sosym manuscript No.
(will be inserted by the editor)

Rigorous Identification and Encoding of Trace-Links in Model-Driven Engineering

Richard F. Paige, Nikolaos Drivalos, Dimitrios S. Kolovos, Kiran J. Fernandes, Christopher Power[1], Goran K. Olsen[2], Steffen Zschaler[3]

[1] University of York, UK
[2] SINTEF, Norway
[3] Lancaster University, UK

Received: date / Revised version: date

Abstract Model-Driven Engineering (MDE) involves the construction and manipulation of many models of different kinds in an engineering process. In principle, models can be used in the product engineering lifecycle in an end-to-end manner for representing requirements, designs and implementations, and assisting in deployment and maintenance. The manipulations applied to models may be manual, but they can also be automated—for example, using model transformations, code generation, and validation. To enhance automated analysis, consistency and coherence of models used in an MDE process, it is useful to identify, establish and maintain *trace-links* between models. However, the breadth and scope of trace-links that can be used in MDE is substantial, and managing trace-link information can be very complex. In this paper, we contribute to managing the complexity of traceability information in MDE in two ways: firstly, we demonstrate how to identify the different kinds of trace-links that may appear in an end-to-end MDE process; secondly, we describe a rigorous approach to defining semantically rich trace-links between models, where the models themselves may be constructed using diverse modelling languages. The definition of rich trace-links allows us to use tools to maintain and analyse traceability relationships.

Key words traceability, semantics, classifications, identification

1 Introduction

Traceability is the ability to interrelate uniquely identifiable entities in a way that matters. The IEEE Standard Glossary of Software Engineering Terminology [22] defines traceability as "the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match." Thus, traceability refers to the capability for tracing artefacts along a set of chained operations, where these operations may be performed manually (e.g., crafting a software design for a set of software requirements) or with automated assistance (e.g., generating code from a set of abstract descriptions).

The relationships between artefacts are called *trace-links* [41]; informally, a trace-link is a relationship between one or more source model elements and one or more target model elements, optionally containing contextual information (e.g., a name for the trace-link). We make this more precise in Section 2. A *trace model* is a structured set of trace-links, e.g., between source and target models. Trace-links may be defined between entire artefacts (e.g., a requirements document and a design document) or between parts of artefacts; they can be used for many different purposes [19,49], such as *impact analysis* (i.e., to identify the effect of changing one artefact on related artefacts), *code generation*, *code regeneration* (i.e., to automatically regenerate previously generated code that is related to a design artefact), *visualisation*, *change control*, *flexible process modelling* and *requirements coverage assessment*. The intended use of the trace-links dictates the meaning imposed on the link [2]. A variety of different semantics can be applied to trace-links, ranging from simple existence (i.e., "there is a relationship between artefacts") to rich semantics [12] amenable to formal analysis.

Model-Driven Engineering (MDE) involves the construction and manipulation of diverse models in an engineering process [47]. In principle, models can be used *end-to-end* in an engineering lifecycle, from requirements engineering through deployment and maintenance; the manipulations that are applied to models can also be automated—for example, using model transformations [24,28,39], code generation (also known as model-to-text

transformation) [45], and validation [27,36]. In the context of MDE, the engineering artefacts of general interest are models, conforming to a metamodel, and are constructed using a set of modelling tools. Traceability in MDE is therefore predominantly concerned with *identifying* and *managing* the trace-links that relate models or elements of models, so as to support the forms of analysis described previously. However, we frequently start to develop models from other kinds of artefacts: informal, natural language descriptions of requirements, spreadsheets, etc. Traceability in MDE needs to consider these artefacts as well, in terms of how models can be traced to other (non-model) artefacts and how (non-model) artefacts can be traced to models.

Traceability in MDE introduces, or concretises, several challenges. A first, important issue is *what constitutes a trace-link*. While this is an issue for all applications of traceability, it is magnified in the context of MDE due to the different ways in which trace-links can be created (i.e., manually and automatically). A second challenge is coping with the substantial increase in traceability information (i.e., trace-links) generated and maintained in an MDE process; in part, this is due to the application of automated model manipulation tools, like model transformations and code generators, that can automatically generate trace-links, at different levels of granularity. Managing this information, and providing appropriate mechanisms to allow engineers to judiciously access it, is a key difficulty. The third challenge is one that is general to traceability solutions: the lack of agreement regarding the *meaning* of trace-links, and hence the type and quantity of information to be recorded with trace-links [12,20]. Part of the lack of agreement is due to there being no consensus on how trace-links can and should be used: there are many different applications, and trace-links and traceability solutions are often assumed to be helpful for all of them.

Overall, there are a number of dimensions and challenges to the traceability problem [49]: *identification* of trace-links; *encoding trace-links*; *evolving* and *maintaining* trace-links; and *applying* trace-links.

The first challenge, trace-link identification, has two aspects: the concrete problem of identifying a set of trace-links given a specific set of artefacts (which we term *trace analysis*); and the meta-problem of how to identify trace-links for arbitrary sets of artefacts. The second challenge focuses on finding efficient, reusable mechanisms for encoding and storing trace-links, particularly for large-scale applications. The third challenge emphasises managing trace-links once they have been recorded, i.e., maintaining links as artefacts evolve. The final challenge involves application of trace-links, e.g., as detailed in [19,49].

This paper makes contributions towards the challenges of trace-link identification, trace-link encoding, and trace-link maintenance and evolution. It does so in two ways.

Firstly, we focus on the problem of *identification* of trace-links, not for a specific set of models, but for arbitrary sets of MDE artefacts. To do so, we describe a *process* for constructing traceability classifications. Classifications[1] are useful for helping to understand and manage complex collections of information, such as the diversity of trace-link semantics applicable to MDE [15,42,49,53]. Classifications of traceability will help us to understand the requirements for repositories and databases of traceability information, and to understand the different ways in which trace information can be processed. Moreover, classifications can help support the process of *acquiring* traces for specific models, which is difficult and has limited tool support [44].

In addition, we define a tool-supported rigorous approach for implementing the results of the above process. We use the example classifications produced using our process to illustrate and validate the approach. The approach is based on a *domain-specific metamodelling language* [56] for traceability, wherein different kinds of trace models can be instantiated and applied for different contexts. The approach is itself implemented in the Epsilon model management framework, which allows well-formedness rules for trace models to be automatically generated. The metamodelling approach thus allows us to rigorously define the semantics of trace-links, as instantiated in trace models, and thereafter subject traceability information to different forms of automated analysis.

