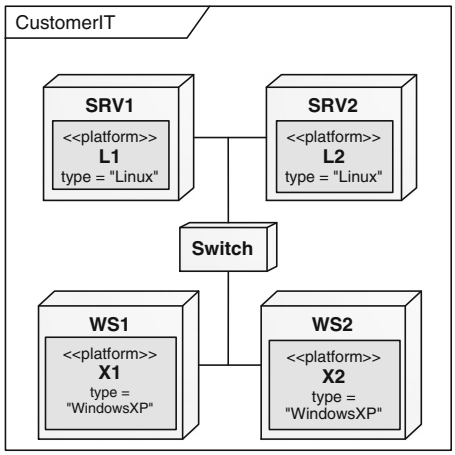

Fig. 6 Example instance of an ITI using a concrete syntax

An ITI is made of ITInfrastructureElements. An ITInfrastructureElement is the overall type and can be a platform of type ITPlatform, a concrete physical device of type ITPhysical or a link of type Link. An ITPhysical is interlinked with another ITPhysical element by means of a Link using the source and target associations. An ITPlatform can be deployed on an ITPhysical by means of the deployed_on association. ITPlatform has an attribute type, which is used to identify mapping between instances of RPlatform specified in RAs and instances of ITPlatform in ITIs. In this simplified case study, we assume that an instance of ITPhysical can only host a single instance of ITPlatform. Furthermore, the deployment of instances of SLogical of SAs is defined by individual mappings of instances of SPhysical and ITPhysical. For simplification, we define that such a mapping exists whenever the value of the attribute name of these components are equal. The deployment of instances of SLogical to instances of SPhysical define that a deployment *should* exist. Whenever a mapping of the related instance of SPhysical to an instance of ITIPhysical exists, a deployment exists because there is a target where it can be deployed to in the ITI. All instances of SLogical that are thus deployed to instances of ITPhysical are meant to be executed on instances of ITPlatfrom component that is deployed on the same instance of ITPhysical. Figure 6 shows a simple example of an ITI of a hypothetical customer.

The ITI named CustomerIT in Fig. 6 consists of five physical devices (denoted as 3D rectangles), which are instances of ITPhysical, and four platforms (⟨⟨platform⟩⟩), which are instances of ITPlatform. The links between physical machines are denoted as simple connections in between. The example shows a simple ITI with physical devices SRV1, SRV2, WS1, WS2 and Switch and platforms L1, L2, X1, X2 and the deployment of these platforms on physical devices. Thus, the hypothetical customer owns these five physical devices and the platforms they are deployed on.

2.2 Challenges

Our cooperation with CA Labs and literature studies revealed that several open challenges exist which are not sufficiently covered by existing proposals. These challenges, which led to the development of the presented approach, are discussed in detail in the following.

2.2.1 Comprehensive traceability

As shown in the introduction, a comprehensive traceability approach has to combine high-level and low-level traceability links in a combined traceability model and not treat them in disjoint approaches. However, just merge both approaches is not sufficient. The missing key is to also provide additional hierarchy information between models and model elements and high-level and low-level traceability links, which is required when it comes to automated maintenance.

This is a foundational prerequisite to many scenarios in our case study. In a specific scenario, e.g., checking consistency between an SA and an RA it is explicitly requested. Consistency between these models is important because RAs act as configuration blueprints for SAs. Considering the example SA and RA of Figs. 2 and 4, the connector Policy in the SA must be consistent with the connector Policy in the RA. Now, the challenge is to capture the request for checking consistency between the SA and the RA and further to reason about consistency based on the details of the SA and RA. Thus, we should use a high-level traceability link between the SA and the RA and a low-level traceability link between the connectors Policy. Additionally, one connector Policy must be a model element of the SA and the other one of the RA. Furthermore, the low-level traceability link depends on the high-level traceability link (see Sect. 3.2.1). The dependency between traceability links may also hold the other way around. Thus, high-level traceability links may depend on a low-level traceability links (see Sect. 3.2.2). In both cases, we need knowledge about the containment dependency of model elements and models otherwise we could not identify traceability links in such dependencies.

In addition, traceability links can have different interpretations. We show two identified interpretations we have to support. Traceability links can be interpreted as something that reveals some information between modeling artifacts. Thus, they only show that some intention holds between depending modeling artifacts. We can also interpret traceability links as the instantiation of some arbitrary operations or manual tasks that require information from the context of traceability links to produce additional information. The information that is produced may range from modifying existing information in the context of the traceability links to creating completely new information that has to be added to the context additionally. The latter interpretation of traceability links, lead to another dependency between traceability links that is defined by overlapping contexts of several traceability links (see Sect. 5.5). This cannot be neglected when it comes to executing the operations or tasks related to the traceability links because the context must be up-to-date, which may depend on executing another operation or task first. We observed the latter interpretation of traceability links and this kind of dependency already in mega models (cf. [4,9,10,18]).

2.2.2 Coexistence of manual and automated traceability maintenance

When considering the degree of automation that can be reached when maintaining traceability information we need to ask if there are cases where manual maintenance is required. Indeed, there are cases when we cannot rely on automation only. For example, the existence of traceability links may depend on specific stakeholders' knowledge that is usually not encoded in any models (cf. [41]). It could also be that there is no technique available to support automatic maintenance of traceability links because there are complex dependencies that need to be checked in order to decide about the maintenance. In the previously mentioned consistency example, the need for checking consistency between an SA and an RA cannot be decided automatically because being interested in checking consistency does not hold all the time. Especially in the beginning of modeling an SA, there are many inconsistencies because the model is not completely defined yet.

Thus, a coexistence of manual and automatic maintenance of traceability information should be considered by our approach. But what maintenance activity should be considered to be manual and which to be automatic? Concerning the (re-)establishing of validity we already said that manual tasks may be required in specific cases (see Sect. 5.5). Concerning the creation and deletion of traceability links, we can have different combinations of automation and non-automation. For example, it might be necessary that traceability links should be automatically created but not necessarily automatically deleted because of history keeping. However, it might also be that traceability links have to be manually created and should be automatically deleted whenever some context

is missing. A common combination is the automatic creation and deletion of traceability links, which may also be used for traceability links which intention is satisfied as long as they exist. In other cases, traceability links might be created and deleted manually whenever information for automatic reasoning is missing or other reasons exist for not automating their maintenance. Thus, our approach has to be able to provide any combination of manual and automatic maintenance processes.

2.2.3 Efficient traceability maintenance

The maintenance of traceability links should be automated if possible. Additionally, the efficiency of maintaining traceability links must be acceptable. Therefore, we already suggested a separation of maintaining the creation/deletion of traceability links and (re-)establishing the validity of traceability links to avoid completely deleting and subsequently re-creating traceability information as soon as their intention becomes suspect. This can lead to an increase of performance because it enables, to some extent, the incremental maintenance of traceability information concerning suspicious traceability links.

In a first step, all traceability links have to be created as efficiently as possible. Because the number of types of traceability links, models and model elements might be high, we need a technique for automatically creating and deleting traceability links with acceptable performance. Thus, the requirement to our technique is that it can be considered as lightweight operation. This means the technique should not include complex analysis or transformations to reason about the existence of traceability links. Furthermore, also the process of applying these techniques to automatically create and delete traceability links should be efficient. However, depending on the technology that is applied, side effects may be possible and thus the process of applying these techniques may not necessarily terminate, and further not necessarily provide a unique result (e.g., termination and confluence of the related rewrite system [27]). Thus, the approach of applying these operations must handle these issues. This depends on the expressiveness of the chosen technology. Another requirement to the lightweight operations, especially to the automated creation, is to decouple the intention of traceability links from the intention of the operations that create the traceability links. In literature, different technologies are applied for deriving traceability links, which do not fulfill this criteria (cf. [16,25,26,36,42]). These traceability links are classified as *derived* traceability links. The opposite are *inferred* traceability links, which is what we require. This classification is taken from [1].

In a second step, the validity of traceability links have to be (re-)established from time to time. We cannot assume that only lightweight operations can be applied but rather heavyweight operations or even manual labor is required to achieve this. Such heavyweight operations might be non-incremental model transformations [16,42] or incremental model transformations [25,26,36], which take models as input and produces models as output. Thus, we rather have to focus on employing an efficient application of these heavyweight operations or carefully plan the application of manual tasks. The efficiency of such an application process might be in danger because the order of applying the operations or even tasks can lead to severe performance deficiencies if an appropriate application order is not considered. This issue arises because we can have contextual overlaps and thus applying in an unfavorable order may lead to multiple unnecessary applications of the operations, which can be time-consuming or even costly. Thus, deriving an appropriate order of applying these operations or manual tasks is required.

2.3 Conceptual overview

Now that we have outlined the open challenges, we give a conceptual overview of our approach and how it addresses the discussed challenges. The follow-up of this section provides a conceptual foundation for this article using a structure that is similar to the rest of this article. We will first explain the foundation of our approach, which is a combined traceability model named dynamic hierarchical mega model. Based on that model, we provide the concepts behind the localization and the execution process for efficient maintenance.

2.3.1 Dynamic hierarchical mega models

To provide a comprehensive traceability approach, dynamic hierarchical mega models contain representations of models and model elements the models are made of. The representations do only contain minimal information about the actual models and model elements they represent due to memory reasons. Nevertheless, technologies for further maintenance can exploit the representation references to navigate to all required information for further reasoning. Because we use representations of models and model elements, a synchronization mechanism is applied, which keeps the representations within dynamic hierarchical mega models synchronized with the actual models and model elements that are observed. Thus, dynamic hierarchical mega models have an up-to-date view on everything that matters. Furthermore, dynamic hierarchical mega models provide traceability links between representations of models and model elements (high-level and low-level traceability links). They also automatically provide hierarchical dependencies between representations of model and their containing model elements (maintained by the synchronization process). These dependencies are exploited to reason about hierarchical dependencies between traceability links. Additionally, hierarchical dependencies between traceability links can be captured by means of hierarchical

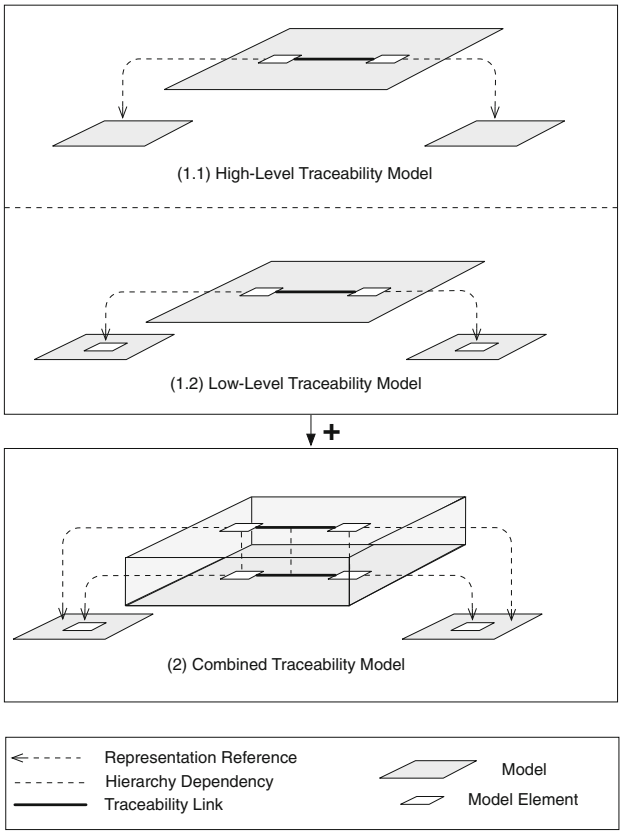

Fig. 7 Combining a high-level and low-level traceability model

dependencies between themselves. We further call these hierarchical dependencies *top-down* and *bottom-up* (see Sect. 3.2).

We classify traceability links into *fact links* and *obligation links*. This classification refers to the two mentioned interpretations of traceability links. A fact link is interpreted as a traceability link that only shows that some intention between modeling artifacts holds (fact). On the contrary, an obligation link is interpreted as the instantiation of some arbitrary operation that is applied to reveal of arrange some information in order to satisfy the intention of the traceability link. Thus, the intention is an obligation that something should hold. Beside traceability links, dynamic hierarchical mega models also capture annotations to traceability links, which are called *characteristics*. A characteristic can be used to additionally provide information that is used by a traceability link or produced by the localization or execution process. Figure 7 shows the concept of integrating a high-level traceability model (1.1) and a low-level traceability model (1.2) into a combined traceability model (2). All details about dynamic hierarchical mega models are given in Sect. 3.

2.3.2 Localization process

The localization process considers lightweight operations to automatically create and delete traceability information within dynamic hierarchical mega models. Therefore, we apply *endogenous* model transformation rules (cf. [33]). The intention of these rules is not to create new models or model elements but to adapt existing dynamic hierarchical mega models by creating/deleting traceability information. We consider these rules as lightweight because these rules only maintain small aspects and because of their endogenous nature. We further refer to the term *localization rules* throughout this article (see Sect. 4.3). Localization rules are further distinguished between *creation rules* (see Sect. 4.3.1), which are responsible for creating traceability information within a certain context, and *deletion rules* (see Sect. 4.3.2), which are responsible for deleting traceability information from a certain context, to decouple the creation from the deletion. This fine-grained distinction facilitates to also manually, e.g., delete traceability information that was automatically created by a creation rule and contrary.

The application of localization rules is obtained by a localization algorithm, which is responsible for the application of all localization rules (see Sect. 4.4). However, because localization rules have side effects that may trigger the application of other localization rules, the localization algorithm might not necessarily terminate or provide a unique result, depending on the order of rule applications. Thus, we restrict the expressiveness of localization rules to overcome this issue but still satisfy our needs (see Sects. 4.4.1 and 4.4.2). Additionally, because traceability links have dependencies between each other (e.g., top-down and bottom-up), the order of applying creation and deletion rules plays an important role concerning performance. Thus, we define an optimized localization algorithm that avoids unnecessary rule applications (see Sect. 4.5) to improve the efficiency.

2.3.3 Execution process

The execution process considers lightweight and heavyweight operations to automatically (re)-establish the validity of traceability links. However, different technologies may be applied, depending on the intention of the traceability links, or even manual tasks are required. Thus, we use an abstract concept for these operations that is not inherently bound to any technology. Such an abstract concept is called *task*. A task requires input from its context and produces some output based on the input. The context of a task is given by a traceability link that is related to the task.

Because traceability links can have overlapping contexts, they can have indirect dependencies with each other through their context, which is further called *data flow* (see Sect. 5.2). Because of data flow dependencies, a careful application of tasks is also required because of performance reasons. Neglecting the application order of tasks may result in unnecessary task applications, which can lead to severe performance issues because of the possibly heavyweight nature of

tasks. A proper ordering of tasks can be conducted because traceability links in data flow dependencies use directed dependencies between modeling artifacts (modeling artifacts that are used as input and output). Thus, we provide an execution algorithm that focuses on deriving a proper task ordering. Therefore, we derive a *task model* that is an abstract representation of all data flows (see Sect. 5.3). Based on the task model we automatically derive an appropriate task scheduling, which is an ordered list of tasks for further application. The execution algorithm further ensures that cyclic dependencies, which can prevent a proper result, are detected (see Sect. 5.4).

3 Dynamic hierarchical mega models

In this section, the foundation of our comprehensive traceability approach is introduced. Therefore, we define a combined traceability model called the dynamic hierarchical mega model. We will explain the structure of dynamic hierarchical mega models by means of its meta model. The meta model is subsequently extended with aspects that are required for the localization and execution process, respectively, in the following sections.[8] We first explain the initial meta model of dynamic hierarchical mega models in Sect. 3.1. In addition, we explain two exemplary dynamic hierarchical mega models in Sect. 3.2 taken from our case study of Sect. 2.1.

3.1 Meta model

The meta model of dynamic hierarchical mega models is specified by means of a class diagram, which is shown in Fig. 8. It has to be noted that this meta model does only contain aspects that are necessary to explain the concepts of this article. Details that are relevant to model management are omitted. Thus, in comparison to the originators of mega models [4,9,10,18], we do not explicitly capture, e.g., meta models of models as first-class entities in the meta model of dynamic hierarchical mega models.

DynamicHierarchicalMegaModel acts as a container for all representations of modeling artifacts, traceability information, etc., which are further maintained by the localization and execution process. An MDE environment is *observed* by a single dynamic hierarchical mega model, which contains representations of all models and model elements in the MDE environment.[9] TraceabilityElement is the root element of all modeling artifact representations and traceability-related elements. A TraceabilityElement is further classified

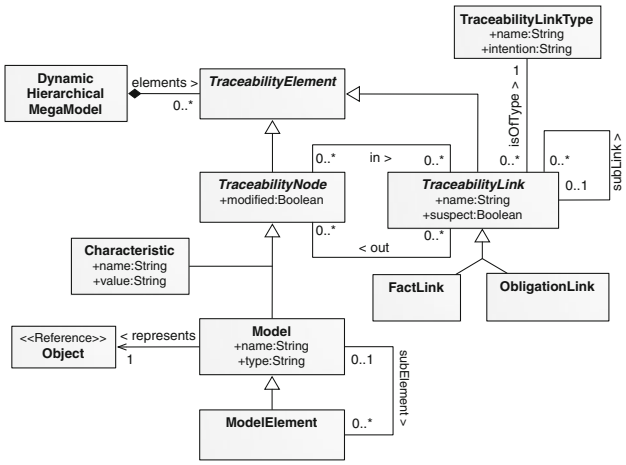

Fig. 8 Foundational meta model of dynamic hierarchical mega models

into TraceabilityNode and TraceabilityLink, which are explained in detail in the following.