The paper is structured as follows. We start with a review of relevant literature on traceability, focusing on MDE, including previous work on traceability classifications and metamodels, as well as tool support for creating, maintaining and managing trace-links. We also precisely define key terminology. Next, we present a lightweight process for constructing traceability classifications, and give an example of its application.

Then, our rigorous approach to encoding trace-link semantics is presented, and we describe a prototype implementation in the Epsilon framework. The approach is illustrated by an example. Next, we connect the two main contributions of the paper, and show an end-to-end example where a traceability classification is produced (by applying the lightweight process), and parts of the classification are encoded using our prototype implementation. Finally, we conclude with a discussion of limitations and insights on directions for future work and extension.

2 Background and related work

2.1 Traceability definitions

This section introduces key terminology in traceability. The next section summarises related work in traceability,

[1] Traceability classifications are sometimes called traceability metamodels or traceability schemas [15].

while Section 2.3 focuses specifically on related work on traceability in MDE.

Traceability was defined in Section 1. We repeat it here for convenience; this definition was originally presented in the context of software development processes, but in principle can be used in other (e.g., system development) processes.

Definition 1 Traceability: *the degree to which a relationship can be established between two or more products of the software development process [22].*

The relationships defined between products in the development process are called *trace-links*.

Definition 2 Trace-link: *a relationship between source and target products in a software development process, where the relationship encodes contextual information (e.g., the name of the trace-link, provenance, time/date information, and other semantic content) [41].*

Within a software development process, various trace-links may be defined, relating many different products. End-to-end traceability refers to a set of products, spanning the entire software development process, that are connected via trace-links [6].

Definition 3 End-to-end traceability: *a set of software development products, from all phases of the development process (e.g., requirements, design, implementation, deployment) that are related by trace-links.*

Much effort on traceability in software development has focused on so-called *requirements traceability*; traceability to requirements is an important use case for traceability management.

Definition 4 Requirements traceability *[19]: the ability to describe and follow the life of a requirement, both forwards and backwards (i.e., from its origins through its development and specification, to its subsequent deployment and use, and through all periods of ongoing refinement and iteration in any of these phases).*

Pre-requirements traceability is concerned with aspects of a requirement's existence prior to its inclusion in a requirements specification; *post*-requirements traceability is concerned with aspects of a requirement's existence that result from its inclusion in the requirements specification [19].

2.2 Software traceability

This section provides a concise overview of the key literature in software traceability; it does not attempt to provide a comprehensive summary of the state-of-the-art. A thorough roadmap was presented in [49], which in turn provided a clear structure for categorising developments in software traceability. Substantial research has been carried out in each category of software traceability:

- *Identification* of trace-links has been considered by many authors, including Dick [12] in his work on rich traceability, which has been carried through to Telelogic (now IBM) products such as DOORS. Cysneiros et al [11] explored trace-link identification for goal-oriented requirements models. Gotel et al's contribution structures [19] explored trace-links between requirements models and specific types of stakeholders. In terms of traceability classifications or schemas, particularly well known is Ramesh and Jarke's seminal classification for requirements [44]. Walderhaug [53] presents a generic schema, particularly for MDE.
- *Generation* of trace-links has been considered by Alexander, in the context of industrial projects [4]. Antoniol et al [5] also explore the recovery of trace-links between code and documentation, and Egyed considers automated requirements traceability [14] via a recording and replaying mechanism. Automated generation of trace-links is important in MDE (see Section 2.3), as trace-links are often produced as a side-effect of operations that manipulate models.
- *Support for representing and maintaining* trace-links has been explored in detail by Cleland-Huang [10], using notification-based approaches that identify change events. Commerical tools (e.g., IBM Rational's DOORS[2] and CORE[3]) also aim to provide support for this. MDE approaches to this problem provide metamodels, often with extension mechanisms, for the representation of trace-links (see the next section) and their analysis.
- Finally, *empirical investigations* of establishing and deploying trace-links have been extensively studied; see, for example [44, 50].

2.3 Traceability in MDE

In this section, we briefly outline related work on traceability specifically in MDE. We attempt to present this work following the basic structure used in the previous section, i.e., in terms of approaches to identify, generate, represent, maintain, and deploy trace-links and traceability solutions. Before doing so, we observe that one of the core ideas in MDE is to generate more detailed artefacts from less detailed models. Such generation implicitly enables the automatic generation of trace-links as a side effect. MDE-based approaches are potentially well-suited to support automated end-to-end traceability.

Research on *generic* approaches to identification of trace-links in MDE is limited; [42] presents a limited classification of trace-links. [1] describes different kinds of trace-links in MDE. traceMaintainer [31] identifies trace-links between different versions of UML models, based on an understanding of the *types* of changes that

[2] http://www.telelogic.com/Products/doors/doors/index.cfm
[3] http://www.vitechcorp.com/products/Index.html

can be made to UML models. We discuss generic approaches to identification in more detail in Section 3.

There has been substantial investigation into generation of trace-links. A number of model transformation tools (which are fundamental to MDE) as well as other model management operations, support the automated generation of trace-links. The ATLAS Transformation Language (ATL) [24] uses higher-order transformations (HOT) [51] to add trace information in the transformation model. The ATLAS Model Weaver has been used to specify trace-links and automatically generate trace-links that conform to a specific weaving metamodel [16]. The Epsilon framework [25] also provides traceability support through an external trace model that can be accessed explicitly in Epsilon's workflow mechanism (which is based on ANT). An Epsilon program, such as a transformation or a model merging operation, can expose trace information (in the form of a trace model) and this information can be accessed by other model management tasks (such as validations) or even non-MDE tasks, such as visualisations generated with GraphViz. The KerMeta framework also provides facilities for capturing trace-links from transformations, through its Traceability Model Development Kit [52]. KerMeta also provides a generic traceability metamodel and helpers for optimising the trace model (which may be a particular issue when managing large collections of trace-links).

Triple graph grammar (TGG) based approaches to model transformation (particularly update-in-place transformations) provide explicit means for capturing trace-links through so-called *correspondence models* [18]. Correspondence models capture trace-links between source and target model elements, and can be further refined and decorated with additional information to help improve performance of transformations. Correspondence models are used in the process of generating a TGG specification of a model transformation. By contrast, trace-links are often a derived side-effect of running a model transformation, e.g., in ATL or Epsilon.

Existing model-to-text languages have support for generating trace-links. In the MOF Models to Text Standard [38], trace-links are defined to be explicitly created by the use of a trace block inserted into the code, allowing model elements to be traced directly to blocks of text. This approach provides user-defined and user-customisable trace-link definitions; this is specifically useful for adding traces to parts of the code that are not easy to generate automatically. A drawback of the approach is a cluttering of the transformation code with explicit information related to traceability. A complementary approach, as taken in MOFScript [40], is to automate the generation of traces based on model element references. This approach is also taken in the Epsilon Generation Language [45].