3.1.1 Traceability node

TraceabilityNode is any kind of element that can be traced by TraceabilityLink. TraceabilityNode has an attribute modified, which is true whenever an instance of the related element was modified. This flag comes into play when the execution algorithm is applied (see Sect. 5). A TraceabilityNode is further classified into Model and Characteristic. A Model is just an explicit representation of a model. Usually, models itself are not explicitly captured in a model but rather are implicitly available by means of files in a file system or as resources in memory. The mega model (cf. [4,9,10,18]) is a model of models and thus we refer to that notion with dynamic hierarchical mega models. For each model that exists in an MDE environment, there is an instance of Model, which represents the model. A Characteristic is another kind of TraceabilityNode that is used to provide additional traceability information in a dynamic hierarchical mega model. Characteristic is quite simple, consisting of an attribute name for identification and an attribute value, which contains the actual characterization value. Additionally, the elements that are contained in a model are explicitly represented as ModelElement, which allows us to preserve the classical notion of traceability between model elements because ModelElement is a Model and thus a TraceabilityNode. The subElement association between Model and ModelElement is used as hierarchical dependency. It joins both concepts of classical traceability for MDE and global model management into a comprehensive traceability approach. This association is important to quickly access the model elements of a model for further reasoning. The Object class is a reference to any modeling artifact. Model and ModelElement use the represents association to reference their representatives. This is an important

[8] When we write dynamic hierarchical mega model, an instance of the meta model and the subsequently extended ones is meant.

[9] The process of keeping the representations synchronized is explained in Sect. 6.

feature because we can access all details that are required for reasoning during maintenance but do not need to copy all details into dynamic hierarchical mega models. Furthermore, a dynamic hierarchical mega model can represent all models and model elements that conform to the same meta-meta model of a dynamic hierarchical mega model.

3.1.2 Traceability link

TraceabilityLink is any kind of dependency between TraceabilityNode and thus it is used to trace all kinds of modeling artifacts and characteristics. TraceabilityLink has an isOfType association to TraceabilityLinkType, which defines the intention of TraceabilityLink. The attribute name of TraceabilityLink is just used for the purpose of readability. The name of TraceabilityLink is actually defined by the associated TraceabilityLinkType. The attribute intention of TraceabilityLinkType is an informal description of the intention. All instances of TraceabilityLink have a reference to exactly one instance of TraceabilityLinkType. TraceabilityLinkType is actually used for management purposes to quickly access existing traceability links of that type. TraceabilityLink provides a self-association subLink, which defines hierarchical dependencies between traceability links. Together with the ability to capture hierarchical dependencies between representations of models and model elements (subElement), we leverage top-down and bottom-up dependencies between traceability links. Thus, if two instances of TraceabilityLink are connected via a subLink reference, they can be either in a top-down or bottom-up dependency. However, it depends on how they are created. If the existence of low-level traceability links depend on the existence of high-level traceability links, top-down dependencies between these traceability links exist. In cases of bottom-up dependencies the same holds but the other way around.[10]

The core of dynamic hierarchical mega models can be considered as a directed acyclic bipartite graph containing of nodes (TraceabilityNode and TraceabilityLink) and edges (associations in and out) (see Sect. 5.2). We facilitate such a bipartite nature of dynamic hierarchical mega models to leverage data flow dependencies. A data flow basically defines a flow of information. In a data flow, TraceabilityLink is interpreted as an operation that produces some output (out) for a given input (in). Additionally, the output of TraceabilityLink can be the input of other TraceabilityLink instances. Thus, a modular composition of TraceabilityLink instances by means of the in and out references is used to build *complex data flow* dependencies (see Sect. 5).

Furthermore, the semantic of the in and out associations depends on their usage. If the intention of a traceability link is just to capture a dependency, both associations can be interpreted as read directions. Thus, a traceability link would be a directed read dependency (in and out). It can also be used as undirected read dependency when only in associations are used. However, if the intention of a traceability link is to trace the flow (creation/modification) of information, the in and out associations are interpreted as read (source) and write (target) references.

We further introduce a classification of TraceabilityLink into FactLink and ObligationLink. The definition of fact links is basically that whenever they exist, their intention is satisfied and thus no further (re-)establishment of the validity is required. Thus, the intention of fact links is interpreted as a *fact*. However, this implies that they *must* be deleted as soon as their intention is not satisfied anymore. On the other side, the definition of obligation links is that the satisfaction of their intention is not coupled to their existence. Thus, obligation links are less strict because their intention defines that something *should* hold but does not necessarily do so. Therefore, obligation links do not necessarily require to be automatically deleted when their intention does not hold anymore. However, obligation links must have tasks related to (re-)establish the validity in case they become suspicious. Additionally, obligation links can also be created manually. Thus, fact links are maintained by the localization process (see Sect. 4) to support automatic creation/deletion whereas obligation links are additionally maintained by the execution process (see Sect. 5) and must not necessarily be considered in the localization process (if manually created or deleted). If the following question can be answered with true, an obligation link should be considered: *"Do I need an additional task to (re-)establish the validity of a traceability link?"*

3.2 Example

In the following, we outline two examples we taken from our case study, which show how dynamic hierarchical mega models are facilitated to express classical traceability between model elements as well as traceability in global model management between models. Additionally, the examples emphasize hierarchical dependencies that can exist between modeling artifacts and between traceability links and how we exploit a dynamic hierarchical mega model to capture these dependencies. The first dynamic hierarchical mega model describes a top-down dependency whereas the second one describes a bottom-up dependency. In these examples, we have to take the dynamics (creation) of these dependencies into account (informally) to explain the differences because both dependencies look syntactically the same in a dynamic hierarchical mega model.

3.2.1 Top-down

The first example is shown in Fig. 9. Before explaining the figure, the syntax has to be explained because it is used

[10] We show two examples of these dependencies in Sect. 3.2.

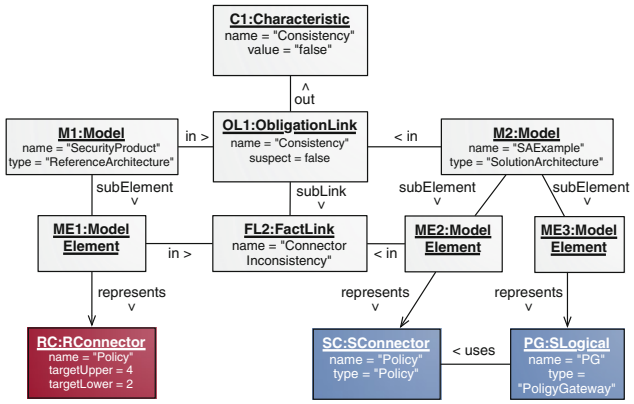

Fig. 9 A dynamic hierarchical mega model: consistency example (top-down)

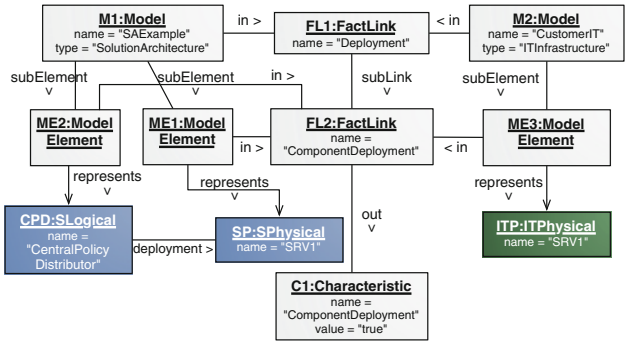

Fig. 10 A dynamic hierarchical mega model: deployment example (bottom-up)

commonly in all examples throughout this article. The examples are shown in abstract syntax which are instances of the meta models. We further have different shadings of the instances. The light-shaded instances denote instances of the meta model of dynamic hierarchical mega models whereas the dark-shared objects are instances of the meta models from the case study.

To the top of the figure, the high-level models M1, representing an RA (cf. Fig. 2), and M2, representing an SA (cf. Fig. 4), are shown and a high-level traceability link (OL1) between them exists. OL1 is an obligation link and we assume that it is created by a modeler to capture the intention that consistency between M1 and M2 should hold. Thus, we let the designer decide whenever to create and delete OL1 because it is an explicit decision by the designer and further consistency may not be checked in all phases of designing an SA. In the considered example, consistency between M1 and M2 holds only if no inconsistency between connectors that are part of M1 and M2 exist. In this particular dynamic hierarchical mega model, only the Policy connector is considered and there is an inconsistency identified, which is expressed by the existence of the fact link FL2 (low-level) between ME1 and ME2. FL2 exists because the connector RC defines that at least two and at most four components can use the connector Policy. However, the related connector SC in the SA[11] is used by only one component PG, which is a consistency violation. The result of (re-)establishing the validity of OL1 is captured by the characteristic C1 that is connected via out with OL1. The value of C1 is *false*, meaning that M1 and M2 are not consistent.

This example shows that we decide about the reasoning of consistency at a higher-level by considering the models M1 and M2. However, if we want to know *why* M1 and M2 are consistent or not, we go into the details of the models by following the subElement references from M1 to ME1 and from M2 to ME2 and ME3 to find the context where consistency of OL1 needs to be considered. Whenever inconsistencies in the identified context are found, fact links exist, which identify these inconsistencies (FL2). By means of the subLink reference we identify whether inconsistencies in the context of OL1 are found or not. In this example, OL1 is not consistent because FL2 is connected via a subLink reference. We call this kind of dependency between traceability links top-down dependency because FL2 is a sub link of OL1 and FL2 requires OL1 to be created for its own existence.

3.2.2 Bottom-up

Another exemplary dynamic hierarchical mega model is shown in Fig. 10. It consists of two fact links FL1 (high-level) and FL2 (low-level). The intention of FL1 is only to depict that a deployment dependency between the models M1, representing an SA (cf. Fig. 4), and M2, representing an ITI (cf. Fig. 6), exists. Thus, to satisfy the intention of FL1, at least one fact link named ComponentDeployment between an instance of SPhysical and ITPhysical of M1 and M2, respectively, has to exist. In this example, such a fact link FL2 exists. FL2 defines a deployment between a physical component SP in an SA and a physical component ITP in an ITI. The existence of FL2 depends on a mapping between the names of SPhysical and ITPhysical instances. FL2 takes also the component into account that should be deployed on the server. In this case, a central policy distributor (CPD) is deployed to a physical component (SP) named SRV1 that also exists in an ITI and thus is available by a customer requesting an SA. FL2 provides a characteristic C1 that is not required in this example but is part of a data flow dependency (see Sect. 5). Whenever an FL2 exists, also an FL1 has to exist between models of the same context (M1 and M2). Therefore, the context of FL2 is required to identify the context of FL1. This bridge is again build by leveraging subElement references between ME2, ME1 and M1 and further between ME3 and M2.

[11] RC and SC are related because the value of attribute type of SC is equal to the value of the attribute name of RC.

This example shows that we can gain traceability information from an existing dynamic hierarchical mega model constellation which reveals, in this particular example, deployment decisions between models. It also shows a bottom-up dependency between the low-level fact link FL2 and a high-level fact link FL1 since the existence of FL1 depends on the existence of FL2.

4 Localization process

In Sect. 3, we have specified the constitution of a dynamic hierarchical mega model by means of a meta model. Now, we introduce how to automatically maintain traceability information in terms of localization. Thus, we are showing how we realize the automatic creation and deletion of traceability information. Therefore, we facilitate endogenous model transformation rules to formally specify lightweight operations and subsequently call them localization rules. We distinguish between creation rules for the creation and deletion rules for the deletion of traceability information.

In the following, we first explain necessary extensions to the meta model of dynamic hierarchical mega models (see Sect. 4.1). Subsequently, we informally introduce model transformation rules, which is the formal foundation of our localization rules (see Sect. 4.2). Then, we explain the constitution of localization rules and their manifestation by means of creation rules and deletion rules (see Sect. 4.3). The application of localization rules is implemented by a model transformation system tailored to the application of these rules. Thus, we also handle known issues of any rule-based system like termination and uniqueness of the result (see Sect. 4.4). Based on these findings, we define an optimized algorithm for localization rules based on a proper partitioning of localization rules (see Sect. 4.5). Finally, we conclude this section by means of an application example (see Sect. 4.6).

4.1 Meta model extensions

In order to manage a set of localization rules the meta model of dynamic hierarchical mega models is extended with LocalizationRule. LocalizationRule is identified by a unique name defined by the attribute name. LocalizationRule further has a ruleURI attribute, which is used to define where to find the actual rule that should be applied during the localization process. LocalizationRule can maintain a specific TraceabilityLinkType, which is expressed by the maintains association. This is not required because a localization rule can also be used to maintain hierarchical dependencies between traceability links (see Sect. 4.3.1). LocalizationRule is further subclassed into CreationRule and DeletionRule to distinguish between the two types of localization rules.

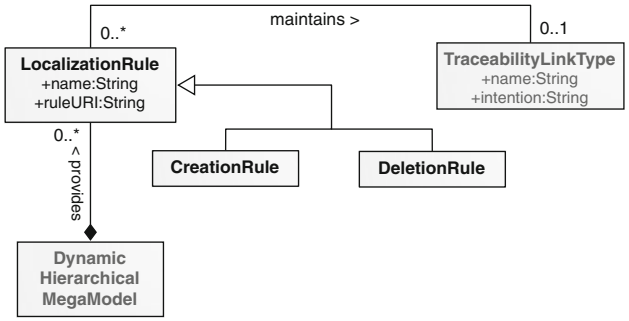

Fig. 11 Meta model extension: localization rules

The extension of the meta model of Fig. 8 is shown in Fig. 11. The alleviated elements are used to show the connection to the initial meta model. The elements that do not change are omitted in this figure.

4.2 Prerequisite: model transformation rules

We briefly outline the semantic of *model transformation rules* informally, which is the underlying formalism of our localization rules.[12] A model transformation rule, or short *rule*, consists of a model pattern describing the *pre-condition* or *left-hand side* (LHS) of the rule and a *post-condition* or *right-hand side* (RHS). When applying a rule to a model, the model pattern of the LHS has to be found in the model. A *model pattern* describes a specific situation in a model by means of required and forbidden elements. The forbidden elements in a model pattern are generally called *negative application conditions* (NACs). Each occurrence of the model pattern in a model is called a *match*, which provides a unique mapping of all required elements of the model pattern to elements in the model and ensures that no such mapping exists for any forbidden element. A match is further modified that it reflects the RHS of the rule. The RHS is specified by elements to be removed/to be created.

We employ *Story Pattern* as formal specification because Story Pattern use model transformation as fundamental specification technique.[13] An exemplary Story Pattern is shown in Fig. 12.

Elements in Story Pattern are objects that can be mapped to instances of meta models by a matching. Each element has a descriptor, e.g., M1:Model whereas M1 is a description

[12] In Appendix A.1, we show a sufficient formalism of graph transformation rules and additional definitions that are used throughout this section. We can map the formalism of graph transformation rules to model transformation rules because models can be considered as attributed graphs.

[13] Practically, Story Pattern are always embedded into *Story Diagrams* for a proper application because Story Diagrams provide additional control-flow concepts. However, we do not explain the concepts of Story Diagrams in this article. More information about Story Diagrams can be found in [29].

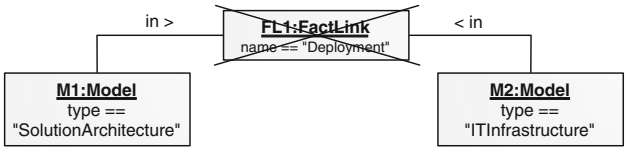

Fig. 12 A Story Pattern including a NAC

that is just used for distinguishing elements in a Story Pattern and Model is the type of the element, which equals the type of the instance in the according meta model. Elements to be created have a ++ annotation and are part of the RHS of a Story Pattern only. Elements to be deleted are annotated with −−. These elements are part of the LHS and the RHS. NAC elements are denoted as crossed out elements and are part of the LHS.

The connections between elements reflect references that need to exist between related elements. The type of the reference is given by an association within a meta model. The type is defined by the annotated description of the reference. For example, in between M1 and FL1 is a reference of the association in between modeling artifacts Model and FactLink in the meta model of dynamic hierarchical mega models.[14] Furthermore, attributes of classes in the meta model can be used as constraints within the LHS or in the RHS to define attribute assignments. For example, name == "SolutionArchitecture" is a constraint, which requires that the attribute name of M1 has a value SolutionArchitecture in order to match. An attribute assignment is specified by the name of the attribute, the assignment operator := and another attribute or constant that should be assigned to the attribute.

4.3 Localization rules

Localization rules are specified by means of Story Pattern and are applied to create and delete traceability information, especially fact links and obligation links. Localization rules do not have the intention of creating or deleting modeling artifacts but traceability information only (endogenous). We further consider concrete localization rules by means of creation rules and deletion rules, which are explained in the following.

4.3.1 Creation rules

The creation rules are responsible to create traceability information, which can be traceability links (fact links and obligation links), characteristics or hierarchical dependencies between traceability links. Therefore, a creation rule defines a model pattern that specifies the context where traceability

[14] A FactLink instance can be connected to an instance of Model because Model is a subclass of TraceabilityNode.

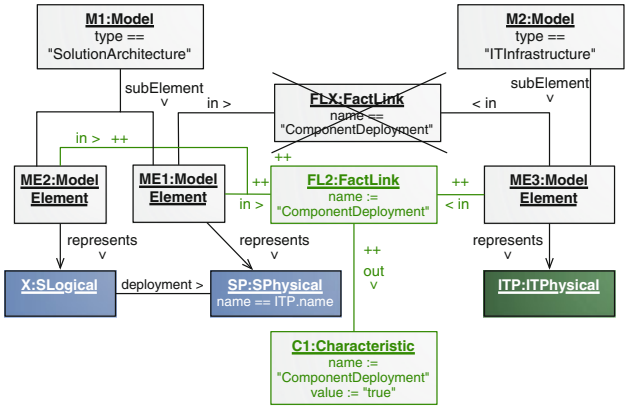

Fig. 13 Creation rule for fact link ComponentDeployment

information needs to be created. The context can be given by a dynamic hierarchical mega model and all referenced models and model elements. Furthermore, creation rules oblique to only create traceability information but not to delete any traceability information. In the following, three creation rules are used to explain the principle of creation rules.[15]

The creation rule of Fig. 13 is responsible for creating a low-level fact link (FL2). The intention of FL2 is to denote that a component deployment exists. Thus, whenever an instance of SLogical (X) has a deployment reference to an instance of SPhyiscal (SP), a deployment should hold. However, a deployment only holds if there is also an ITI (M2) that contains an instance of ITPhysical (ITP), which has a name that is equal to the name of the instance of SPhysical (SP). It is also important that no other fact link exists between such a situation because we do not want to generate this information twice. Thus, we apply a fact link in the model pattern denoted by FLX as NAC. Thus, the model pattern in of Fig. 13 will lead to a match whenever this situation can be found in a dynamic hierarchical mega model. When a match exists, the RHS is executed, which creates FL2, C1 and all required references between. The characteristic C1 just defines that the intention of FL2 holds.[16]

The creation rule of Fig. 14 describes the creation of a high-level fact link (FL1). The intention of FL1 is to denote that a deployment between two models of type SA and ITI exists. However, this depends on the fact that at least one component deployment between elements of these models exists. Thus, the model pattern of this creation rule has to match for a situation that contains a model M1 of type SA and a model M2 of type ITI. M1 must contain a model element ME1 that represents an instance of SPhysical (SP) whereas M2 must contain a model element ME2 that represents an instance of

[15] These creation rules are used to create traceability information as shown in Fig. 10 of the previous section.

[16] We can create these kind of characteristics for all fact links. However, they are optional.

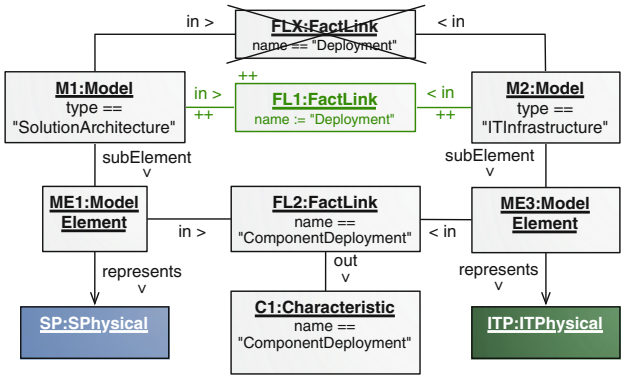

Fig. 14 Creation rule of Deployment

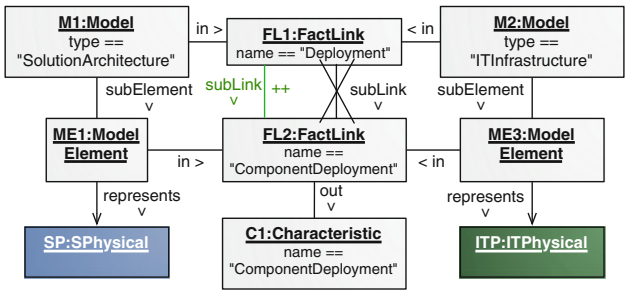

Fig. 15 Creation rule for sub-linking Deployment and ComponentDeployment

ITPhysical (ITP). Between ME1 and ME3, a low-level fact link FL2 named ComponentDeployment must already exist, which is connected to a characteristic C1 via out with the same name as FL2. Additionally, a match is prevented if a high-level fact link (FLX) named Deployment between M1 and M2 already exists. Whenever a match can be found, the RHS creates a high-level fact link (FL1), which is named Deployment. FL1 is additionally referenced via in references to M1 and M2. Thus, the traceability links Deployment and ComponentDeployment are in a bottom-up dependency because the high-level traceability link FL1 depends on the existence of a lower-level traceability link FL2. Because of that dependency, this creation rule can only match if the creation rule of Fig. 13 already applied in an overlap of its own context.

Now, we are able to automatically create two kinds of fact links called ComponentDeployment and Deployment. However, in the example of Fig. 10 there is a subLink reference between these fact links, which cannot be created yet. Therefore, we have to specify another creation rule, which does not create a traceability link, but rather hierarchical dependency (subLink) between these fact links. We cannot define its creation within the creation rule of Fig. 13 because other traceability links called ComponentDeployment can be added subsequently, which would not be covered in that case. We can also not define its creation within the creation rule of Fig. 14 because only one fact link called ComponentDeployment would be matched. Figure 13 shows a creation rule that is responsible for creating a subLink reference between a fact link called Deployment and a fact link called ComponentDeployment. The subLink reference is further only created when no other subLink between those two traceability links exists yet (Fig. 15).

4.3.2 Deletion rules

In contrast to creation rules, deletion rules are responsible for deleting traceability information only. Particularly, a deletion rule should reverse the effect of a related creation rule, which means to delete all information created by the related creation rule. The deletion rule should match if the intention of the traceability link it is responsible for does not hold anymore. The intention does not hold anymore if the required context is no longer available.[17]

4.3.3 Derived deletion rules

Creation rules and deletion rules can potentially exist on their own, e.g., a creation rule for a specific traceability link type could exist without a corresponding deletion rule and vice versa (especially for obligation links). In the case of fact links, we support that if a pair of such rules exists (a creation rule with a corresponding deletion rule concerning a specific traceability link) the deletion rule can be automatically derived from the creation rule, like described below, which guarantees the proposed maintenance of fact links.

A systematically derived deletion rule has first to check whether the context of a traceability link is no longer available. The LHS of the deletion rule can be derived from the LHS of the creation rule and the traceability information, which is added by the creation rule when it is applied. The LHS of the deletion rule can be constructed by negating the previously described elements of the LHS of the creation rule (plus the created elements). This can be automatically done by replacing each element of the LHS of the creation rule with its forbidden element in the form of a NAC.[18] If the context cannot be matched successfully, the traceability links and its residual context will be subsequently deleted.

Figure 16 shows a derived deletion rule from the creation rule shown in Fig. 14, which checks if at least one of the necessary elements in the context of the fact link FL1 is missing. If this is the case, the fact link FL1 is deleted and the shown rule is able to automatically remove all traceability

[17] This holds for fact links especially. It does not necessarily hold for obligation links as explained in Sect. 3.

[18] NACs in Story Pattern have OR semantics. Thus, if at least one element in the context of a deletion rule cannot be matched, a match exists and the related traceability information will be deleted.

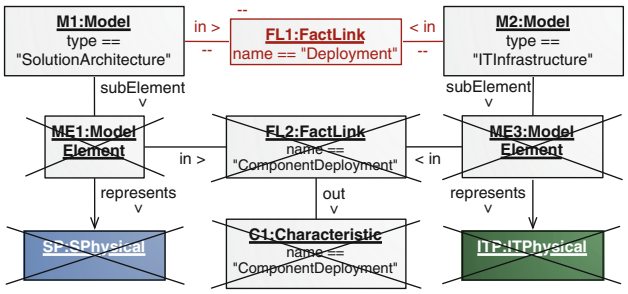

Fig. 16 Derived deletion rule of Deployment

information, which is previously created by the creation rule shown in Fig. 14.

4.4 Localization algorithm

A first straightforward solution would be applying a given set of deletion rules $\mathcal{R}_d$ and creation rules $\mathcal{R}_c$ until a fix point has been reached. In this case whenever a rule is applied successfully, all rules have to be applied again because the application of one rule could potentially enable the application of another rule. Such a simple solution is shown by means of the following pseudo-code algorithm.

```
1  procedure SimpleLocalization()
2  parameter: M, R_d, R_c
3  return fixPoint(M, R_d ∪ R_c)
```

Listing 1 A simple localization algorithm

The procedure fixPoint, used within the above described pseudo-code algorithm, is defined in Appendix A.1. The termination condition of the fix point computation is satisfied if no matches for rule applications were found where the application results in any change. However, it is not clear if this condition will always be true after a finite number of rule applications (termination) and whether a unique result exists.

4.4.1 Termination

Creation rules in $\mathcal{R}_c$ never result directly in a new match for any deletion rule in $\mathcal{R}_d$ due to the fact that deletion rules by construction can only have a new match when an element has been deleted and creation rules never delete an element. However, it is possible that a deletion rule application results in a new deletion rule match and that the deleted element has been created beforehand by a creation rule. Therefore, in general the rule application of $\mathcal{R}_c \cup \mathcal{R}_d$ until a fix point has been reached as employed in Listing 1 not necessarily terminates. We can however exclude such infinite sequences of rule applications by first computing the fix point for $\mathcal{R}_d$ and than for $\mathcal{R}_c$ as depicted by Listing 2. As the application of $\mathcal{R}_d$ until a fix point is reached guarantees that no invalid traceability links[19] remain and the application of $\mathcal{R}_c$ until a fix point is reached guarantees that all valid traceability links are created, this reordering also leads to the required result.

```
1  procedure SufficientLocalization()
2  parameter: M, R_d, R_c
3  M' := fixPoint(M, R_d)
4  M := fixPoint(M', R_c)
5  return M
```

Listing 2 A sufficient localization algorithm

To show the termination of the fix point in case of the deletion rules $\mathcal{R}_d$, we can simply count the strong monotonously decreasing number of elements after each rule application. By construction holds for all rules $r \in \mathcal{R}_d$ that they delete more elements than they create and thus are strong monotonous decreasing with respect to the number of elements (see Definition 1). For such a rule set of strong monotonously decreasing rules holds that the repeated application until a fix point has been reached always terminates. This is the case because the overall number of elements of a dynamic hierarchical mega model is finite and decreases with each rule application and thus only finite many rule applications are possible.

The same trick obviously does not hold for the creation rules $\mathcal{R}_c$. Nevertheless, we can use the idea of *exclusive graph pattern* and *excluded graph pattern* to also check the required termination for $\mathcal{R}_c$.

An exclusive graph pattern for a rule r is a model pattern p contained in the pre-condition of r such that for any match of that model pattern only one successful application of that rule is possible (see Definition 2). An example for an exclusive graph pattern is a rule r, which includes a model pattern p with a forbidden element (NAC) in the pre-condition. When r also adds in the post-condition the same element such that the same match is not valid any more, p is an exclusive graph pattern. An excluded graph pattern for a rule r is a model pattern p such that the post-condition of r cannot contribute to any new match for p (see Definition 3). More details can be found in Appendix A.2. An example for an excluded graph pattern is a rule r where the application of r only creates or deletes elements, which are not part of the model pattern p.

Figure 17 shows an example of a rule including a model pattern, which fulfills the conditions to be an exclusive as well as an excluded graph pattern. The model pattern itself is the pre-condition contained in the shown rule.[20] The model pattern is exclusive because the NAC has the same type of element (with the value Deployment of the attribute name) as it is created when the rule is applied. Additionally, the NAC as well as the newly created element is associated to the same element within the rule. After the first application,

[19] Traceability links that are missing context or have forbidden context.

[20] All elements, which are not annotated with ++.

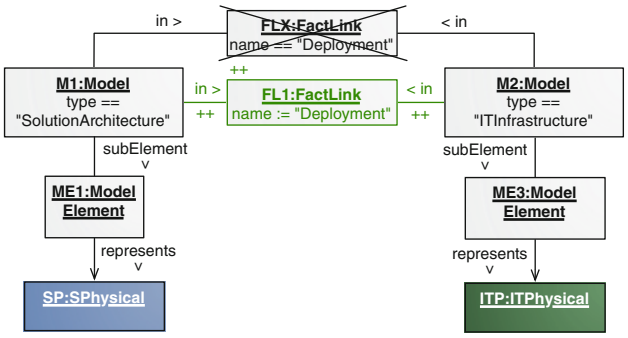

Fig. 17 A rule including a model pattern, which is exclusive as well as excluded

the NAC prevents that the same rule could be repeatedly applied to same context. From this it follows that the post-condition prevents that the rule could be applied multiple times to the same context.

The model pattern of the shown rule would be not excluded for a set of rules if the post-condition of the shown rule contributes in any way to a new match of any rule, which includes the same model pattern. This could be potentially the case if the post-condition of the shown rule creates or deletes an element, which is included in the pre-condition of any rule in the rule set. However, the only element that is added by the post-condition is the element of type FactLink[21] (with the value Deployment of the attribute name). By construction exactly this type is also included as NAC and as a consequence the application of the shown rule could only *invalidate* existing matches but not contribute directly to a new match. From this it follows that the model pattern of the shown rule is an excluded graph pattern. If we can show for a rule $r \in \mathcal{R}$ and a model pattern p of the rule r holds that p is an exclusive graph pattern for r and p is an excluded graph pattern for all $r' \in \mathcal{R}$, no infinite sequence of rule applications of r can exist.

In the following, we describe why these conditions are fulfilled by construction for a set of creation rules $\mathcal{R}_c$. For each rule $r \in \mathcal{R}_c$ holds that at least one element and one forbidden element, like in cases of the forbidden element FL1 (with the value Deployment of the attribute name) shown in Fig. 14 is included in the pre-condition of p of each rule r. Such a forbidden element plus the rest of the pre-condition is an exclusive graph pattern for that rule.[22] As no rule in $\mathcal{R}_c$ creates any modeling artifacts but only traceability information, it also holds that all pre-conditions of the rules, which contain by construction at least one element that represents a modeling artifact and not only elements from a dynamic hierarchical mega model, are excluded graph pattern for all rules in $\mathcal{R}_c$ by construction. We can thus conclude that the application of the rules in $\mathcal{R}_c$ until a fix point is reached always terminates, as we know that each rule $r \in \mathcal{R}_c$ can only be applied finitely often.

4.4.2 *Uniqueness of the result*

Another problem is the question whether the result of applying the rules is independent of the order how the rules are applied. A pair of rules $(r, r') \in \mathcal{R}^2$ is *parallel independent* if for all elements and matchings for both rules holds that the rules can be applied in any order and that any application of r for a match does not disable another match for r' and vice versa. If the application of all rules in $\mathcal{R}_c$ terminates and all pairs of rules (r, r') with $r, r' \in \mathcal{R}_c$ are pair wise independent the sufficient conditions are fulfilled for a so-called strong confluent rule application system (cf. [27]). More details about the confluence of graph rewriting rules can be found in Appendix A.3.

A set of rules $\mathcal{R}_c$ is potentially not confluent if a critical pair exists (cf. [27]). Conceptually, such a pair exists if a situation is possible where the ordering of two rule applications result in different outcomes as one rule application blocks the other. Such a situation does not necessarily lead to a different overall result for applying a set of rules until a fix point is reached. However, it is sufficient to exclude that such critical pairs exists to conclude that the set of rules is locally confluent and confluence follows. Concerning the independence of the order of rule applications for the creation rules $\mathcal{R}_c$ one important property is that each $r \in \mathcal{R}_c$ only creates elements. Without any forbidden elements in the model pattern and without the deletion of elements, the application of the rules within $\mathcal{R}_c$ lead to identical results independent of the ordering (cf. [35]). Hence, rules within $\mathcal{R}_c$, that are not parallel independent, can only result from forbidden elements.

It holds that in the LHS of each creation rule only one forbidden element is contained, which is the same as the element that will be created if the rule is successfully applied. This forbidden element can lead to the problem that a rule cannot be applied any more for a match when another rule has created the same element. However, by construction (see Sect. 4.3.1) such a case is not possible because only traceability information are created and the elements are only created by a single rule. Also the forbidden element in the NAC is exclusive for this rule. Thus no critical pair can exist in $\mathcal{R}_c$ and following $\mathcal{R}_c$ is local confluent, which implies in the case of termination that a uniquely defined result is obtained for any initial dynamic hierarchical mega model.

As outlined before for our termination argument, the deletion rules do not really contribute to the result, but help to

[21] Also the in associations are created, but associations on their own without a connected element are not allowed to exist within a model pattern in our approach.

[22] Each rule $r \in \mathcal{R}_c$ is related to exactly one type of traceability link and this type is included in the model pattern p of r in the form of a NAC.

minimize the creation effort by only erasing elements where the context required by the creation rules is missing. Therefore, also for the combinations of creation and deletion rules the same uniquely defined outcome must result.

4.5 Optimized localization algorithm

The sufficient algorithm is not very efficient in case of larger rule sets. While the termination proof guarantees that no infinite sequence of rule applications can happen, in the worst case we may choose an ordering for the rule set $\mathcal{R}$ which requires to run over the whole set $|\mathcal{R}|$ times only taking into account that the last rule triggers the second to last and so forth.

In the following, we will show the impact of dependencies between rules and how they can be exploited to optimize the ordering. We say that a rule r_2 *depends directly* on a rule r_1 if r_2 does have required elements or references in its pre-condition, which are created by r_1 or r_2 or does have forbidden elements in its pre-condition, which are deleted by r_1 (see Definition 5). Informally for a pair of rules, one rule depends directly on another if the rule application of one rule can potentially influence the applicability of the other rule.

The existence of such dependencies can help us to find valid orderings to later optimize the execution. Given two rules r, r' and if r' does not depend directly on r the property holds that first executing rule r until a fix point has been reached and afterwards executing r' (till fix point reached) leads to the same outcome as the execution of both rules alternatively until a fix point has been reached.[23] This is the case because r has no element in its pre-condition, which is affected by any execution of r' (because r does not directly depends on r').

We use this concept to reason about the dependency of rule sets and further to derive a dependency-related ordering of rule sets. Given a set of disjoint rules sets $\{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$ with $\mathcal{R}_i \subseteq \mathcal{R}$, we define for any $i \neq j$ that $\mathcal{R}_i$ *not depends* on $\mathcal{R}_j$ if no $r_i \in \mathcal{R}_i$ and $r_j \in \mathcal{R}_j$ exists with r_j depends directly on r_i (see Definition 6). If this is fulfilled for all $\mathcal{R}_i$ we are able to compute the fix point for each set separately rather than simultaneously for all rules in $\mathcal{R}_1, \ldots, \mathcal{R}_n$. Finally we can derive a dependency-related ordering of rules sets $\langle \mathcal{R}_1, \ldots, \mathcal{R}_n \rangle$ that is a *valid ordering* of $\mathcal{R}$ for any $i < j$ so that $\mathcal{R}_i$ not depends on $\mathcal{R}_j$ (see Definition 7). For a valid ordering we can conclude that the given order can be used to apply the rules of the different rule sets one after another without changing the outcome.