In terms of support for representing and storing trace-links, common practice in MDE is to store them in a model based on some form of traceability metamodel. In [23], a simple metamodel tailored to transformation traceability is presented. A more sophisticated approach is to define a generic metamodel, such as the Unified Traceability Scheme discussed in [30] or the traceability framework for software product lines presented in [48]. This metamodel, with proper extensibility mechanisms, should be able to encode any type of traceability links. Another option is to define a core traceability metamodel that encodes a basic set of common features and then define extensions for different types of traceability links. Although this can be implemented with generic weaving tools, the semantics of metamodel extensions are not yet clearly defined. Further discussion on the advantages and disadvantages of a generic metamodel versus a small core that can be extended for each traceability domain can be reduced to a general discussion about generic modeling languages (such as the UML), versus domain-specific languages, a subject clearly outside the scope of this paper.

In terms of storing trace-links that are established by an operation that manipulates a model, Kolovos et al [26] observe that this can be done in two ways: either by embedding trace-links in the models, or by storing them externally in a separate new model. The first approach gives a human-friendly view of the trace-links, but it only supports trace-links between elements in the same model. The external approach has the advantage of having the trace information separated from the model and therefore avoids polluting the models and allows definition of traces between different models.

In the MOF QVT Request for Proposals (RFP) issued by OMG in 2002, traceability is defined as an optional requirement [39]. The specification describes three model transformation languages that can be used: Relations, Core, and Operational Mappings. In the Relations and Operational Mappings languages, trace-links are created automatically without user intervention. In the Core language, a trace class must be specified explicitly for each transformation mapping. The QVT Operational (procedural) implementation that is part of the Eclipse M2M project [17] is an example of this kind of support. The implementation does not store trace models as external files, that would be inter-exchangeable between tools. This can be seen as a disadvantage of the tool [29].

Acceleo Pro Traceability [35] is a traceability tool developed by Obeo that handles traceability links between model elements and code and vice versa. This tool enables round trip support; updates in the model or the code are reflected in the connected artefacts. Analyses are also available using the traces as input, but since this is a commercial tool, restricted information describing the solution is available. It seems to be based on similar ideas that are used in MOFScript where model elements are traced to exact positions in files. traceMaintainer, which was mentioned earlier, provides support for

maintaining trace-links between versions of UML models, based on a set of patterns derived from changes to UML models.

There is as of yet limited reported results on empirical studies of traceability in MDE; the MODELPLEX project[4] and the AMPLE project[5] are investigating using different kinds of trace-links in a number of industrial applications of MDE.

3 Identification of trace-links in MDE

In this section we focus on the problem of *identification* of trace-links in MDE. The key question that we aim to address in this section is: What kinds of trace-links do we want to represent and encode in MDE? This question can be answered in two ways: by identifying the kinds of trace-links of interest in general; that is, in an arbitrary project; and by identifying the kinds of trace-links of interest in the specific, for a given *set of models*. The former is particularly valuable as it helps in the process of deriving requirements for storage of trace-links, maintenance of trace-links, and generation of trace-links, as well as providing some indication of what information needs to be encoded for each kind of trace-link. This, in turn, helps us understand and capture the *semantics* of trace-links.

The number and different kinds of trace-links in MDE is large, in part because of the large number of potential applications for traceability [49]; many different kinds of case- and domain-specific trace-links have been identified [1,42,44]. When confronted with a large and potentially complex set of related concepts that need to be understood, a common approach is to produce a *classification*, where the structure provided by the classification helps to promote understandability and identify commonalities (and points of variation). Classifications, and their study, are widespread in scientific endeavour [7,32], and are widely used in software engineering, e.g., for understanding human error in safety critical systems [33], or for understanding taxonomies in Wikipedia [55].

Classifications of the different kinds of trace-links in MDE have therefore been produced and presented in the literature. For example, classifications given in terms of scenarios of use of traceability are proposed by [41,53]. Classifications in terms of specific domains have been produced by [44] for requirements engineering, for business applications [46], and for usability and accessibility [43]. These classifications emphasise different attributes, characteristics, and viewpoints (ranging from conceptual models to concrete designs that can form the basis of an implementation) on traceability. In particular, two categories of classifications can be identified in the literature: classifications that focus on *explicit trace-links* (which are captured directly in models themselves using a suitable concrete syntax, such as UML dependencies), and such that focus on *implicit trace links*, where trace information is generated or arises due to application of one or more model management operations such as a transformation or model comparison.

Whereas a number of trace-link classifications (including ones for MDE) have been presented in the literature, there is little guidance on how to systematically build them. Since trace-link classifications will be used to support decisions about trace-link maintenance, generation and evolution, and these decisions will impact the technological decisions that are made about supporting MDE processes, these classifications are critical artefacts and guidance for their creation would be helpful. Guidance in the form of a *process* for engineering trace-link classifications would be helpful, not only for building new – perhaps domain-specific – classifications in the first place, but also for *evolving and maintaining* trace-link classifications as requirements change: for example, as new MDE operations become available, as the languages being used evolve, and as new stakeholder requirements for applications of traceability arise.

We have developed guidance in the form of a process for constructing trace-link classifications; the process is derived from one in [9], for classifying kinds of model-based contracts. The process is called the *Traceability Elicitation and Analysis Process (TEAP)*. Its aim is to elicit and analyse traceability relationships within an MDE process, in order to determine how these relationships should fit into a trace-link classification.

We now describe the process in more detail, and then give an example how the process has been used to identify and classify trace-links in the context of a large European project, EU4All[6]. A second example is presented in Section 5, where we also show how to encode example trace-links.

TEAP is an iterative process, consisting of three main activities: Elicitation, Analysis, and Classification (outlined in Table 1). Elicitation involves studying the domain and available scenarios to help identify new trace-links; Analysis involves developing understanding of the new trace-links' semantics and their relationship to other trace-links; and Classification involves structuring the consolidated set of existing trace-links, and new trace-links.

When applying TEAP, we typically bootstrap from an existing trace-link classification—for example, [41,44, 53]. Such classifications typically provide basic infrastructure, such as the notion of a trace-link and modelling elements, upon which to gradually introduce further structures. Existing classifications are iteratively and incrementally refined over a number of TEAP *cycles*. Each cycle enriches the existing classification based on the scenarios used for analysis, and the key attributes of trace-links that are of interest (e.g., date/time stamps).

[4] www.modelplex-ist.org

[5] www.ample-project.net

[6] www.eu4all-project.eu

Activity	Description
Elicitation	Identify new kinds of trace-links through domain and scenario analysis.
Analysis	Understand relationships between new trace-links and existing trace-links; identify constraints.
Classification	Build and/or refine a classification.

Table 1 TEAP activities

Should new scenarios or requirements arise, the classification can be evolved to accommodate these changes by carrying out further TEAP cycles.

TEAP is also a triggered process; the triggers for executing TEAP cycles are:

- a new model management operation has been defined, in which case cycles should be executed in order to refine the classification
- the system development process has changed; thus, cycles should be executed in order to refine how we handle model management workflows (e.g., orchestrations of transformations, code generation steps).
- one or more modelling languages have changed to include new model relationships, or changed model relationships, in which case cycles should be executed in order to refine and extend the explicit link classification.

3.1 Example: applying TEAP to building a classification

We now give a concise example of applying TEAP to building a classification. The example is based on requirements for provision of traceability in a large European project; a second example is given in the sequel.

The example comes from a large-scale project that is, at least partly, applying MDE techniques and tools: the European Approach for Accessible Lifelong Learning (EU4ALL). This project is focused on providing a general service framework to support people with disabilities and older adults in lifelong learning endeavours. This project comprises several international partners including the University of York, the Open University in the UK and Universidad Nacional de Educación a Distancia (UNED) in Spain along with several education and user group representatives.

The emphasis is on understanding the different services that are required by these distinct user groups and what are the requirements for those services. Due to this, the project conducted extensive elicitation activities from both students and educational professionals regarding their preferences and needs for educational systems. This resulted in a massive collection of non-model artifacts, with quantitative survey data, personal interviews and focus groups all yielding results that needed to be incorporated into the final requirements. From the initial data collection, the following steps were undertaken by the requirements team:

1. Each data source (survey, interviews, focus group) was divided into smaller subsections of data that were then categorized by the topic addressed by that subsection. These are referred to hereafter as *topic notes*.
2. Topic notes were analysed both within categories and across categories to produce a comprehensive set of user goals.
3. User goals were grouped together to produce service classifications and goal models.
4. User requirements were written for service classifications.

As can be seen, the various artefacts produced in the project were pre-requirements specification, lending evidence that this process can be used over the entire lifecycle of a development process, providing true end-to-end rich traceability.

The EU4All classification was produced in several iterations, where each iteration was triggered by one of the conditions mentioned earlier. Firstly, an initial structure was constructed for recording basic information. In this case, the basic structure was extremely simple, connecting user goals to original data sources. Implicit trace-links were considered, but the application of automated model management operations was not deemed necessary at that time; thus, the classification did not include such links. Finally, in the last TEAP iteration, information for explicit trace-links was added, including information to be recorded *between* models (i.e., model-model relationships) but particularly between models and non-model artefacts.

Within this classification, we produced the following key kinds of trace-links in the classification:

- *satisfies* links relate sets of requirements to particular user goals represented in the user goal model
- *refine* links relate the user goal model to the topics covered by the requirements elicitation activity outcomes
- *support* links relate the goal model to the actual goals generated from the analysis of the requirements

This system of trace-links provides means of linking the goal model forward to the requirements generated by the project while also linking the model back to the originating material that was generated from the stakeholders.

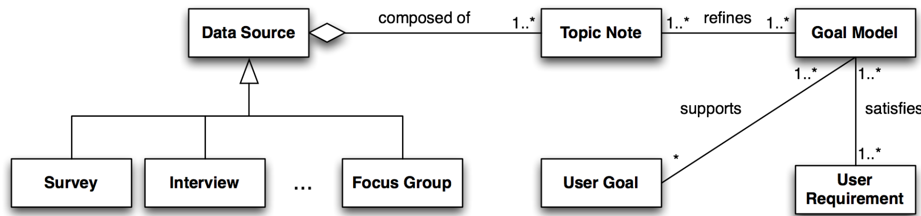

Fig. 1 Snapshot of traceability classification for the EU4ALL project

4 Encoding trace-links

The previous section focused on the problem of how we can *identify* trace-links in the context of an MDE process. In this section we focus on how we propose to encode trace-links, so that the trace-links become amenable to manipulation by tools. Thus, we focus on how to define trace-links with tool-supported semantics.

First, it is important to define precisely what is meant by *semantically rich trace-links.* Semantically rich trace-links possess three characteristics:

1. They are typed.
2. They conform to a case-specific traceability metamodel.
3. The case-specific metamodel should be accompanied by a set of case-specific correctness constraints, which express validity requirements that cannot be captured by the metamodel itself.

The combination of typing, a case-specific traceability metamodel, and case-specific correctness constraints provide a means of encoding the semantics of trace-links in a way that supports manipulation by tools. In the following sections, we will explain briefly why these three characteristics are important, and how trace-links which possess these characteristics can be developed.

4.1 Typed Trace-links

To support automated manipulation of traceability information, trace-links should be typed. In this way, the various semantic heterogeneities among the different trace-links can be captured, hence richer and more precise analysis and reasoning (human or computerized) can be facilitated. Additionally, creation of illegitimate links can be prevented. For example, consider the case where we need to define a trace metamodel which allows the establishment of trace-links between instances of A from metamodel MMa and instances of B from metamodel MMb, but no links between two instances of A or two instances of B. By specifying the typed link A-*to*-B, the user avoids the generation of the invalid link between two instances of A or two instances of B. These link types are specific to creating mappings between metamodels MMa and MMb and have a meaning only in this context.

We and others have argued [23, 26] that trace-link kinds should be defined in a *case-specific trace metamodel*, in order to enable richer, case-specific analysis. We discuss this in more detail in the next section.

4.2 Case-Specific Traceability Metamodel

In the spirit of MDE, the models which contain the traceability information should also conform to a metamodel. The argument for this is uniformity and standardisation: if traceability information is also a model, then standard MDE tooling can be used to process (e.g., transform, validate, verify, analyse, visualise) this information. To accomplish this, there are two approaches: use of a general purpose trace metamodel or use of multiple case-specific trace metamodels. In the case where a general purpose trace metamodel is used, a trace-link can connect any number of elements, of any type and in any model. However, we have argued above that such freedom is not always desirable, because the establishment of illegitimate traceability links cannot be prevented. Additionally, a general purpose trace metamodel cannot capture case-specific traceability information with rigorously defined case-specific semantics. For example, in the case where we want to represent trace-links between a class diagram and a relational database model, and we know that links exist between classes of the former model and tables of the latter, a generic trace metamodel would allow the establishment of illegitimate links such as class–column links.