Using the attained insights that the sufficient algorithm terminates and has a unique outcome, we can optimize this algorithm by generating a valid ordering of rule sets $\langle \mathcal{R}_1, \ldots, \mathcal{R}_n \rangle$ and compute the fix points for each rule set independently. However, as the dependencies between creation and their related deletion rules are by construction the same and when a set of creation rules is independent of another set of creation rules this also applies to the related deletion rules. Therefore, we can delay the execution of the deletion rules until the next iteration of applying creation rules that depend on them. This is in fact a possible improvement, as therefore fewer elements may be deleted as the already applied creation rules may have enabled some required context elements and thus less application deletion rules may result. Please note that as depicted in Listing 3 to ensure termination we still have to split $\mathcal{R}_i$ into its deletion and creation rules and execute them separately in the right order.

```
1  procedure OptimizedLocalization()
2  parameter: M, ⟨R_1, ..., R_n⟩
3  //⟨R_1, ..., R_n⟩ is a valid ordered list of rule sets
4  for i in 1 to n do
5    M' := fixPoint(M, R_i ∩ R_d)
6    M := fixPoint(M', R_i ∩ R_c)
7  endfor
8  return M
```

Listing 3 An optimized localization algorithm

4.6 Example

To show how the localization rules and the localization algorithm works together, we give an example that refers to the example of Fig. 10. The example contains three chronologically ordered snapshots of a dynamic hierarchical mega model as shown in Fig. 18. The first snapshot (a) of Fig. 18 describes the initial situation before applying the localization algorithm. As shown in the figure, the already created traceability link FL2 does not reflect its intention due to manual changes. In comparison to Fig. 10, the name of SP changed from SRV1 to SRV2. Thus, FL2 should be deleted because its intention of reflecting a correct deployment dependency is not satisfied anymore.[24] Therefore, the localization algorithm[25] is applied. First, all deletion rules are applied. The first successful deletion rule application removes FL2 and C1 because its context does not exist anymore. Then, FL1 is removed subsequently by another deletion rule application, too, because there is no other fact link called ComponentDeployment connected to FL1. After the application of all deletion rules reached their fix point, the dynamic hierarchical mega model looks like shown in snapshot (b) of Fig. 18. The next step of the localization algorithm is applying all creation rules. The first successful creation rule creates FL2 between ME1 and ME4 and a characteristic C1, which reflects the new

[23] If the rule applications have a unique result and terminate at all.

[24] A correct deployment dependency would be if FL2 is connected via in to ME4.

[25] We use the sufficient algorithm of Listing 2 for this example, which first applies all deletion rules and subsequently all creation rules.

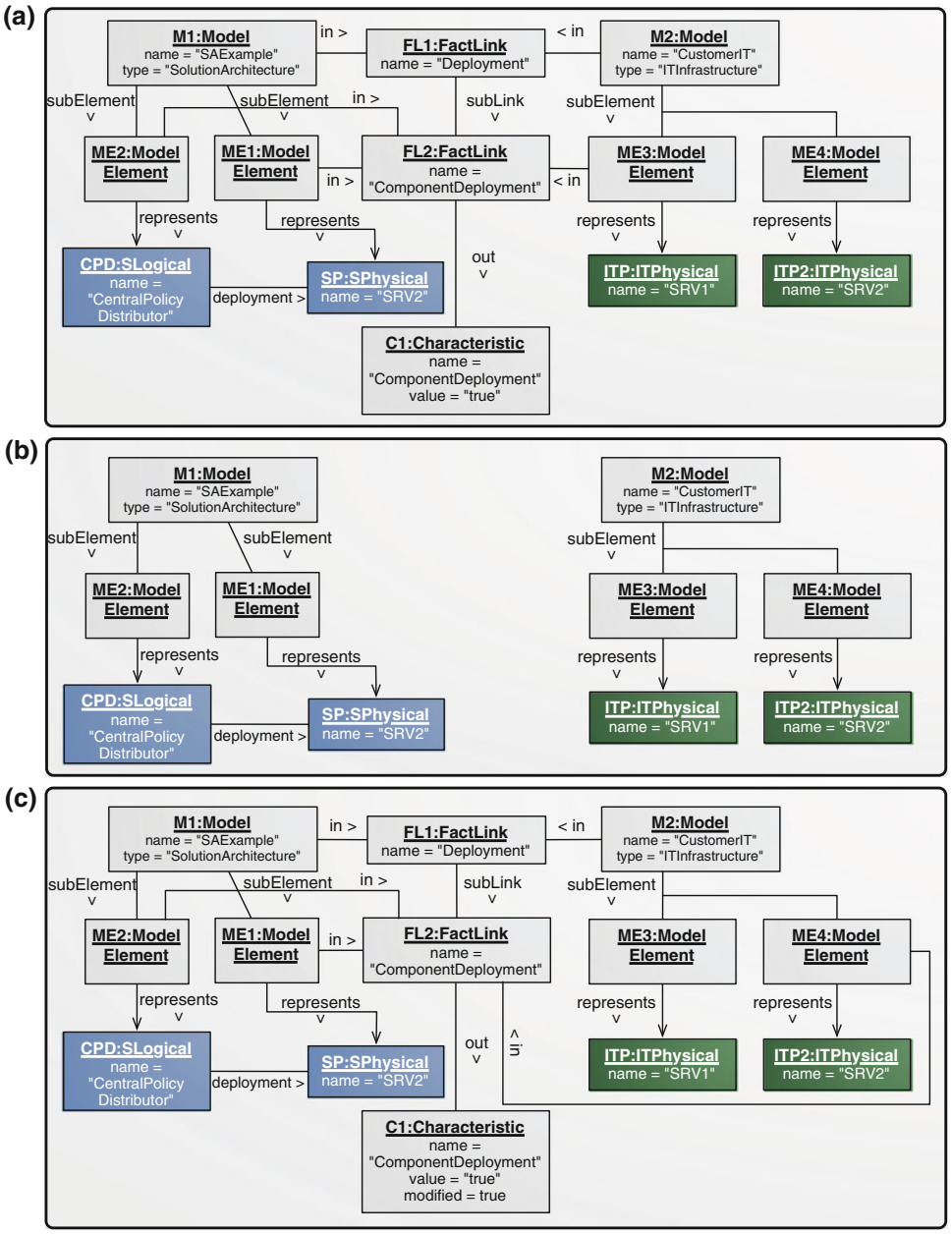

Fig. 18 Example of applying localization rules. **a** Dynamic hierarchical mega model before first round of the localization algorithm (name of SP changed from SRV1 to SRV2). **b** Dynamic hierarchical mega model after applying all deletion rules. **c** Dynamic hierarchical mega model after applying all creation rules subsequently

component deployment dependency. Then, FL1 is created again between M1 and M2 and finally a subLink reference is established between FL2 and FL1. The localization process is finished because all traceability links are valid again.

5 Execution process

In the previous section, we explained that fact links and also obligation links can be automatically created and deleted by means of endogenous model transformation rules. However, we made the restriction that whenever fact links become suspect it is sufficient to check whether the fact links still have to exist. Thus, it is sufficient to execute deletion rules, related to the fact links, to (re)-establish their validity. Whenever fact links are not deleted by their deletion rules, we consider these fact links as not suspect anymore until their context change again.

When it comes to obligation links, things are different. The validity cannot be (re-)established by only structurally

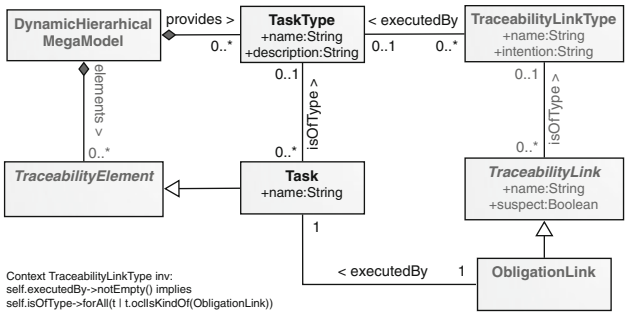

Fig. 19 Meta model extension: Tasks

checking the context.[26] Thus, more comprehensive techniques may be required. Therefore, we introduce the concept of tasks for the subsequent maintenance of the validity of obligation links. We do not propose any specific technology behind tasks since available technologies that can be applied are manifold. Furthermore, obligation links can have diverse intentions that cannot be covered by fact links and thus we may apply different kinds of technologies behind tasks, e.g., consistency checks or development activities such as model transformations, etc. In certain cases, even manual tasks are required to (re-)establish the validity of obligation links.[27]

In this section, we are also interested in a solution to reveal a sequence of tasks that needed to be executed in order to (re-)establish the validity of obligation links that became suspect. Simply applying tasks in an arbitrary order is not sufficient because tasks are generally not independent due to dependencies in between. We need to identify these dependencies and than order the tasks in an appropriate sequence for further execution.

We start by presenting an extension of the meta model of dynamic hierarchical mega models. Then, we define how data flow dependencies between traceability links yield to dependencies between tasks. Subsequently, a task model is introduced, which is used by the task scheduling (execution algorithm) to provide an appropriate sequence of tasks. We close this section with an example from our case study.

5.1 Meta model extension

To apply the execution process on dynamic hierarchical mega models, we extend their meta model with the concept of tasks as depicted in Fig. 19.[28]

TaskType is related to a specific technology for automating the maintenance of obligation links. It can also be used as a plan for the subsequent maintenance obligation links. The attribute name of TaskType is unique, which means that only a single instance of TaskType with a specific name can occur in a dynamic hierarchical mega model. TaskType is associated to exactly one TraceabilityLinkType via the executedBy association between TaskType and TraceabilityLinkType. This association denotes that the validity of an obligation link can be (re-)established by a task that references a task type that can execute a traceability link type, which is the type of the obligation link. TaskType is always related to exactly one TraceabilityLinkType, expressed by the association executedBy, which denotes that a task Task of the specific TaskType can subsequently maintain the TraceabilityLink of a specific TraceabilityLinkType. If an instance of a TraceabilityLinkType that is referenced with an instance of a TaskType exists, then for all instances of ObligationLink, which are related to the instance of TraceabilityLinkType, there is an instance of Task, which is related to the instance of TaskType.

5.2 Task dependencies

As mentioned previously, data flow dependencies are possible in dynamic hierarchical mega models. Data flows are defined by traceability links and related traceability nodes. Additionally, the flow direction is given by the references connecting the traceability nodes (in and out). We further have to distinguish between *atomic data flows* and *complex data flows*. Each obligation link is considered as an atomic data flow because it takes data and further provides data based on the consumed data. Complex data flows exist as soon as two obligation links are indirectly coupled via their context, e.g., the first obligation link provides data that is consumed by the second obligation link.

In Fig. 19 it is shown that obligation links can be executed by tasks. This relationship makes the obligation link acting as the signature of the related task. Thus, we can use the context of obligation links equally as the context of tasks. Now, we have the same dependencies between tasks as between obligation links. In the following, we show a complex data flow in two different situations. The situation reflects the current modification state of a dynamic hierarchical mega model. We use the modification state to explain the task scheduling later on.

Figure 20 shows a dynamic hierarchical mega model containing two obligation links (OL1 and OL2), each one related to a task (T1 and T2). The illustration shows a complex data flow beginning with the obligation link OL1, which requires a traceability node TN1 as input and subsequently produces a characteristic C1 as output. C1 is further consumed by the obligation link OL2. Additionally, OL2 requires TN2 to produce the characteristic C2. OL1 is executed by the related task T1 of type TT1 and OL2 is executed by task T2 of type TT2. The complex data flow is a combination of two atomic data flows. The atomic data flows are OL1 and OL2. To (re-)establish the

[26] Applying deletion rules.

[27] In the following of this section an example is shown.

[28] The OCL constraint defines that an instance of TraceabilityLinkType can only be related with an instance of TaskType if TraceabilityLinkType acts as type of obligation links.

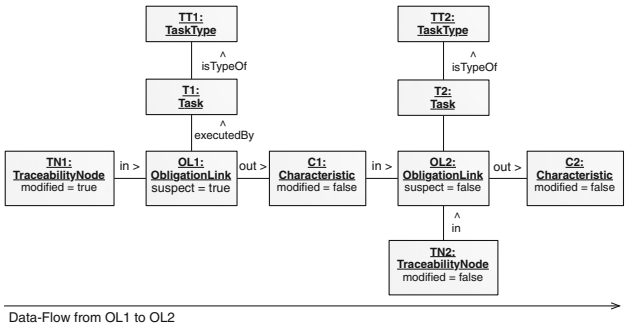

Fig. 20 Illustrating a complex data flow dependency (Situation 1)

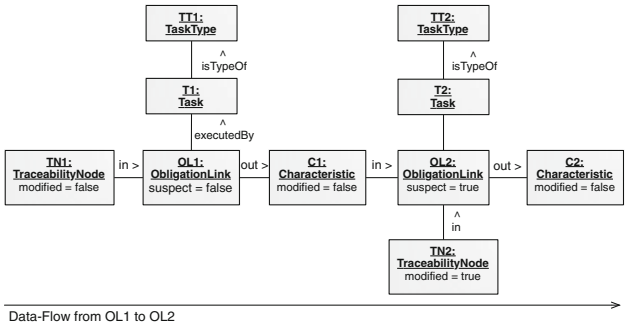

Fig. 21 Illustrating a complex data flow dependency (Situation 2)

validity of obligation links, they need to be executed in a correct sequential order because of their dependencies in the complex data flow.

Concerning Fig. 20, the execution of T2 depends on the results of the execution of task T1 since T1 produces information stored in C1, which in turn is consumed by task T2. The example also shows that TN1 was modified and thus OL1 is rendered to be suspect. Thus, task T1 needs to be executed to (re-)establish the validity of OL1. However, executing task T1 may modify C1 and thus renders OL2 suspect, too. Thus, also the validity of OL2 needs to be (re-)established, which implies the subsequent execution of task T2.

Another situation could be that only TN2 was changed and thus is marked as modified, which is shown in Fig. 21. This implies that OL2 is rendered to be suspect. Thus, to (re-)establish the validity of OL2, task T2 needs to be executed, which does not require the subsequent execution of another task. Thus, traceability links and further tasks can be defined in a modular manner. An important benefit of such a modularity is that (1) data can usually be obtained locally and (2) tasks can be incrementally executed depending on the current modification state.

5.3 Task model

To provide a proper sequence of tasks, which have to be (re-)executed in order to (re-)establish the validity of the related obligation links, we define a simple task model. The task

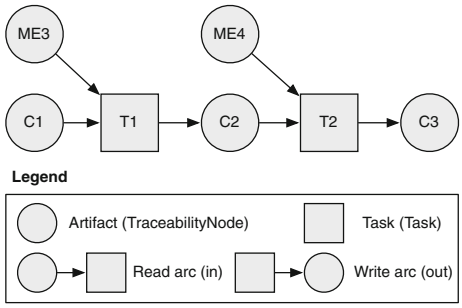

Fig. 22 Task model representing a complex data flow

model is further used by the execution algorithm deriving a task scheduling, which contains a proper sequence of tasks. A task model is a directed, acyclic and bipartite graph (V, E) with vertices V and edges E. We have two types of vertices, which are V_a called artifacts (representations of traceability nodes) and V_t called tasks (representations of tasks) such that $V = V_a \uplus V_t$. The edges E are divided into *read arcs* $E_r \subseteq V_a \times V_t$ and *write arcs* $E_w \subseteq V_t \times V_a$ and thus $E \subseteq V_a \times V_t \cup V_t \times V_a$. These edges are derived from in (read arc) and out (write arc) references between traceability nodes and the related obligation link of the task. The task model can represent multiple complex data flows because the task model does not need to have only connected components. Thus, a dynamic hierarchical mega model can be completely interpreted as a task model for further task scheduling. Figure 22 shows an exemplary task model that bases on the dynamic hierarchical mega model of Fig. 24.

More formal details about the task model can be found in Appendix B.1.

5.4 Task scheduling

When considering the execution of tasks, we only know that those tasks have to be (re-)executed whenever the related obligation link becomes suspect (see suspect flag). This happens if a traceability node in the context of the obligation link was modified (see modified flag). However, as the execution of tasks may also modify traceability nodes and thus could trigger other tasks in a complex data flow it is not suitable to arbitrarily select tasks and further execute them because unnecessary executions can be made and thus severe performance issues may occur. Thus, we have to find a proper task schedule for all tasks that have to be (re-)executed. We apply a task scheduling algorithm on basis of the previously introduced task model to derive such a proper task schedule. To derive a task schedule from a task model, the task model has to be topologically sorted first.[29] Whenever a complex data flow contains cycles, the topological sort will detect this and

[29] The task model in Fig. 20 is topologically sorted (from left to right).

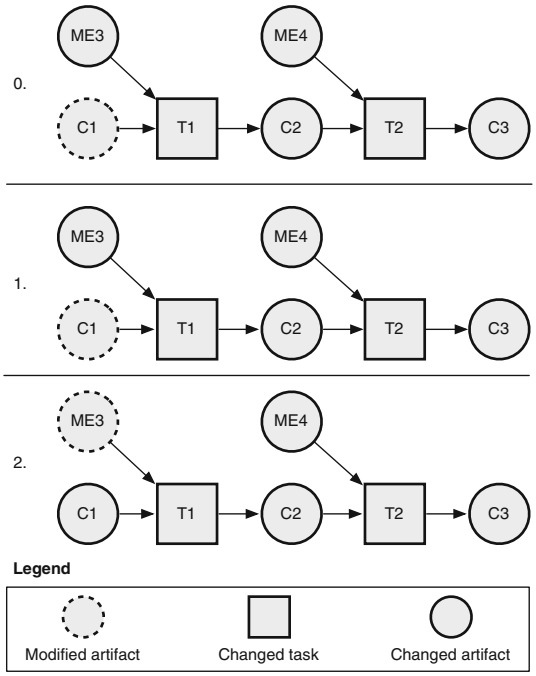

Fig. 23 Exemplary change propagation of the task scheduling algorithm

the task scheduling process is stopped because the execution of tasks would not terminate at all. The task scheduling algorithm is divided into two phases. First, all modified artifacts are considered and based on these artifacts the impact of changes is propagated through the task model by a topological traversal. This is an iterative process where in each step all tasks that are connected to at least one modified or changed artifact are marked as changed, too. In the same iteration, all artifacts are marked as changed, which are connected via write arcs to the previously marked tasks. When all necessary changes are propagated and the first phase of the task scheduling algorithm is finished, a task schedule is derived in a second phase by selecting all changed tasks in a backward traversal adding the tasks sequentially to the beginning of the list.[30] The change propagation of the task scheduling algorithm is exemplarily shown in Fig. 23 by means of the task model shown in Fig. 22.