On the other hand, a case-specific trace metamodel can be defined for each traceability scenario. Such a metamodel can capture case-specific trace-links with well-defined semantics, that potentially include case-specific correctness constraints. Due to the case-specific semantics of the trace metamodels, as well as due to their typed nature, they can prevent users and tools from establishing illegitimate traceability links. Additionally, the trace models which conform to such metamodels can be analyzed in a more rigorous manner.

An example of case-specific trace information is the *link rationale* associated with particular types of trace-links. A description of a trace-link's <<raison d'être>>, as well as the assumptions made behind its creation, can provide additional information about its validity. The

rationale behind a trace-link should capture why a link exists, the assumptions under which the link is valid, and the various alternatives and argumentation behind the choice of one of those alternatives. Additionally, link rationale can support accountability of trace-links. While this concept could be represented in a general-purpose trace metamodel (i.e., as a metaclass), such an approach would not let us easily encode case-specific information that supports rigorous analysis – we would have to encode case-specific information using general concepts such as strings, or other primitive datatypes, thus making analysis more difficult to carry out.

Despite the fact that case-specific trace metamodels require some effort to construct, and also some extra effort to support the direct communication of tools with heterogeneous trace metamodels, we argue that the definition of such metamodels can provide flexibility and specificity for supporting analysis, as well as the potential for automatically generating parts of traceability tools, in a way that is beneficial to a system development project.

4.3 Correctness Constraints

Apart from the aforementioned type safety, there are often additional constraints that need to be specified that the trace metamodel cannot capture by itself. For example, in the context of the previous example, an additional constraint may be that for each instance of *A* from *MMa* there exists one and only one *A-to-B* link, that links it with an instance of *B* from *MMb*. Such a constraint can be specified using a constraint language which supports the expression of cross-model constraints (where the models may conform to different metamodels). The Object Constraint Language [36] currently lacks such capabilities as it does not provide constructs for expressing inter-model constraints[7]. Exemplar constraint languages that support establishing inter-model constraints include the Epsilon Validation Language (EVL) [13] and the XLinkit toolkit [8].

4.4 Example

In this section, we give a concise example of how to define and encode semantically rich trace-links. A further, more realistic example, follows in Section 5.

The aim of this example is to demonstrate how traceability information can be captured between a component model and a class model. Such a scenario can arise in a Component-Based Development Environment, where class diagrams are used to refine the architecture specified by component diagrams into a concrete design. In this example, we use two simple metamodels, the *ClassMetamodel* and the *ComponentMetamodel*, which are illustrated at the top and bottom of Figure 2 respectively. Our aim is to capture traceability links between models which conform to those two metamodels, i.e. the *ComponentModel* and the *ClassModel*. The trace model which will hold the trace information is the *ComponentClassTraceModel* and conforms to the *ComponentClassTraceMetaModel*. More precisely, we want to trace instances of Package from the *ClassMetamodel* and instances of Component from the *ComponentMetamodel*. Additionaly, we want to capture links between instances of Method from the *ClassMetamodel* and instances of Service from the *ComponentMetamodel*. Finally, the following exemplar constraints must be satisfied by the trace model (*ComponentClassTraceModel*):

- **(C1)** For each instance of Service in the *ComponentModel* there is exactly one instance of *ServiceMethodTraceLink* in the *ComponentClassTraceModel* that links it with an instance of Method in the *ClassModel*.
- **(C2)** If an instance of a Service in the *ComponentModel* is linked to an instance of a Method in the *ClassModel* via a *ServiceMethodTraceLink*, then the component in which the service is defined must also be linked with the Package in which the Class that contains the Method is defined.

The first step of the solution is to define the *ClassComponentTraceMetamodel* case-specific traceability metamodel that is displayed as a shaded package in the middle of Figure 2. The metamodel specifies a *TraceModel* container, an abstract *TraceLink* class, and the *ComponentPackageTraceLink* and *ServiceMethodTraceLink* classes that extend *TraceLink*. The *ComponentPackageTraceLink* defines two references: the package reference of type *Package*, and the component reference of type *Component*, from the respective metamodels. Similarly, the *ServiceMethodTraceLink* defines the service and method references which are of type *Service* and *Method* respectively. By specifying explicitly the supported traceability links, we pre-empt establishment of illegitimate links (e.g. tracing a *Component* to a *Method*).

However, the metamodel cannot enforce the additional constraints C1, C2 by itself. To express and enforce such constraints, we use EVL [13], which, as we discussed earlier, is an OCL-like constraint language applicable to inter-model constraints.

In Listing 1 constraint C1 applies to all instances of *Service* in the *ComponentModel* and in line 5 it calculates the number of *ServiceMethodTraceLink* in the *ComponentClassTraceModel* that link the *Service* with a *Method* from the *ClassModel*. In line 6 it returns true if exactly one trace-link is found and false otherwise. Then in line 9 it reuses the *count* variable calculated in the *check* part of the constraint to generate an appropriate

[7] OCL could be used for this purpose by combining the source, target and trace metamodels and applying OCL expressions to models that conform to this unified metamodel.

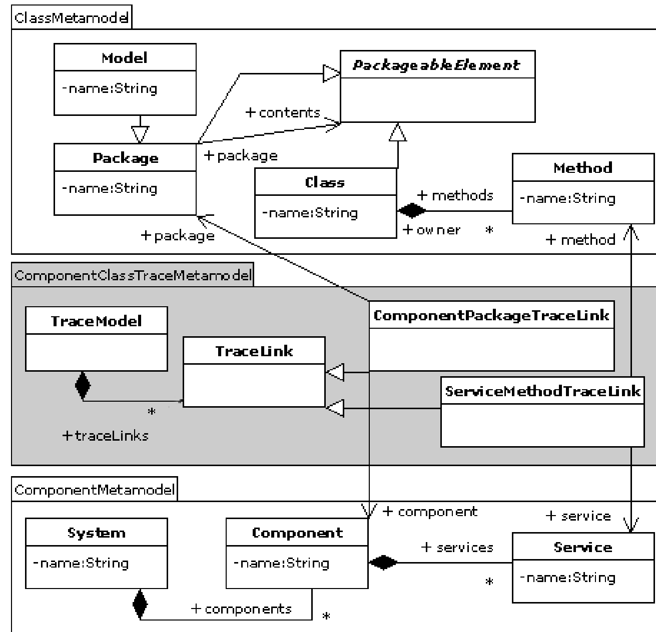

Fig. 2 ClassMetamodel, ComponentMetamodel and ClassComponentTraceLinkMetamodel

error message according to whether zero or more than one trace-links have been found.