The change propagation requires two iterations. In the initial situation, we only have a modified artifact (C1). In the first iteration, the task T1 and subsequently the artifact C2 are marked as changed because at least one artifact connected via read arcs is modified or changed (C1). In the second and last iteration, the change propagation has impact to the task T2 and subsequently to the artifact C3. Now, the change propagation phase is completed and the second phase is initiated. The second phase derives a task schedule that contains all changed tasks in a proper order. In this example, the task schedule $\omega = (T1, T2)$ contains tasks T1 and T2. In a subsequent process the task schedule has to be executed by executing the tasks in a sequential order. After executing all tasks, all intentions of the obligation links are (re-)satisfied and the execution process is finished.

5.5 Example

To make the previously shown task model and the task scheduling more concrete, a simple example from our case study is introduced. Figure 24 shows a dynamic hierarchical mega model that represents a situation that is the result of a successful application of the localization process. The traceability link constellation is used to further evaluate the deployment time a central policy distributor (CPD) takes to deploy all its security policies to all indirectly connected security agents. During our cooperation with CA Labs, we noticed that knowing the deployment time of a central policy distributor and validating that it does not exceed a certain time limit is an important performance requirement. In practice, to conduct the deployment time of the central policy distributor, it is simulated in a test lab. In the test lab, the performance of the deployment of security policies is simulated under different load situations. The data to reason about the performance of deploying security policies is shown in Fig. 24. It shows a complex data flow made of two obligation links (OL1 and OL2). The intention of OL1 is to provide the time it takes to deploy all security policies of a CPD to all its clients. Therefore, task T1 is used as a plan to set up a test lab for the simulation of deploying security policies under different load situations. The execution of T1 is conducted manually and the result of the execution is the worst case time to deploy all security policies. This deployment time has to be manually added to the characteristic C1 that is connected via out to OL1. The intention of OL2 is to check whether the deployment time of CPD fulfills a given constraint C. Therefore, task T2 is provided. It is executed by an operation that automatically evaluates whether the given deployment time of C1 of CPD fulfills the deployment time constraint C connected to CPD. In our example, the deployment time is measured with 8 min.[31] The result of this check is subsequently stored in the characteristic C2. In this particular example, the result of checking the constraint is true because $8 \leq 10$.

Applying the task scheduling to the dynamic hierarchical mega model of Fig. 24 is already shown in Fig. 23. It is shown that T1 needs to be executed before T2 because T2 depends on the result of T1. Whenever the input of OL1 changes the suspect flag of OL1 is set to true, which indicates that T1 needs to be (re-)executed because the value of C2 may be invalid. This will subsequently render OL2 to become suspect

[30] Details about a task schedule and the task scheduling algorithm can be found in Appendix B.2 and B.3, respectively.

[31] We define that the unit is minutes in this case.

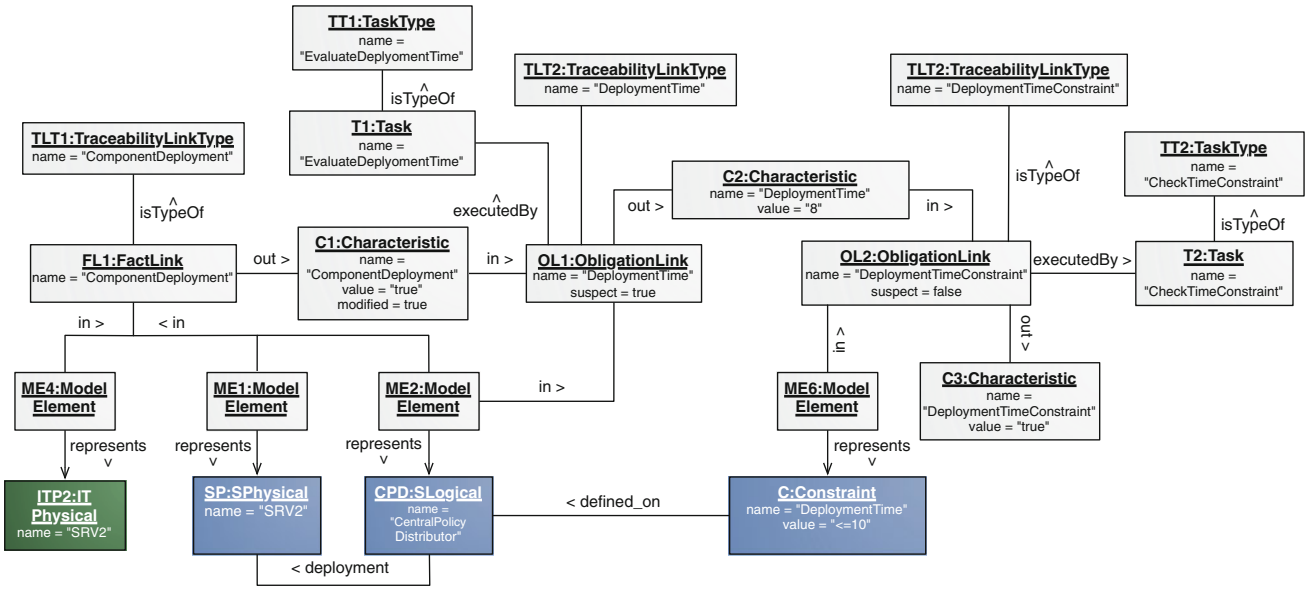

Fig. 24 Example of a dynamic hierarchical mega model considering a data flow dependency between OL1 and OL2 based on our case study

and thus T2 needs to be (re-)executed, too. If the constraint C would change, only OL2 gets suspect and thus the task scheduling would only estimate task T2 to be (re-)executed. The situation in Fig. 24 is already after the execution of the tasks T1 and T2 but after changing the deployment of CPD from SRV1 to SRV2 (see Fig. 18).

6 Prototype and application

During our cooperation with CA Labs, we developed an early prototype, which contains a modeling environment for the models presented in the case study as outlined in Sect. 2.1. It further implements the concept of dynamic hierarchical mega models with efficient maintenance support (localization and execution process). The prototype has been implemented on top of Eclipse[32] as a set of exchangeable Eclipse plug-ins. A screenshot of the implemented prototype is shown in Fig. 25. The modeling overview is shown to the left, which provides a structured view on all models in an MDE environment. It is further used to trigger actions by a developer working with the tool. In the middle, the modeling editors of our prototype are shown. Currently, the SA editor is open, which contains a more detailed version of the example as used in this article. We also implemented a simple highlighting function for elements in the context of traceability links whenever the intention of the traceability link is not satisfied, which is visualized by means of an overlaying red cross. For example, the Policy connector provided by the PolicyGateway component is used by too many SecurityAgent components, which refers to our consistency example of Fig. 9. In the following, we first explain use cases of our prototype and then introduce an architectural overview. Finally, activities are explained which describe the interplay of the architectural building blocks in more detail.

6.1 Use cases

Two roles/stakeholder are considered in our prototype implementation. A tool configurator, who is responsible for setting up a proper Eclipse-based MDE environment, and a developer, who is actually using the configured environment. A tool configuration provides proper domain-specific modeling support (meta models and editors) used within the problem domain of an MDE environment. To provide automated maintenance support, a tool configuration further defines traceability link types, localization rules for the localization process, and task types as well as operations related to tasks for (re-)establishing the validity of obligation links.

A developer can use the pre-configured MDE environment to provide models for his/her facilitating traceability information that are maintained by dynamic hierarchical mega models. Additionally, a developer configures the maintenance mode of the environment. The maintenance modes are explained in Sect. 6.3. The use cases are shown in Fig. 26.

6.2 Architecture

Figure 27 shows a high-level architecture of our prototype with the employed Eclipse plug-ins and the used Eclipse extensions. DHMM (Dynamic Hierarchical Mega Models) is the overall container of all plug-ins of the prototype. The

[32] http://www.eclipse.org/.

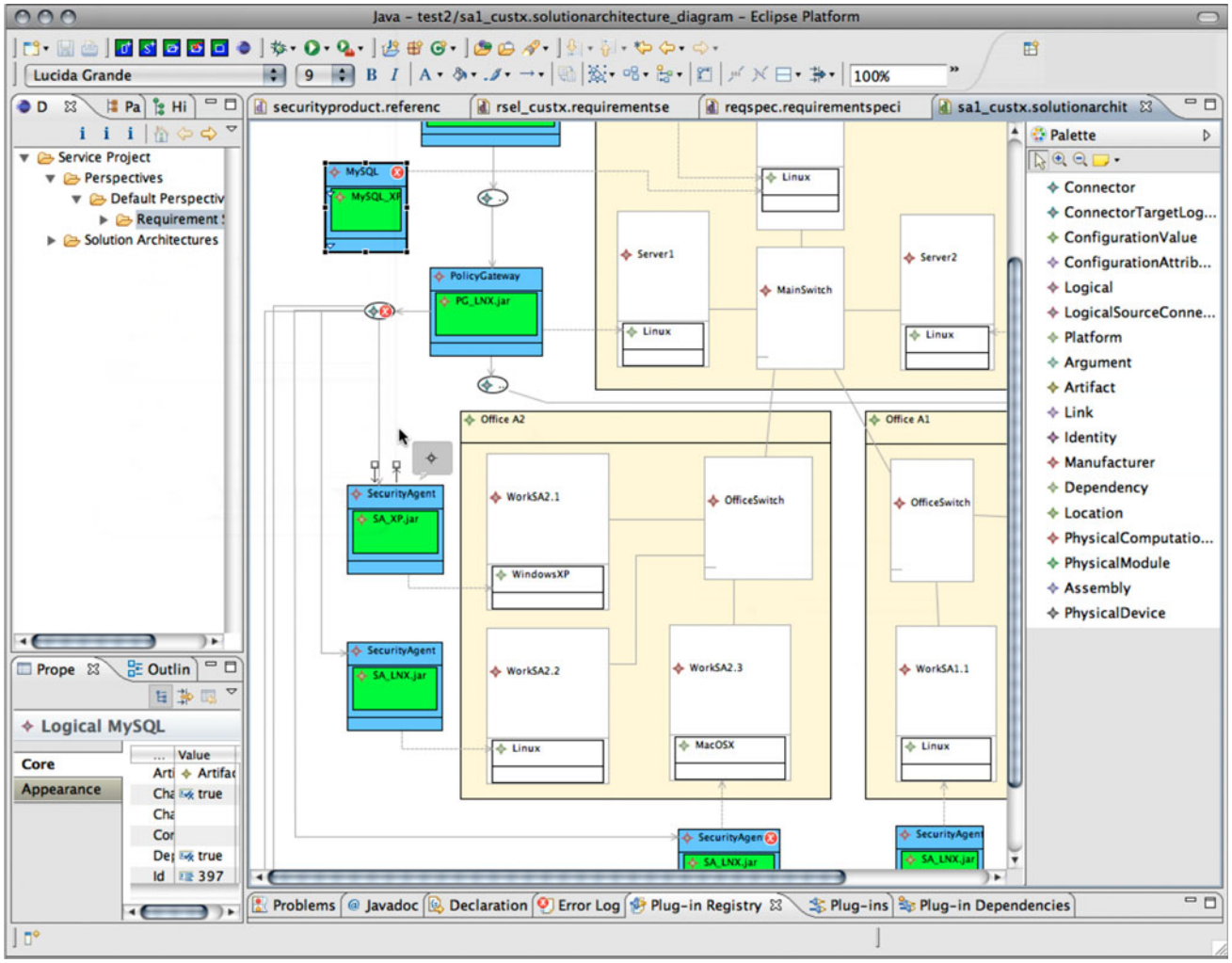

Fig. 25 Screenshot of an early prototype implementation in Eclipse using the EMF framework and GMF

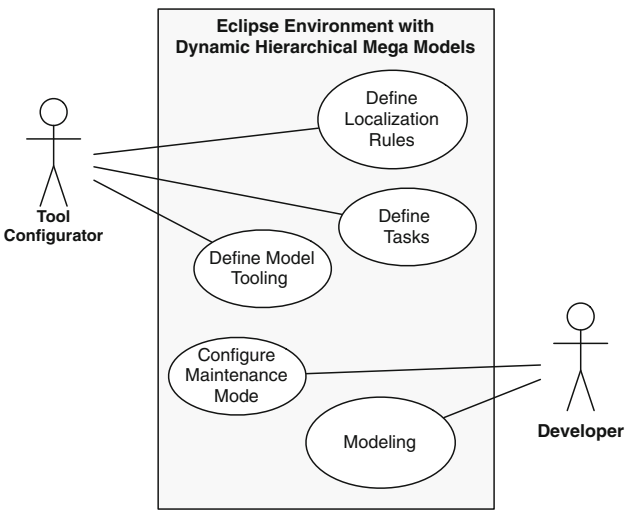

Fig. 26 Use cases for configuring/using an Eclipse-based MDE environment with dynamic hierarchical mega models

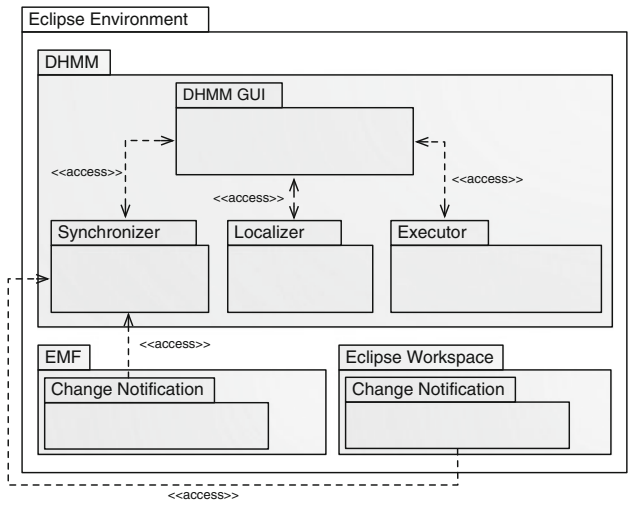

Fig. 27 Prototype architecture

prototype implementation is build on top of Eclipse and uses the extensions EMF[33] and GMF.[34] EMF is used for modeling purposes and GMF for defining a concrete syntax of the EMF models. The prototype consists of four main components as explained in the following:

1. *DHMM GUI*: coordinates all processes of the prototype and provides GUI elements to a developer for using the environment.
2. *Synchronizer*: handles the synchronization of dynamic hierarchical mega models with domain-specific models. It receives change notifications from all registered models and model elements for keeping the representations in a dynamic hierarchical mega model up-to-date instantaneously including modifications, creations and deletions.
3. *Localizer*: is responsible for applying all creation and deletion rules on a given dynamic hierarchical mega model and its referenced models and model elements. In contains the algorithm from Sect. 4.4.
4. *Executor*: handles the creation of a task schedule and further the triggering of tasks when automatically applicable.

6.3 Traceability maintenance modes

The traceability maintenance process triggers the localization and execution process. The actual application of tasks in the execution process can be requested by a developer on demand. We provide three different maintenance modes that can be used in different scenarios. These modes are explained in the following:

1. *Online mode*: the maintenance process is running in short cycles in the background periodically. A developer can always request the latest status and does not have to wait for the system like in the on-demand mode.
2. *On-demand mode*: at certain points in time, a developer demands for maintenance. A developer will have to wait before he/she can continue to work with the models, but receives feedback when required.
3. *Batch mode*: in this mode synchronization, localization and execution are conducted outside the environment in a longer running batch job. To avoid that a developer is blocked until the batch job has been finished, the batch processing is only triggered when a developer is not working with the environment (breaks, over night).

The online and on-demand mode is described in more detail in the following. We do not consider the batch mode explicitly because technically it does not differ from the on-demand mode, except for the triggering interval, which depends on the developer's individual intention. The activity diagram of the online mode is shown in Fig. 28. It is triggered whenever changes are made by a developer. If these changes cause changes in a dynamic hierarchical mega model, the localization and subsequently the execution process are triggered automatically.[35]

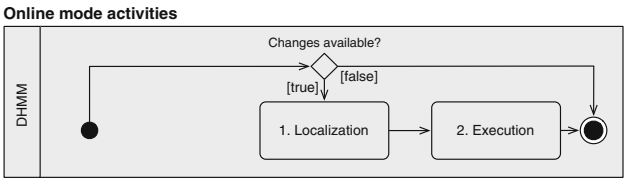

Fig. 28 Activity diagram of traceability maintenance using the online mode

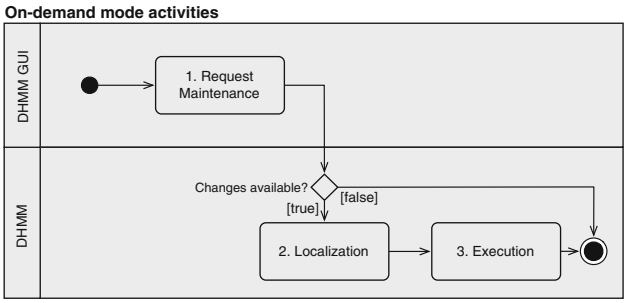

Fig. 29 Activity diagram of traceability maintenance using the on-demand mode

The activity diagram of the on-demand mode is shown in Fig. 29. Here the trigger is explicitly given by a developer through the DHMM GUI. A developer requests maintenance, which then triggers the localization and subsequently the execution process if changes to a dynamic hierarchical mega model are available.

In the following, the activities describing the process of synchronization, localization and execution are explained in more detail.

6.3.1 Step 1: synchronization

Dynamic hierarchical mega models have to be synchronized with modeling artifacts within the MDE environment. The synchronization process is driven by change notifications that are provided by Eclipse and EMF. The change notifications are processed by the synchronization process on-the-fly, which is necessary for maintenance in online mode and the on-demand mode. Thus, the synchronization cannot be triggered by a developer explicitly but is triggered by the system instantaneously.

[33] http://www.eclipse.org/modeling/emf/.

[34] http://www.eclipse.org/modeling/gmf/.

[35] The execution process does not necessarily execute tasks. This depends on the explicit request from the developer.