Listing 1 Constraint C1 expressed in EVL

```
1 context ComponentModel!Service {
2   constraint C1 {
3     check {
4       var count := ComponentClassTraceModel!
            ServiceMethodTraceLink.
5         all.select(sml|sml.service = self).size();
6       return count = 1;
7     }
8
9     message {
10      if (count = 0) {
11       return 'Service ' + self.name + ' does not
            trace to a method';
12      }
13      else {
14       return 'Service ' + self.name + ' traces to
            many methods';
15     }
16   }
17 }
```

In Listing 2 constraint C2 is evaluated against all instances of *ServiceMethodTraceLink* in the *ComponentClassTraceModel* and checks that there exists at least one *ComponentPackageTraceLink* that links the component in which its *service* is defined with the Package in which the class that contains the *method* is defined.

Listing 2 Constraint C2 expressed in EVL

```
1 context ComponentClassTraceModel!
      ServiceMethodTraceLink {
2   constraint C2 {
3     check : ComponentClassTraceModel!
          ComponentPackageTraceLink.all
4       .exists(cpl|self.service.component = cpl.
            component
5         and self.method.owner.package = cpl.package)
6     message : 'The package in which method '
7       + self.method.name + ' is defined does not
            trace '
8       + 'to the component in which service '
9       + self.service.name + ' is defined'
10  }
11 }
```

In this example we have demonstrated our approach on a simple but representative example. To establish rigorous trace-links between elements of the exemplar *ComponentMetamodel* and *ClassMetamodel* we have introduced a new metamodel (*ComponentClassTraceMetamodel*) in which every legitimate type of traceability link is represented as a separate metaclass that contains references to the types of elements it can link. Moreover, we have demonstrated that additional constraints which cannot be captured by the traceability metamodel are necessary for rigorously specifying legitimate traceability links, and shown how such constraints can be captured using a constraint language (EVL) that supports accessing multiple models simultaneously.

4.5 Summary

The combination of a typed case-specific trace metamodel with inter-model constraints that can be checked automatically prevents users and tools from establishing trace-links without meaning. As well, such an approach enables trace-links that can be automatically validated

and analyzed to discover potential omissions and inconsistencies. Such inconsistencies can arise either during the establishment of the trace-links or later on in the lifecycle of the models among which traceability links have been established.

Through experimentation with defining a number of case-specific trace metamodels we have identified a set of recurring patterns both in terms of the structure of the metamodels and in terms of the additional constraints that guarantee the semantic integrity and completeness of the trace models. A description of these recurring patterns can be found in [34]. The existence of recurring patterns hints that a higher-level of abstraction is potentially beneficial for defining trace metamodels. By this rationale, we have developed the Traceability Metamodelling Language (TML) [34], a domain specific language which promotes the identified patterns into first-class artefacts and which is dedicated to the construction of trace metamodels with the aforementioned characteristics. Currently, TML supports only model-to-model traceability, but we are working on extending this language to support traceability among all kinds of artefacts created throughout a MDE process.

In the next section, we will present a more realistic example, which demonstrates both identification, and encoding of trace-links.

5 Example: identifying and encoding trace-links

We now present an example where we show how to identify and encode semantically rich trace-links. The example thus illustrates the use of both key contributions of the paper: the traceability elicitation and analysis process (Section 3), and the encoding scheme presented in Section 4.

Our example comes from requirements for traceability from the European project MODELPLEX, which is using MDE for building, maintaining and analysing complex systems. MODELPLEX is a 40-month integrated project, with a mandate to improve productivity in the development of complex systems through the use of MDE. The project is driven by case studies from four industrial partners – SAP, Telefonica, Western Geco, and Thales Information Systems – providing real complex system scenarios, to which MDE technology (e.g., architectural modelling, model transformation, performance analysis, simulation, model composition) is to be applied. Each case study has concrete traceability requirements, which were developed iteratively and incrementally during the first phase of the project. These can be summarised as follows.

- The ability to record trace-links that result from applying model management operations: model-to-model (M2M) transformations, model-to-text (M2T) transformations, compositions, simulations, and refinements.
- The ability to manually create trace-links between MDE artefacts, e.g., between an architectural model and a use case model, between a weaving model and a design model. Manual creation of trace-links can involve modelling tools, or the use of a textual domain-specific language tailored for one or more of the case studies.
- the ability to (typically manually) create trace-links between MDE artefacts (e.g., models) and non-MDE artefacts (e.g., requirements stored in a MANTIS repository, PDF documents). This is a necessary requirement as some of the MODELPLEX partners did not use MDE technologies in their everyday practice at the start of the project; moreover, non-MDE artefacts will always play a substantial role in the MDE process – for example, for early requirements elicitation and description.
- the ability to store and retrieve trace-links from a repository.

We used TEAP to produce a classification for the kinds of trace-links for MODELPLEX. The classification was produced in several iterations, where each iteration was triggered by one of the conditions mentioned in Section 3. Firstly, an initial structure was constructed for recording basic information: artefacts and links between them. This structure was represented as a trivial metamodel, based on [53]. Next, information for implicit trace-links was added, focusing on trace information to be recorded as a side-effect of carrying out a model management operation (particularly M2M and M2T transformation, which were the most widely used operations throughout the MODELPLEX project). In the third iteration, information for explicit trace-links was added, focusing on trace information to be recorded *between* models (i.e., model-model relationships) and between models and artefacts (such as informal requirements). This refinement populated the classification with concrete kinds of trace-link such as:

- *consistent-with* links, where two model elements must remain consistent with each other, e.g., an activity and a class.
- *dependency* links, where the structure and meaning of one set of model elements depends on a second set.
- *satisfies* links, to indicate that properties or requirements captured in an artefact are satisfied by a model. Variants on *satisfies* links include *verifies* links (which involve a specific mechanism, such as testing) and *certifies* links (which also link to external standards and arguments for safety or security).
- *allocated-to* links, used when information in a non-model artefact is allocated to a specific model element that represents the information.
- *refines* links, where the behaviour of one model element is refined (and is specified in more detail) by a second model element.

In practice, these different kinds of trace-links were identified and added to the classification over the course of several short iterations.

A snapshot of the MODELPLEX traceability classification (focusing specifically on the explicit trace-links) is shown in Fig. 3. This structure is part of a larger traceability classification that also includes the implicit trace-links and basic infrastructure mentioned above. A fuller presentation can be found in [42].

Within the MODELPLEX project, there were several pre-determined milestones, and as each milestone approached it became necessary to produce and deliver concrete, prototypical tool support for specific scenarios of traceability, in order to support *modelling* of traceability information, and to support *completeness analysis* for requirements traceability. The traceability classification was presented to the MODELPLEX partners, and specific trace-links of value were identified as candidates for implementation to support requirements traceability. We now describe an example of selecting and implementing MODELPLEX trace-links, using the traceability scheme presented in Section 4.