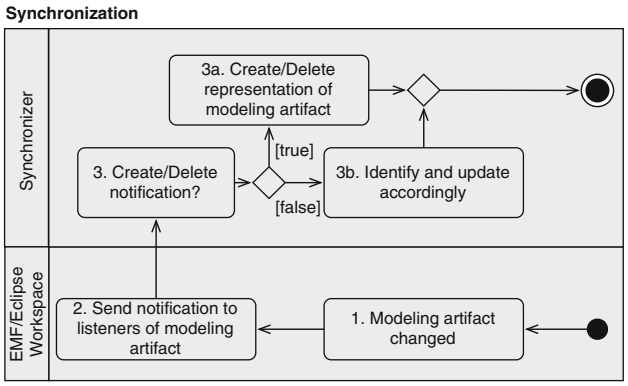

Fig. 30 Activity diagram of the synchronization activity

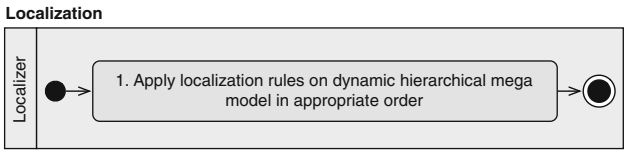

Fig. 31 Activity diagram of the localization activity

Figure 30 shows the internals of the Synchronization activity. When modeling artifacts change[36] these changes are propagated from Eclipse and EMF by means of change notification to the Synchronizer. The Synchronizer then distinguishes between created, deleted and modified modeling artifacts (Activity 3 in Fig. 30). In the case of a created or deleted modeling artifact either a new representation of modeling artifact is created in a dynamic hierarchical mega model or an existing representation of is deleted from a dynamic hierarchical mega model (Activity 3a in Fig. 30). When an existing modeling artifact is modified, its representation within a dynamic hierarchical mega model is updated accordingly (Activity 3b in Fig. 30), e.g., rendering the representation as being modified.

6.3.2 Step 2: localization

In response to any of these changes described in the preceding paragraph, we have to maintain a dynamic hierarchical mega model by executing deletion rules and creation rules in order to maintain all traceability links accordingly because they may become suspicious due to previous changes. The Localization consists of a single activity that applies localization rules in an appropriate order within the Localizer component. This activity is outlined in detail in Fig. 31.

[36] Changes can be the creation, deletion or modification of modeling artifacts.

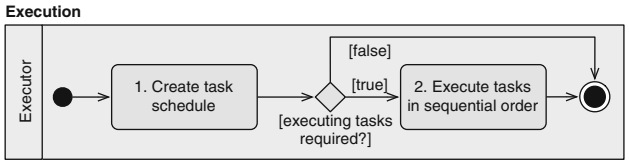

Fig. 32 Activity diagram of the execution activity

6.3.3 Step 3: execution

The Execution activity (see Fig. 32) is relevant whenever at least one obligation link is available in a considered dynamic hierarchical mega model. If this is the case, the Executor component derives a task schedule, which is created in Activity 1. In Activity 2, as many tasks as possible are automatically executed in a sequential order given by Activity 1. However, Activity 2 is only triggered when requested by a developer.

6.4 Summary

In our case study, dynamic hierarchical mega models serve as the basis for executing traceability-maintenance-related activities. Our approach enabled the automation of many consistency and generation issues in our case study. For example, we identified a set of consistency-related issues between the RA and SA models.

7 Evaluation

In this section, we will investigate the scalability of our approach in terms of applicability (see Sect. 7.1) and performance (see Sect. 7.2).

7.1 Applicability evaluation

We applied our approach to a case study in cooperation with CA Labs. The domain of our case study is modeling component-based architectures, concrete configurations and deployments of components and IT infrastructures as deployment targets. Therefore, we defined appropriate tooling within an MDE environment providing EMF models and GMF based editors. We defined several traceability link types concerning inconsistencies between RAs and SAs, deployment between SAs and ITIs, and furthermore traceability link types reflecting the compliance to specified resource constraints.[37] For each of these traceability link types we specified creation rules and derived deletion rules. Additionally, for traceability link types that are responsible for obligation links, we defined task types and implemented simple tasks by means Story Diagrams for operationalization. We were

[37] Not shown in this article.

able to define models within our domain and further automatically maintain traceability information concerning our defined traceability link types automatically whenever possible.

However, our approach is not restricted to the domain of our case study. One could apply the approach to the requirements engineering domain by providing appropriate tooling for this domain, which can be done within EMF and GMF. Dynamic hierarchical mega models automatically adapt to all EMF models that are conform to Ecore and that contain a concept to uniquely identify modeling artifacts. This is required to identify representations in dynamic hierarchical mega models whenever the represented modeling artifacts change.[38] The localization rules are also not bound to a specific domain. When adapting to a new domain, traceability links need to be identified and defined. Furthermore, proper localization rules need to be specified and in certain cases tasks that are used for validating the intention of obligation links. We can reuse traceability link types, localization rules and tasks within the same domain across different MDE environments. Thus, a tool configurator has to specify them only once. Thus, the efforts of migrating our approach to other domains, then the one presented in our case study, is manageable.

In the meantime, we also applied this approach to another research project about the integration of MDE techniques to IT Service Management (ITSM) [24]. The domain of that approach is similar to the domain of the case study in this article. We used dynamic hierarchical mega models to realize Key Performance Indicators (KPIs). KPIs are represented by means of traceability links or a combination of traceability links. The KPIs are automatically located and further their validity is (re-) established by means of tasks. Furthermore, we applied the principle of finding inconsistencies between configuration models and reference models, too.

7.2 Performance evaluation

In this section, we will discuss scalability concerning performance of the processes *Synchronize* (see Sect. 6.3.1), *Localization* (see Sect. 6.3.2) and *Execution* (see Sect. 6.3.3). We further reason about the performance by considering the three different maintenance modes, which are *Online mode*, *On-demand mode* and *Batch mode*, which are explained next.[39] The different maintenance modes result in different performance requirements concerning the expected response time of the synchronization, localization and execution process. Especially, as long as a dynamic hierarchical mega model is not updated accordingly, no modifications, i.e., at the SA, can be done without the risk to create non-resolvable inconsistencies within a dynamic hierarchical mega model [14]. In the on-demand mode the response time must be acceptable (1–15 s) while it must be nearly unnoticeable ($\leq$1 s) in the online mode.

7.2.1 Step 1: synchronization

The first step is the synchronization of a dynamic hierarchical mega model with all modeling artifacts in the MDE environment. In such a way that a dynamic hierarchical mega model is adequately reflecting all modeling artifacts. As explained in Sect. 6.3.1, we use a notification mechanism that reacts as soon as a change event is sent. As the related changes are made *incrementally* only where indicated by the notification events, the synchronization of a dynamic hierarchical mega model happens nearly instantaneously and we noticed that the synchronization scales nearly perfectly. Thus, no further evaluation is conducted for the synchronization process.

7.2.2 Step 2: localization

The second step is the localization of traceability links. In this case, the performance issue is related to the application of creation and deletion rules. Thus, it depends on the size of a dynamic hierarchical mega model, the number of localization rules and the complexity of these rules. In general, the algorithm for applying localization rules on a dynamic hierarchical mega model has impact on the performance (using the simple algorithm vs. the optimized algorithm as presented in Sect. 4). In comparison to both of these algorithms, we evaluate another localization algorithm, which is completely batch oriented. This batch algorithm does not consider any deletion rules and thus deletes all existing traceability information beforehand and then applies all creation rules using the simple algorithm. This comparison is of interest since it shows the performance increase we gain from introducing deletion rules. For our case study, we can conclude that localization is computed rather rapidly since no large-scale models have to be maintained. However, to study scalability also for larger models we used artificial examples for the evaluation to ensure that many rule applications occur. Therefore, we also use artificially generated creation rules and derived deletion rules to focus on scalability issues. Our first measurement parameter is split into four groups. The difference between these groups is the number of localization rules and the constitution of the applied localization rules. The constitution of localization rules refers to the degree of depending localization rules, which is the hierarchy depth between traceability links. For example, a hierarchy depth of three means that there is a top-down or bottom-up dependency across three traceability links.

[38] We omitted this issue because it is a technical aspect.

[39] All evaluations are conducted on an Apple MacBook Pro with a 2.4 GHz Intel Core 2 Duo CPU and 4 GB main memory. We used Eclipse 3.3.2 and JRE 1.5 for executing the scenarios.

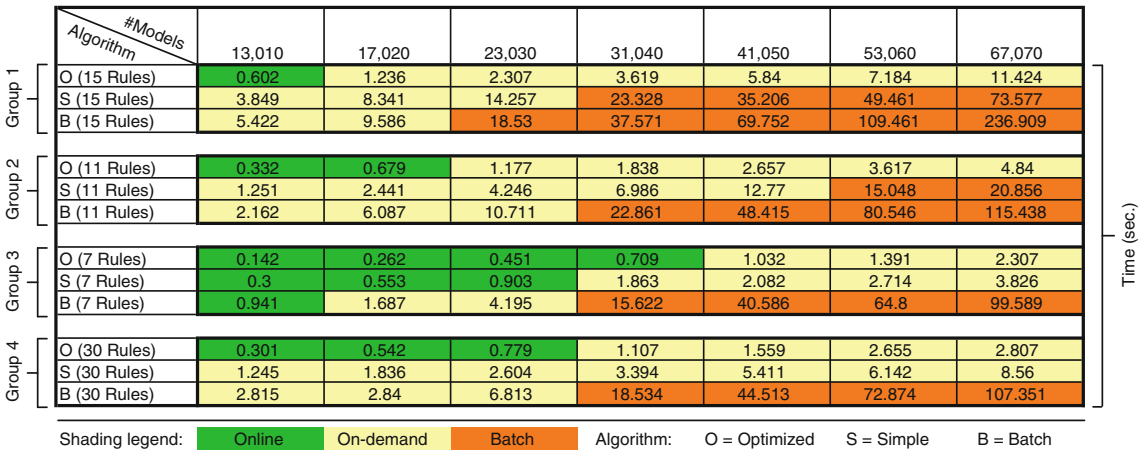

Fig. 33 Evaluation of different localization algorithms using different setups

1. *Group 1*: applies 15 rules. The rules are constituted in a way that there is a hierarchy depth of eight, which is very high in practice. The assumption is that the simple algorithm does not scale very well if the rules are ordered unfavorable, which is important as more rules are dependent.
2. *Group 2*: applies 11 rules with a maximum hierarchy depth of four.
3. *Group 3*: applies 7 rules with a maximum hierarchy depth of two. We assume that all algorithms scale best in this group since the rules do not have many dependencies.
4. *Group 4*: applies 30 rules. However, this case is different to the previous groups because the constitution of the rules is different. This group is made of Group 3 and additional rules of a maximum hierarchy depth of two are added. We assume that the algorithms will not scale worse than in Group 3 since the scalability highly depends on the hierarchy depth and further the number of depending rules.

Obviously, the second parameter that affects the performance is the size of a dynamic hierarchical mega model since all algorithms have to iterate over all elements when applying a localization rule. The following evaluation comprises a dynamic hierarchical mega model of size from 13,010 to 67,070.[40] The size does not contain traceability links and characteristics. The results of the evaluation are shown in Fig. 33. It shows that the algorithms behave as expected. The best performance of all algorithms is reached in Group 3. The optimized algorithm could be used in the online mode up to a dynamic hierarchical mega model size of about 31,040 elements. The simple algorithm does not scale well but can also be used for the online mode up to a size of approximately 23,030. In contrast, the batch algorithm could be used on demand between 17,020 and 23,030 and further only in batch mode. In the case of Group 1, the evaluation shows that only the optimized algorithm is able to be applied in on-demand mode. The simplified algorithm can only be used in on-demand mode if the size of a dynamic hierarchical mega model is smaller than approximately 31,040. When using the batch algorithm without deletion rules, up to 23,030 model elements can be dealt with in on-demand mode. As expected, the scalability of Group 4 is nearly as good as the scalability of Group 3. However, the simplified algorithm can only be applied in on-demand mode. The same tendency holds for the batch algorithm.

7.2.3 Step 3: execution

The execution can result in relevant costs concerning computation as well as manual labor. As the execution related to the traceability links is outside the scope of this work, we cannot make any statement at this point whether or not the external programs or manual activities related to this step do scale. Instead, we only consider the question of whether deriving a schedule for the task is appropriately fast. As the algorithm to derive the schedule is basically topological sort and thus linear to the size of a dynamic hierarchical mega model, no real scalability problems has to be expected even when a dynamic hierarchical mega model becomes quite large. This assessment is based on our experiments where the task scheduling effort was neglectable.

8 Related work

In this section, we present the related work of traceability approaches in the domain of MDE. We focus only on

[40] Please note that the numbers of elements refer to multiple models. According to [15] single models from different industries may in practice have up to 36,000 elements, the maximal considered size therefore relates two large models or multiple smaller ones.

traceability approaches for structured modeling artifacts, focusing on the degree of traceability link maintenance support and the capability to apply traceability to global model management. Therefore, this section is split into approaches within classical traceability that do not support traceability in global model management and approaches explicitly proposing a traceability solution within global model management.

8.1 Classical traceability approaches

In [8] it is illustrated how traceability can support the domain of product-line engineering at a conceptual level by showing the inherent dependencies that exist between models in the product line engineering domain. However, this approach focuses on syntactical issues of traceability models within the product-line engineering domain. The approach does not consider traceability maintenance at all.

Drivalos et al. [13] is an extensive work on the advantages and disadvantages of traceability models. Our dynamic hierarchical mega model is related to their notion of a *general-purpose traceability model*. In contrast to their definition of a traceability model, we distinguish between models and model elements and hierarchical dependencies between them. Their approach uses formally specified Epsilon Validation Language (EVL) constraints to define consistency of traceability links. However, the approach is not used for the maintenance of these links meaning the automatic creation/deletion of traceability links and further (re-)establishing their validity.

In [16] a traceability model is defined, which supports the concept of chains of traceability links (data flows). Therefore, the traceability model has a focus on traces, which consist of an ordered set of steps whereas each step contains traceability links. However, their approach only considers traceability between model elements and not between models. Furthermore, the approach only provides derived traceability links by means of a model transformation. Thus, maintenance is restricted to the initial creation and validation by applying a model transformation that is used to transform source models to target models.

The approach of Vanhooff et al. [42] is closely related to the approach shown in [16]. The contribution of their approach is that a model transformation can take traceability links of previous transformations as input. Thus, the focus of their approach is to feed a model transformation with available traceability links and further incorporate new links, created as a by-product of a model transformation, into a traceability model. Thus, their approach is also restricted to the initial maintenance of traceability links related to a specific model transformation.

Le Dang et al. [30] is about traceability in Accord|UML. The paper shows an approach of how to include requirements into the modeling process and how to identify potential traceability links expressed by the SysML requirements profile. They have proposed a specific structure by means of traceability facilities in SysML, which enables the management of traceability links relating requirement elements to other model elements. However, this approach does not allow links between arbitrary model elements. Traceability links between model elements of different abstraction layers are realized by deriving them from model transformations, which allows to link source elements with corresponding target elements. Thus, this approach does support the initial creation of traceability links, but has no further maintenance support.

In [5] a multi-faceted traceability approach is discussed. Multi-faceted traceability means that traceability links are used to connect development artifacts in different views. The approach proposes to capture traceability links as a side effect of development tasks. Traceability links may also be captured and classified by any kind of formally specified rules, which would enable at least the initial creation of traceability links. In comparison to our approach, they mention subsequent validation activities as we propose in a subsequent execution process. Further, the paper outlines requirements to a proposed traceability approach but no further elaboration.

Aleksy et al. [3] is an approach to trace requirements across different models and levels of abstraction. Therefore, a modeler initially creates traceability links, which represent the modeler's knowledge about the requirements and their realization in the developed system. Their approach automatically creates traceability links by analyzing transitive closures of traceability links. However, it is not clear what happens in later life-cycles of the automatically derived traceability links. Thus, the approach does not provide full maintenance support for traceability links.

The approach in [32] supports automated traceability maintenance by recognizing development activities performed by a modeler. Development activities are formally specified and changes of certain model elements trigger a *LinkUpdateManager*. This manager is responsible for updating traceability links that are related to the changed model elements. In their paper, it is not mentioned how traceability links are actually created and updated. This task is related to any kind of action without any specific formalism.

Aizenbud-Reshef et al. [2] present a framework for defining operational semantics of traceability links for automated traceability link maintenance as well as determining the validity of the traceability links in cases of changes of model elements. They propose to apply the *event-condition-action* triple paradigm to specify the operational semantics of traceability links. However, they do not provide a clear definition of the action semantics of traceability links and further no implementation of the approach exists. Their recommendation is that action semantics should be able to create new model elements, which may affect the existence/validity of other traceability links. Thus, a careful analysis of all

operational semantics would require to proof termination/confluence of the maintenance algorithm.

The approach in [44] extends/refines the approach shown in [2] by providing a detailed traceability model. However, concerning the automated traceability link maintenance, their approach suffers from the same deficiencies as the approach in [2]. There is no detailed concept, implementation or prototype available.

In [12] an approach is shown that specifies a general purpose traceability meta modeling language (TML), which can be used within different MDE tools that support different meta-meta models. Therefore, instances of a TML can be automatically transformed into meta models of any specific meta-meta model (e.g., Ecore or MOF) using ETL. Additionally, a set of constraints is generated using EVL for checking the conformity between instances of the transformed meta model and the transformed meta model itself. Thus, TML enables the construction and maintenance of traceability meta models and accompanying constraints. However, the focus of that paper is on the construction and maintenance of traceability *meta models*. We are focusing on the construction and maintenance of *generic* traceability *models* that are conform to the dynamic hierarchical mega model. Nevertheless, the idea could be applied to have more type safe traceability links and to improve interoperability.

All of the considered approaches above differ in their degree of traceability link maintenance support. It ranges from non-traceability maintenance support [8,13], through approaches that allow initial maintenance of traceability links [3,5,16,30,42] and a more or less overall maintenance of traceability links [2,32,44]. The maintenance support of the approaches [2,44] do only exist in theory whereas in [32] it is not mentioned in sufficient detail. The approach [12] is different because it supports the creation and maintenance of traceability meta models.

8.2 Traceability in global model management

In [7] there is a clear separation of classical traceability (*traceability in the small*) and traceability in global model management (*traceability in the large*). Classical traceability is considered as the ability to define weaving models that are used to relate model elements of different models. The approach should help to *un-pollute* traceability models in global model management by putting traceability-related information into a mega model. Therefore, the common mega model is adapted by replacing *simple* traceability links with traceability models that have source and target relationships to different models in the mega model. In their particular case, a traceability model is a weaving model implementing traceability between models by defining mappings between model elements. Traceability information is further automatically established by weaving instantiated models into a traceability model. Our approach is different because we do not apply weaving as a technique to maintain traceability. Using weaving techniques is restricted regarding maintenance support for traceability links. If the source models change, the whole traceability model needs to be (re-)generated. Furthermore, there is no concept for applying subsequent validation techniques. We do not attempt such a strict separation between classical traceability and traceability in global model management. Our dynamic hierarchical mega model contains models as well as model elements in combination and with explicit hierarchical dependencies.