5.1 Setting up the Context

The example we present focuses on an early milestone in the project, where requirements traceability needed support. Providing this support entailed identifying the specific artefacts that required traceability support, identifying the kinds of trace-links needed, and implementing these trace-links using the scheme presented in Section 4. We summarise the first step in this section, and describe the final two steps in Section 5.2.

In the requirements engineering (RE) literature, the RE effort is divided in two broad phases, the *early-phase* and the *late-phase* [54]. The *early-phase* activities include those that consider how the intended system can be integrated in the organisation, how it can meet the organisational goals, why the system is needed, what alternatives exist, etc. The emphasis of this phase is on understanding the various *"whys"* that underlie the system requirements rather than on *"what"* the system should do. The knowledge obtained from the *early-phase* activities will be used to guide the activities in the later phases of the software development.

In contrast to the *early-phase* of RE, the *late-phase* focuses on coming up with a detailed requirements specification as well as on the completeness, consistency, and automated verification of this specification. Following [54], it is argued that because *early-phase* RE activities have different objectives and goals from those of the the *late-phase* ones, it is appropriate to have different modelling support for the two phases.

According to [3], the i* framework [54,21] is well suited for *early-phase* requirements capture, while the Unified Modelling Language (UML) [37] can be used for *late-phase* requirements capture. UML is not adequate to deal with early requirements capture and analysis, since it cannot describe and evaluate alternatives and their relationship to organisational objectives. Instead, the i* framework can be used , since it allows for a better description and reasoning of the various organisational relationships among the agents of a system as well as the understanding of the rationale of the decisions taken. Hence, both UML and the i* framework can be used complementarily to support the entire RE phase. However, since different notations will be used to represent different but possibly overlapping views of the system, traceability information must be captured and maintained among the different views.

i* [21,54] is a modelling framework which focuses on the strategic actor relationships. In contrast to modelling languages like UML or Entity-Relationship diagrams, i* provides abstractions which are capable of expressing organisational relationships among the various organisational agents and the rationale behind the various decisions taken. The various participants of the organisational setting are actors with intentional properties such as goals and beliefs. These actors depend on each other in order to fulfill their objectives and have their tasks performed. The i* framework consists of two models: the Strategic Dependency (SD) model and the Strategic Rationale (SR) model.

The SD is a graph whose nodes represent the system actors and the vertices represent dependency relationships among the various system actors. The goal of an SD model is to capture the motivation and the rationale of actor's activities. An SD model distinguishes four types of dependencies; goal dependencies, task dependencies and resource dependencies. Furthermore a goal can be either a hard goal or a soft goal. In the RE literature, a soft goal is a goal, which cannot be precisely defined. A soft goal is similar to a hard goal except that the criteria for whether a soft goal is achieved are not clear-cut and a priori. In a goal dependency, an actor depends on another actor for the fulfillment of one goal. In a resource dependency, an actor depends on another actor for a resource, while in a task dependency an actor depends on another actor to carry out a task. Actors can be classified into agents, roles and positions. An agent is an actor with a concrete physical existence. A role is an abstract characterisation of the behaviour of a social actor within some specialised context, while a position is a set of roles of a single agent. A simplified metamodel for an SD model can be found in figure 4.

Late requirements can be expressed in languages such as UML, using, e.g., class diagrams. In the context of RE, a class diagram can be thought of as a conceptual model of the system under consideration. It describes the various entities involved in the system, their relationships, the domain model constraints, the system scope as well as the domain-vocabulary. Hence, it provides a structural view of the system under consideration and it can