Salay et al. [38,39] is a sophisticated approach joining classical traceability and traceability in global model management called macro modeling. This is realized by a multi model, which contains models and traceability links in between. A traceability link is defined by a meta model, which contains meta model elements and associations of the meta models it relates to as well as meta model morphisms to map-related meta model elements to meta model elements of the meta model of the traceability model. Traceability links are automatically maintained by checking the meta model morphisms on the instances of the models. Concerning the expressiveness of the technology for automatically maintaining traceability links it is not clear whether existing traceability information can be used as a condition for establishing other traceability links. Furthermore, it is not clear how traceability link maintenance is realized in later life-cycles.

The above approaches support traceability in a global model management context. However, both approaches are restricted within their maintenance support as well as their kind of integrating traceability into global model management. These approaches define individual traceability models for each dependency between models. These models are automatically derived by using weaving technology and further a meta model morphism technique. Thus, both approaches are somehow restricted to the initial traceability link maintenance rather full traceability link maintenance. Thus, these approaches do not consider a subsequent execution phase for (re-)establishing the validity of traceability links. Concerning the integration of their traceability models, these approaches have a strict separation between models and model elements. Thus, there are neither explicit dependencies between models and model elements nor hierarchical traceability links.

9 Conclusions and future work

In this article, we presented a comprehensive traceability approach by means of a combination of high-level traceability models (mega models) and low-level traceability models, which is called dynamic hierarchical mega models. A dynamic hierarchical mega model uses hierarchical dependencies between high-level and low-level modeling artifacts

and between traceability links at different levels to glue both traceability models into a combined traceability model. This combined traceability model leverages reasoning about traceability links by considering different levels of abstraction. We also presented an efficient way of maintaining traceability links on top of dynamic hierarchical mega models by separating maintenance between a localization and execution process. The localization process considers lightweight operations to automatically create and delete traceability information within dynamic hierarchical mega models. We apply endogenous model transformation rules, which are called localization rules. These rules are further distinguished between creation rules and deletion rules to decouple the creation from the deletion. This fine-grained distinction facilitates to also manually, e.g., delete traceability information that was automatically created by a creation rule and contrary. The algorithm that applied these rules to a dynamic hierarchical mega model also deals issues of termination and confluence, which appears in any rule-based system. The execution process considers heavyweight operations to automatically (re)-establish the validity of traceability links, which can be complex model transformation or even manual tasks. Thus, we introduced an abstract concept called task that is not inherently bound to any technology. For these tasks we defined an algorithm that automatically derives accurately ordered list of tasks for a subsequent efficient execution. The prototype demonstrates how the traceability maintenance process is applied. An evaluation has further demonstrated that the approach scales also for larger models.

One limitation of our approach is that it does not allow to integrate external modeling tools with different meta-meta models (e.g., MOF). Currently, we only support Ecore as meta-meta model. However, similar to other approaches [17] and our own works [21,23] we can facilitate adapters that can be employed to overcome this issue and leverage heterogeneity. This conforms to the star integration pattern for the meta-meta model Ecore. Another possibility to improve interoperability is to adapt the idea as introduced in [12].

It is to be noted that the achieved high degree of automation is not always possible. In our specific case study, we were able to derive nearly all traceability links automatically. However, sometimes not enough information is available to automate the creation/deletion of traceability links. In our future work, we will thus study how manual localization and execution can be integrated more smoothly and, which integration works best for the case of heuristic rather than exact rules.

Another limitation of the early prototype are the localization rules and the tasks we applied. We used Story Diagrams as underlying formalism. However, at that point in time we had to generate Java code from Story Diagrams to execute them on dynamic hierarchical mega models. This is a major drawback concerning the flexibility and applicability of our approach because new rules need to be defined externally and further compiled/linked into the MDE environment. In continuative implementations of our approach we integrated a Story Diagram interpreter [22], which improves the flexibility tremendously. Thus, we are able to model localization rules and, in certain cases, also tasks by means of Story Diagrams within the MDE environment and directly execute them without restarting the environment. This is also important whenever we want to provide a library for traceability link types and task types and their related localization rules and tasks because we only need to load the models into the MDE environment and directly apply them. However, improved flexibility comes at a price. Unfortunately, the performance of interpreting Story Diagrams is currently not comparable to executing compiled Java code. Therefore, we plan to integrate incremental approaches [14,43] even for the localization process (synchronization and execution are already incremental).

We further plan to integrate this work with our work on incremental model synchronization with Triple Graph Grammars [25,26] to study how the incremental updates of the derived traceability links can be integrated with the localization via creation and deletion rules for this case.

Additional improvements that are planned are to facilitate the modeling of localization rules by means of a higher-level modeling language to automatically derive operational localization rules. Currently, we have to model operational localization rules by hand.

In [24] we started integrating reporting on top of dynamic hierarchical mega models for improved reasoning about traceability information. We like to extend the analysis capabilities of our approach by additionally defining generic/specific analysis views on top of dynamic hierarchical mega models. For example, we defined an analysis view that highlights high-level model dependencies only. Thus, we can provide additional analysis views on dynamic hierarchical mega models, which additionally increases the value of our approach.

Furthermore, the current task scheduling algorithm makes the assumption that there are no alternative tasks to achieve the same required semantics. However, in practice frequently there are multiple options such as manual procedure or several automated techniques, which can do the job. Therefore, an extended task model, which takes development costs as well as accuracy of the results into account and computes an optimal or nearly optimal schedule, is a valuable goal.

Depending on the artifacts which have been modified, different tasks may be applicable and thus the derivation of a proper schedule should be extended to also cover such cases. In some cases besides simple read arcs and write arcs in the task model may be extended to read/write arcs. We plan to also study, which implications such an extension of the task model requires.

Acknowledgments We thank Ethan Hadar, Irit Hadar, Armir Jerbi and Moran Kupfer supporting our research at CA, Israel. This research was partially funded by CA Labs, CA Inc. In addition, we want to thank the students Mark Liebetrau and Dusting Lange, who helped to realize the prototype, and Sebastian Waetzoldt for reading an early version of this article carefully. Finally, we thank Gregor Gabrysiak for proof reading earlier versions of the article and all reviewers for their fruitful comments, which helped to improve the quality of this article.

Appendix A: Formal foundations for localization

A.1 Graph transformation systems

We use graph transformation systems (GTS) [37] for attributed graphs as underlying formal model that is sufficient to our needs. To keep the definitions short, we restrict our definitions to the case without attributes and refer to the literature for a more detailed introduction. A graph G is a pair (V, E) with V a set of vertices and $E \subseteq V \times V$ a set of edges. In addition, a set of types $\mathcal{T}$ and a function $t : V \cup E \to \mathcal{T}$ assigning types to all elements is assumed. In our setting the types $\mathcal{T}$ and the function t can be derived from the given meta models. A graph G' *matches* a graph G, if there exists an isomorphic function *iso* that maps all elements of G' to elements of G of the same type. We write $G \approx_{iso} G'$ respectively $G \approx G'$ if the specific *iso* does not matter. A graph pattern $P = (P^+, P^-)$ is a pair of graphs P^+ and P^- with $P^+ \subseteq P^-$. P *matches* a graph G, if there exists an isomorphic function *iso* that maps all positive elements of P (P^+) to elements of G of the same type and no isomorphic function *iso'* exists which extends *iso* and map at least one negative element of P (P^- but not in P^+) to elements of G of the same type. We write $P \subseteq_{iso} G$ and have $iso(P^+) \subseteq G$. A graph transformation rule $r = ((L^+, L^-), R)$ consists of an LHS in the form of a graph pattern (L^+, L^-) and an RHS R. A graph transformation rule $r = ((L^+, L^-), R)$ is *applicable* to a graph G if G matches (L^+, L^-) and when no rule with higher priority can be applied. The set of all possible matches for a rule r and graph G can be computed as depicted in Listing 4.

```
1  procedure match()
2  parameter: G, r = ((L+, L−), R)
3  return {iso|∃iso : (L+, L−) ⊆iso G}
```

Listing 4 Computing all matches

During the *application* of a rule $r = ((L^+, L^-), R)$ to a graph G, the elements that are in L^+ but not in R are removed from G, and elements that are in R but not in L^+ are added to G. The application of a rule r at a graph G for a given mapping *iso* is described in Listing 5.

```
1  procedure apply()
2  parameter: G, r = ((L+, L−), R), iso
3    choose iso' with iso'(L+) = iso(L+) and iso'(R \ L+) ∩ G = ∅
4    G' := G \ iso(L+ \ R) ∪ iso'(R)
5  return G'
```

Listing 5 Apply a rule at a given match

A *graph transformation system* $S = (\mathcal{R})$ consists of a set of graph transformation rules $\mathcal{R}$, defining all possible transformations in the transformation system. The state of S is a graph G typed via t. Given a set of rules $\mathcal{R}$ and a start graph G, which is the current state of S before applying the rules $\mathcal{R}$, Listing 6 provides an algorithm that transfers G into a successor state G'' of S by means of a fix point calculation. G' is reached when the algorithm terminates.

```
1  procedure fixPoint()
2  parameter: G, R
3  boolean changed := true
4  while (changed) do
5    changed := false
6    forall (r ∈ R)
7      while (match(G, r) ≠ ∅) do
8        choose iso ∈ match(G, r)
9        G' := apply(G, r, iso)
10       if (G' ≠ G) //did changes occur?
11         changed := true
12         G := G'
13       endif
14     endwhile
15   endforall
16 endwhile
17 return G'' = G
```

Listing 6 Apply rules until a fix point has been reached

A.2 Termination

Definition 1 (*Strong Monotonous Decreasing*) A set of deletion rules $\mathcal{R}_d$ is *strong monotonous decreasing* with respect to a subset type set $\mathcal{T}' \subseteq \mathcal{T}$ iff all deletion rules $r \in \mathcal{R}_d$ delete more instances of $\mathcal{T}'$ than they create.

Lemma 1 *For a strong monotonous decreasing set of deletion rules $\mathcal{R}_d$ for a subset type set $\mathcal{T}' \subseteq \mathcal{T}$ holds that the repeated application until a fix point has been reached always terminates.*

Proof For each deletion rule application the overall number of instances of $\mathcal{T}'$ decrease. Therefore, any possible series of deletion rule applications must end after finite many steps as the number of initial instances of $\mathcal{T}'$ is finite and decreases with every rule application. □

Definition 2 (*Exclusive Graph Pattern*) An *exclusive graph pattern* for a rule r is a graph pattern p contained in the precondition of r such that for any match of that graph pattern only one successful application of that rule is possible.

Definition 3 (*Excluded Graph Pattern*) An *excluded graph pattern* for a rule r is a graph pattern p such that the post-condition of r cannot contribute to any new match for p.

Lemma 2 *For a rule r, a rule set $\mathcal{R}$ with $r \in \mathcal{R}$ and a graph pattern p holds for any graph G that no sequence with infinite many rule applications of r can exist, when p is an exclusive graph pattern for r and p is an excluded graph pattern for all $r' \in \mathcal{R}$.*

Proof Assuming that $M = match(G, r)$ is the finite set of matches for a graph pattern p of a rule r and a start graph G. As p is an excluded graph pattern for all rules $r' \in \mathcal{R}$, no rules in $\mathcal{R}$ can result in a new match for p. In addition, r itself ensures that its application for any match of p modifies the resulting graph in such a manner that the considered match is not valid for r any more. Therefore, the overall number of matches for p, which is a superset of the matches for r, decreases strong monotonically with each application of r and thus there can be only finite many rule applications of r. □

A.3 Uniqueness of the result

Definition 4 (*Critical Pair* [27]) Two rules r and r' are *parallel independent* if the application of r for one match cannot disable a match for r' and vice versa.

A pair of rules $(r, r') \in \mathcal{R}^2$ is a *critical pair* if a graph G and a match for both rules for G exists such that the application is not parallel independent.

Lemma 3 *A set of rules $\mathcal{R}$ describes a local confluent system if all critical pairs are strongly confluent.*

Proof See [27] for the proof. □

A.4 Optimized localization algorithm

Definition 5 (*Dependence of Rules*) Given two rules r_1 and r_2 we say that the r_2 *depends directly* on r_1 (written $r_1 \leftarrow_d^1 r_2$) if r_2 does have required vertices or edges in its pre-condition, which are created by r_1 or r_2 or does have forbidden vertices or edges in its pre-condition, which are deleted by r_1. We further define $\leftarrow_d^{i+1}$ as $\{(r, r') | (r, r') \in \leftarrow_d^i \vee \exists r'' : (r, r'') \in \leftarrow_d^i (r'', r') \in \leftarrow_d^i\}$ and define the *depends* ($\leftarrow_d$) as $\bigcup_{i=1}^{\infty} \leftarrow_d^i$.

Lemma 4 *For two rules r and r' with $r' \not\leftarrow_d r$ holds that first executing the rule r until a fix point has been reached and then the rule r' until a fix point has been reached results in the same outcome as the execution of both rules in an arbitrary ordering until a fix point has been reached* (*if the application has a unique result and terminates at all*).

Proof As r has no elements in its pre-condition which are affected by r' due to $r' \not\leftarrow_d r$, any execution of r' can be done after all executions of r. □

Definition 6 (*Dependence of Rule Sets*) Given a set of rules sets $\{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$ with $\mathcal{R}_i \subseteq \mathcal{R}$, we define for any $i \neq j$ that $\mathcal{R}_i$ *not depends* on $\mathcal{R}_j$ (written $\mathcal{R}_i <_d \mathcal{R}_j$) if no $r_i \in \mathcal{R}_i$ and $r_j \in \mathcal{R}_j$ exists with $r_j \leftarrow_d r_i$.

Lemma 5 *For two sets rules $\mathcal{R}$ and $\mathcal{R}'$ with $\mathcal{R} <_d \mathcal{R}'$ holds that first executing the rules of $\mathcal{R}$ until a fix point has been reached and then the rules of $\mathcal{R}'$ until a fix point has been reached results in the same outcome as the execution of the rule set $\mathcal{R} \cup \mathcal{R}'$ until a fix point has been reached* (*if the application has a unique result and terminates at all*).

Proof As no rule $r \in \mathcal{R}$ has elements in its pre-condition which are affected by any $r' \in \mathcal{R}'$ due to $r' \not\leftarrow_d r$ as implied by $\mathcal{R} <_d \mathcal{R}'$ we can use Lemma 4 to see that we can first consider all rules of $\mathcal{R}$ independent of those of $\mathcal{R}'$ until a fix point has been reached. □

Definition 7 (*Dependency-related Ordering*) A list of rules sets $\langle \mathcal{R}_1, \ldots, \mathcal{R}_n \rangle$ is a *valid ordering* of $\mathcal{R}$ iff $\mathcal{R}_i \subseteq \mathcal{R}$, $\mathcal{R}_1 \cup \cdots \cup \mathcal{R}_n = \mathcal{R}$, $\mathcal{R}_i \cap \mathcal{R}_j = \emptyset$ for any $i < j$ holds $\mathcal{R}_i <_d \mathcal{R}_j$.

Lemma 6 *For a valid ordering $\mathcal{R}_1, \ldots, \mathcal{R}_n$ of $\mathcal{R}$ holds that executing the rules of $\mathcal{R}_i$ until a fix point has been reached for each $i = 1, \ldots, n$ one after another results in the same outcome as the execution of the rule set $\mathcal{R}$ until a fix point has been reached* (*if the application has a unique result and terminates at all*).

Proof Our claim results from the repeated application of Lemma 5 for a step-wise decomposition of $\mathcal{R}$ into $\mathcal{R}_1, \ldots, \mathcal{R}_n$. □

Appendix B: Formal foundations for execution

B.1 Task model

For a task $v_t \in V_t$, we can define the pre-condition as $Pre(v_t) := \{v_a \in V_a | (v_a, v_t) \in E_r\}$ and post-condition as $Post(v_t) := \{v_a \in V_a | (v_t, v_a) \in E_w\}$. We say that the pre-condition of an update task $v_t \in V_t$ is fulfilled by a given set of artifacts V' if $Pre(v_t) \subseteq V'$. In addition, we have the relevant artifacts $V_a^r \subseteq V_a$, which have to be up-to-date[41] when necessary, a set of manually modified artifacts V_a^m (see modified flag of a traceability node), a possibly empty set of not-yet-edited artifacts V_a^e (they are empty and have never been edited), and a possible empty set of changed artifacts $V_a^c \subseteq V_a$, which have been modified since the last update round. We can further assume that $V_a^c \cup V_a^m$ and V_a^e are

[41] Usually, this are all traceability nodes of a dynamic hierarchical mega model.

disjoint ($(V_a^c \cup V_a^m) \cap V_a^e = \emptyset$) as we can exclude that an artifact in V_a^e has been changed or manually modified at the same time. For an initial dynamic hierarchical mega model, the set of initially provided artifacts equals the changed ones V_a^c. The remaining ones $V_a \setminus V_a^c$ are in the set of empty artifacts V_a^e. When deriving the outlined task model from a dynamic hierarchical mega model, the bipartite nature is guaranteed by construction. However, guaranteeing that the task model does not contain cycles, which would imply an infinite sequence of updates, has to be checked by topological sort in $O(|V| + |E|)$ steps.

B.2 Task schedule

A sequence $w \in {V_t}^*$ is a *schedule* iff for $w = v_{t_1} \circ \cdots \circ v_{t_n}$ holds for any $i \neq j$ that $v_{t_i} \neq v_{t_j}$. For a schedule $w \in {V_t}^*$ with $w = w' \circ v_t$ we can define the post-condition recursively for a given set of artifacts V' as $Post_*(w' \circ v_t, V') := \{v_a \in V_a | (v_t, v_a) \in E_w\} \cup Post_*(w', V')$ and $Post_*(\epsilon, V') := V'$ for the empty word. A *valid schedule* for a given set of artifacts V is then a schedule $w \in {V_t}^*$ such that for any decomposition into a task sequence, a task node and a task sequence w_1, v_t, w_2 such that $w_1 \circ v_t \circ w_2 = w$ holds

$$Pre(v_t) \subseteq Post_*(w_1, V) \wedge Post(v_t) \cap Post_*(w_1, V) = \emptyset.$$

We write $V \rightarrow_w V'$ to denote that w is a valid schedule leading from V to $V' = Post_*(w, V)$ or only $V \rightarrow_w$ if V' does not matter. For a valid schedule essentially holds that no task is executed, which inputs have not been properly updated before, and that no artifact is updated twice or that it is updated when it was already up-to-date before. A schedule $w \in {V_t}^*$ is a *complete schedule* for a given set of changed artifacts V_a^c, empty artifacts V_a^e and required artifacts V_a^r iff

$$\begin{aligned} &\exists V \subseteq V_a \setminus V_a^e, V' \subseteq V_a \setminus (V_a^c \cup V_a^e) : \\ &\qquad V \cup V' \supseteq V_a^m \wedge V \rightarrow_w \wedge \\ &V' \rightarrow_{w'} \wedge V_a^r \subseteq Post_*(w, V) \uplus Post_*(w', V'). \end{aligned}$$

While w denotes the schedule required to propagate changes, w' denotes a complete schedule, which is disjoint from w. The condition $V \cup V' \subseteq V_a^m$ together with $Post_*(w, V) \uplus Post_*(w', V')$ (which implies $Post_*(w, V) \cap Post_*(w', V') = \emptyset$) ensures that the sequence does not overwrite any manually added artifact. A complete schedule $v_t^1 \circ \cdots \circ v_t^n$ is *minimal* if we cannot derive another complete schedule by erasing any task. If, in the case of an update, no complete schedule can be found, the available tasks are not sufficient to derive the required updates. This can only be the case, when a dynamic hierarchical mega model exists for which a relevant artifact cannot be generated by any task, when the list of non-empty artifacts is simply not sufficient to derive all relevant artifacts or when a conflict to overwrite a manual-added artifact could not be avoided.

B.3 Compute task schedule

As a traceability link has only a single task assigned and do not overlap concerning their written artifacts, it holds that all artifacts can only be derived by a single task ($\forall v_{t_1}, v_{t_2} : Post(v_{t_1}) \cap Post(v_{t_2}) = \emptyset$). For this case computing a valid schedule is rather straight forward: To compute a complete schedule for a given task model (V, E), a set of relevant artifacts V_a^r, a set of empty artifacts V_a^e, a set of changed artifacts V_a^c and a set of manually modified artifacts V_a^m, we proceed in two phases: (1) We propagate through the graph the impact of changes by a topological traversal such that all artifacts, which need an update, are marked and compute all tasks, which execution is possible. (2) Then, we determine a schedule of tasks, which do the required updates.

```
1  procedure TaskScheduling()
2  parameter: BTM = (V, E), V_a^r, V_a^e, V_a^c, V_a^m
3  //(1) propagate the impact of changes
4  V_a^n := ∅ // artifacts that need an update
5  E_w^n := ∅ // write edges that need an update
6  V_a^u := V_a \ V_a^e // required artifacts
7  V_a^d := ∅ // set of relevant artifacts that need an update
8  V_t^u := ∅ // tasks that can be executed
9  (v_1, ..., v_n) := topologicalSort(BTM);
10 for v in v_1, ... v_n do
11  // compute need to update
12  if (v ∈ V_t and ∃(v', v) ∈ E_r : v' ∈ V_a^n) then
13    E_w^n := E_w^n ∪ {(v, v') ∈ E_w}
14  elseif (v ∈ V_a) then
15    if (v ∈ V_a^c or ∀(v, v') ∈ E_w : (v, v') ∈ E_w^n)) then
16      V_a^n := V_a^n ∪ {v}
17    endif
18    if (v ∈ V_a^m or (v ∈ V_a^c and ∀(v, v') ∈ E_w : (v, v') ∈ E_w^n)) then
19      print("Warning:update vs. manual/changed")
20    endif
21  endif
22  // compute whether task can be executed
23  if (v ∈ V_t and ∀(v', v) ∈ E_r : v' ∈ V_a^u and
          ∀(v, v') ∈ E_w : v' ∉ V_a^c ∪ V_a^m)
24    then
25      V_t^u := V_t^u ∪ {v}
26      V_a^u := V_a^u ∪ {v'|(v, v') ∈ E_w}
27  endif
28 endfor
29 //(2) determine task list
30 w := ϵ // list of task nodes (schedule); initially empty
31 V_a^d := V_a^n ∩ V_a^r
32 for v in v_n, ... v_1 do
33  if (v ∈ V_a^d) then
34    if (∃v' ∈ V_t^u with (v', v) ∈ E_w) then
35      // v' is unique due to ∀v_{t_1}, v_{t_2} : Post(v_{t_1}) ∩ Post(v_{t_2}) = ∅
36      w := v' ∘ w
37      V_a^d := V_a^d ∪ {v''|(v'', v') ∈ E_r}
38    else
39      // no solution exists
40      return ϵ;
41    endif
42  endif
43 endfor
44 return w;
```

Listing 7 Task-Scheduling algorithm

In Listing 7 line 3–7 it is outlined how we propagate the impact of changes through the graph by a topological traversal such that the write arcs are marked as needs-update. They are marked iff at least one artifact, read by the source task, is marked with needs-update or changed. An artifact

is marked as needs-update iff all write arcs pointing to it are marked. If an artifact has to be updated, which has been changed or which has been manually modified, we print out a warning. In parallel we compute which tasks can be executed at all to exclude that they require reading an empty model or trying to write on a changed or manually modified artifact. In $O(|V| + |E|)$ we can thus compute which artifacts have to be updated. To determine the schedule we select the tasks to be executed in a backward traversal starting from the artifacts, which require an update as described in line 28–42 of Listing 7. We start by looking only into the artifacts that are relevant and need an update, as if a relevant element needs no update. It is not necessary to compute an update. V_a^d is initialized as the set of all relevant artifacts, which are marked as needs-update, and the schedule S is initialized as ϵ. We then consider the topologically greatest artifact in V_a^d and select the unique task v' to derive it. We then add v' to the schedule, remove the considered artifact from V_a^d and add all elements to V_a^d, which are required to execute v'. In $O(|V| + |E|)$ we can thus computed a complete schedule of tasks, which can be executed. If the algorithm does not find a solution, then there is none. Moreover, the computed solution is necessarily optimal as each selected task is also required.

References

1. Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., Shaham-Gafni, Y.: Model traceability. IBM Syst. J. **45**(3), 515–526 (2006)
2. Aizenbud-Reshef, N., Paige, R.F., Rubin, J., Shaham-Gafni, Y., Kolovos, D.S.: Operational semantics for traceability. In: ECMDA-TW'05: Proceedings of 1st Workshop on Traceability, Nurnberg, Germany, pp. 7–14. SINTEF, 7–10 November 2005
3. Aleksy, M., Hildenbrand, T., Obergfell, C., Schwind, M.: A pragmatic approach to traceability in model-driven development. In: Heinzl, A., Appelrath, H.-J., Sinz, E.J. (eds.) PRIMIUM. CEUR Workshop Proceedings, vol. 328. CEUR-WS.org (2008)
4. Allilaire F., Bézivin J., Brunelière H., Jouault F.: Global model management in Eclipse GMT/AM3. In: Proceedings of the Eclipse Technology eXchange Workshop (eTX) at ECOOP'06 (2006)
5. Asuncion, H.U.: Towards practical software traceability. In: ICSE Companion '08: Companion of the 30th International Conference on Software Engineering, pp. 1023–1026. ACM, New York (2008)
6. Asuncion, H.U., François, F., Taylor, R.N.: An end-to-end industrial software traceability tool. In: ESEC-FSE '07: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, pp. 115–124. ACM, New York (2007)
7. Barbero, M., Fabro, M.D.D., Bézivin, J.: Traceability and provenance issues in global model management. In: Oldevik, J., Olsen, G.K., Neple, T. (eds.) ECMDA-TW'07: Proceedings of 3rd Workshop on Traceability, Haifa, Israel, pp. 47–55, June 2007. SINTEF (2007)
8. Berg, K., Bishop, J., Muthig, D.: Tracing software product line variability: from problem to solution space. In: SAICSIT '05: Proceedings of the 2005 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries, pp. 182–191. South African Institute for Computer Scientists and Information Technologists, Republic of South Africa (2005)
9. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the large and modeling in the small. In: Model Driven Architecture. Lecture Notes in Computer Science (LNCS), vol. 3599, pp. 33–46. Springer, Berlin (2005)
10. Bézivin, J., Jouault, F., Valduriez, P.: On the need for megamodels. In: Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (2004)
11. Dömges, R., Pohl, K.: Adapting traceability to project-SP. Commun. ACM **41**(12), 54–62 (1998)
12. Drivalos, N., Kolovos, D.S., Paige, R.F., Fernandes, K.: Engineering a DSL for software traceability. In: First International Conference, SLE 2008, Toulouse, France. Lecture Notes in Computer Science (LNCS), vol. 5452, pp. 151–167, 29–30 September 2009. Springer, Berlin (2009)
13. Drivalos, N., Paige, R.F., Fernandes, K., Kolovos, D.S.: Towards rigorously defined model-to-model traceability. In: ECMDA-TW'08: Proceedings of 4th Workshop on Traceability, Berlin, Germany. SINTEF, 9–12 June 2008
14. Egyed, A.: Instant consistency checking for the UML. In: ICSE '06: Proceedings of the 28th International Conference on Software Engineering, pp. 381–390. ACM, New York (2006)
15. Egyed, A.: Fixing inconsistencies in UML design models. In: Proceedings of the 29th International Conference on Software Engineering (ICSE), pp. 292–301, May 2007. IEEE Computer Society Press, Minneapolis (2007)
16. Falleri, J.R., Huchard, M., Nebut, C.: Towards a traceability framework for model transformations in Kermeta. In: ECMDA-TW'06: Proceedings of 2rd Workshop on Traceability, Bilbao, Spain. SINTEF, 10–13 July 2006
17. Farkas, T., Hein, C., Ritter, T.: Automatic evaluation of modelling rules and design. In: 2nd Workshop "From Code Centric to Model Centric Software Engineering: Practices, Implications and ROI". Bilbao, Spain, 10–13 July 2006
18. Favre, J.-M.: Towards a basic theory to model driven engineering. In: 3rd Workshop on Software Model Engineering (WISME), Lisboa, Portugal (2004)
19. Finkelstein, A.: A foolish consistency: technical challenges in consistency management. In: DEXA '00: Proceedings of the 11th International Conference on Database and Expert Systems Applications. Lecture notes in computer science (LNCS), vol. 1873, pp. 1–5. Springer, London (2000)
20. France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: FOSE '07: 2007 Future of Software Engineering, pp. 37–54. IEEE Computer Society Press, Washington (2007)
21. Giese, H., Hildebrandt, S., Neumann, S.: Towards integrating SysML and AUTOSAR modeling via bidirectional model synchronization. In 5th Workshop on Model-Based Development of Embedded Systems (MBEES) (2009)
22. Giese, H., Hildebrandt, S., Seibel, A.: Improved flexibility and scalability by interpreting story diagrams. In: Magaria, T. Padberg, J., Taentzer, G. (eds.) Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009), vol. 18. Electronic Communications of the EASST (2009)
23. Giese, H., Meyer, M., Wagner, R.: A prototype for guideline checking and model transformation in Matlab/Simulink. In: Giese H., Westfechtel, B. (eds.) Proceedings of the 4th International Fujaba Days 2006, Bayreuth, Germany. Technical Report, vol. tr-ri-06-275, pp. 56–60. University of Paderborn, Paderborn (2006)
24. Giese,. H., Seibel, A., Vogel, T.: A model-driven configuration management system for advanced IT service management. In:

Bencomo, N., Blair, G., France, R., Jeanneret, C., Munoz, F. (eds.) Proceedings of the 4th International Workshop on Models@run.time at the 12th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MoDELS 2009), Denver, Colorado, USA. CEUR Workshop Proceedings, vol. 509, pp. 61–70, October 2009
25. Giese, H., Wagner, R.: Incremental model synchronization with triple graph grammars. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Genova, Italy. Lecture Notes in Computer Science (LNCS), vol. 4199, pp. 543–557. Springer, Berlin (2006)
26. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Softw. Syst. Model. **8**(1), 3 (2009)
27. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: First International Conference, ICGT 2002 Barcelona, Spain. Lecture Notes in Computer Science (LNCS), vol. 2505, pp. 161–176, 7–12 October 2002. Springer, London (2002)
28. IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology (2004)
29. Köhler, H.J., Nickel, U.A., Niere, J., Zündorf, A.: Integrating UML diagrams for production control systems. In: Proceedings of the 22nd International Conference on Software Engineering (ICSE), pp. 241–251. ACM, Limerick (2000)
30. Le Dang, H., Dubois, H., Gérard, S.: Towards a Traceability model in a MARTE-based methodology for real-time embedded systems. Innov. Syst. Softw. Eng. **4**(3), 189–193 (2008)
31. Limón, A.E., Garbajosa, J.: The need for a unifying traceability scheme. In: Oldevik, J., Aagedal, J. (eds.) ECMDA-TW'05: Proceedings of 1st Workshop on Traceability, Nurnberg, Germany, pp. 47–56. SINTEF, 7–10 November 2005
32. Mäder, P., Gotel, O., Philippow, I.: Enabling automated traceability maintenance by recognizing development activities applied to models. In: 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 49–58. IEEE Computer Society Press, New York (2008)
33. Mens, T., Gorp, P.V.: A taxonomy of model transformation. In: Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005), vol. 152. Elsevier, Amsterdam (2006)
34. Paige, R.F. Olsen, G.K., Kolovos, D.S., Zschaler, S., Power, C.: Building model-driven engineering traceability classifications. In: ECMDA-TW'08: Proceedings of 4th Workshop on Traceability, Berlin, Germany. SINTEF, 9–12 June 2008
35. Plump, D.: Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence. pp. 201–213. Wiley, Chichester (1993)
36. Ráth, I., Bergmann, G., Ökrös, A., Varró, D.: Live model transformations driven by incremental pattern matching. In: Proceedings of the 1st International Conference on Model Transformation (ICMT), 2008, Zürich, Switzerland. Lecture Notes in Computer Science (LNCS), vol. 5063, pp. 107–121, 1–2 July 2008. Springer, Berlin (2008). Zeigt effizientes Pattern Matching auf Basis von RETE-Networks. Schränkt Suchraum ein
37. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation: Foundations, vol. 1. World Scientific, Singapore (1997)
38. Salay, R. Mylopoulos, J. Easterbrook, S.: Managing models through macromodeling. In: ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 447–450. IEEE Computer Society, Washington, DC (2008)
39. Salay, R., Mylopoulos, J., Easterbrook, S.: Using macromodels to manage collections of related models. In: 21st International Conference, CAiSE 2009, Amsterdam, The Netherlands. Lecture Notes in Computer Science (LNCS), vol. 5565, pp. 141–155, 8–12 June 2009. Springer, Berlin (2009)
40. Spanoudakis, G., Zisman, A.: Software traceability: a roadmap. In: Handbook of Software Engineering and Knowledge Engineering: Recent Advances, 3rd edn, pp. 395–428. World Scientific, Singapore (2005)
41. Van Gorp, P., Altheide, F., Janssens, D.: Traceability and fine-grained constraints in interactive inconsistency management. In ECMDA-TW'06: Proceedings of 2rd Workshop on Traceability, Bilbao, Spain. SINTEF, 10–13 July 2006
42. Vanhooff, B., Van Baelen, S. Joosen, W., Berbers, Y.: Traceability as input for model transformations. In: ECMDA-TW'07: Proceedings of 3rd Workshop on Traceability, Haifa, Israel, pp. 37–46. SINTEF, 11–15 June 2007
43. Wagner, R., Giese, H., Nickel, U.: A plug-in for flexible and incremental consistency management. In: Proceedings of the Workshop on Consistency Problems in UML-based Software Development at the UML Conference 2003, San Francisco, USA, Technical Report. Blekinge Institute of Technology (2003)
44. Walderhaug, S., Johansen, U., Stav, E., Aagedal, J.: Towards a generic solution for traceability in MDD. In: Neple, T., Oldevik, J., Aagedal, J. (eds.) ECMDA-TW'06: ECMDA Traceability Workshop, Bilbao (Spain). SINTEF, 10–13 July 2006

Author Biographies

Andreas Seibel studied computer science at the University of Paderborn, Germany. He was a student assistant between 2004 and 2007 at the software engineering group. In 2007 he graduated at the University of Paderborn. Subsequently he gets a PhD Student at the Hasso Plattner Institute for Software Systems Engineering at the University Potsdam under supervision of Prof. Dr. Holger Giese. His research interests as a PhD Student are model management, model-based analysis, traceability in model-driven engineering and model transformations.

Stefan Neumann studied computer science at the University of Paderborn, Germany. He received the degree Bachelor of Computer Science in 2005 and the degree Dipl.-Inform in 2007 at the University of Paderborn. Since 2007 he is a PhD Student and Research Assistant at the Hasso Plattner Institute in Potsdam, Germany, at the group of Prof. Holger Giese. His research interests include model-based development of embedded-systems, model transformation/synchronization as well as model management using graph transformation techniques.

Holger Giese is a full professor at the Hasso Plattner Institute for Software Systems Engineering at the University Potsdam. Beforehand he was assistant professor for object-oriented specification of distributed systems in the Software Engineering Group of the University of Paderborn since 2001. He studied technical computer science at the University Siegen and received his engineering degree in October 1995. He received a doctorate in Computer Science at the Institute of Computer Science at the University of Münster in February 2001. His research focus is the model-driven development of software-intensive systems covering the specification of dynamic and flexible systems by services, collaborations, patterns, and components, approaches to analyze and formally verify such models, and approaches for model synthesis. The main focus are systems that are typically distributed systems, embedded real-time systems as well as systems that are capable to adapt and coordinate themselves. Furthermore, he does research on model transformation, concepts for generating source code for structure and behavior, and the general problem of model integration during the process of model-driven development. He is member of the Association for Computing Machinery, the IEEE Computer Society, and the German Informatics Society.