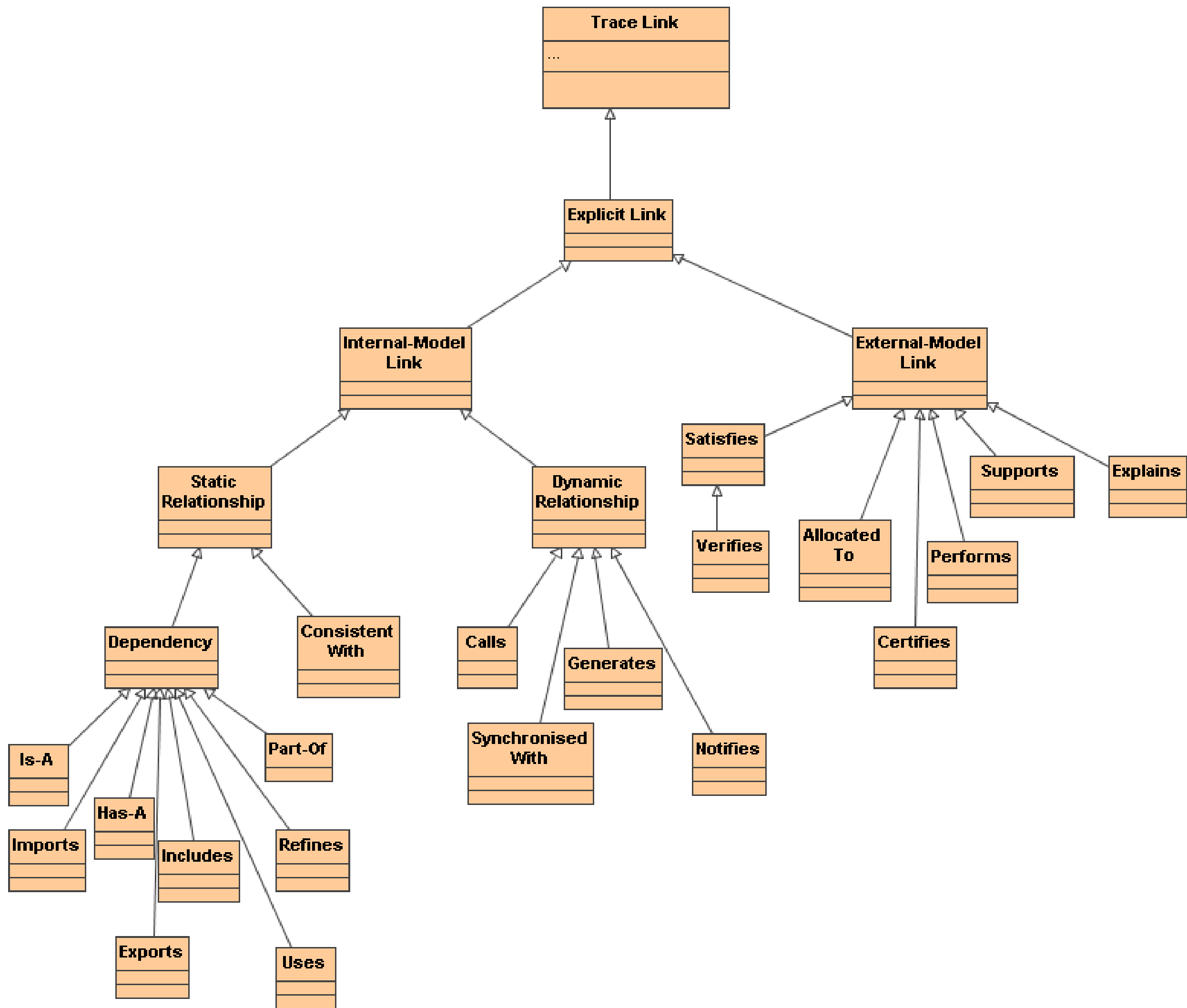

Fig. 3 Snapshot of part of a traceability classification of explicit MDE links

be complemented by other dynamic views, such as Use Case Models [37]. An example of a class diagram metamodel is in Figure 5.

The metamodels in Figures 4 and 5 capture the artefacts that were to be traced. In the next section we describe the kinds of trace-links that were required, and how they relate to the MODELPLEX classification of Figure 3. Then we explain how to implement these trace-links.

5.2 Applying the Approach

The following kinds of trace-links, relating the requirements models presented in Figures 4 and 5, were identified. Two specific trace-link types in Figure 3 were found to be useful.

- *consistent-with* trace-link: instances of the *Actor* class in the i* framework will be mapped to instances of the *Class* class in the OO model.
- *consistent-with* trace-link: a *Task* in a SD model will be mapped to an *Operation* of the relevant class in the OO model.
- *consistent-with* trace-link: instances of the *Resource* class in the i* framework will be mapped to instances of the *Class* class in the OO model.
- *refines* trace-links: instances of the *HardGoal* class and instances of the *SoftGoal* class in the i* framework will be mapped to instances of the Attribute class, which belong to relevant classes in the context class diagram. Since this context class diagram is part of the late-phase of RE, the recording of the various domain entities' goals and their satisfaction in the form of class attributes is considered to be beneficial.

In practice, we found that *consistent-with* trace-links were used frequently for capturing relationships between early artefacts, e.g., conceptual models and early requirements. Relationships between later artefacts (e.g., architectural models, implementations) generally involved dy-

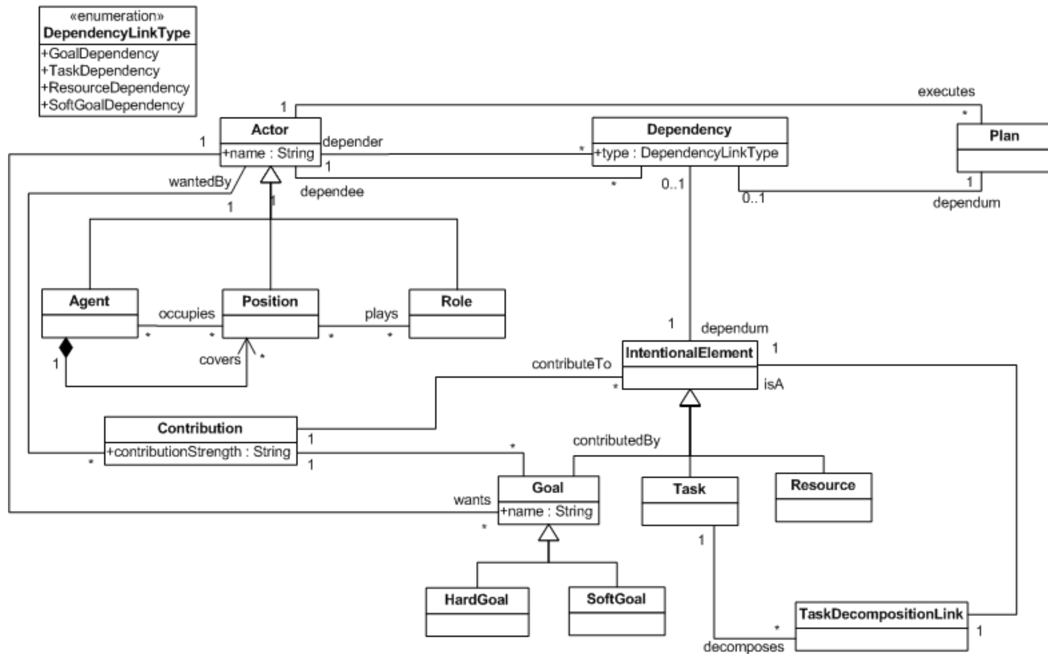

Fig. 4 The SD Metamodel of the i* Framework

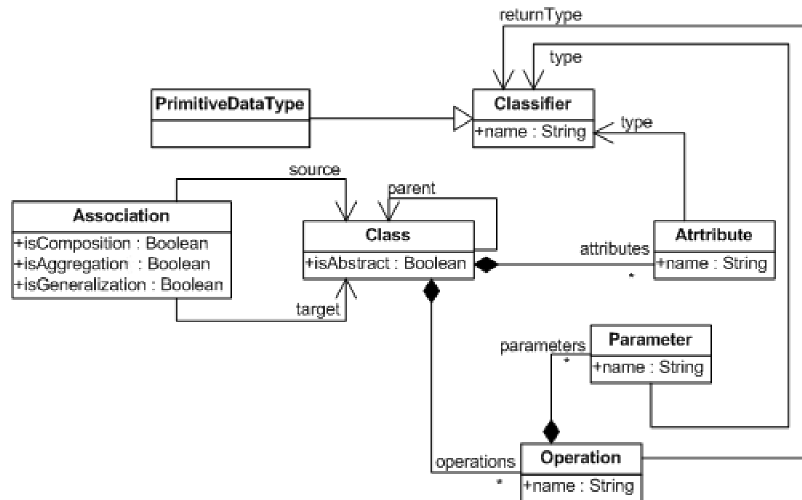

Fig. 5 Class Diagram Metamodel

namic trace-links and *uses* trace-links, amongst others. Further experiments and applications will be required to fully clarify the scope of applicability of different kinds of trace-links.

We now implement these trace-links, using the scheme of Section 4. The first step is to define the *IStarOOTraceMetamodel*. This metamodel specifies the *TraceModel* class, which acts as a container for the various trace-links. Additionally, the *TraceLink* abstract class is specified, which is extended by the various case specific trace-link classes. Namely, it is extended by the *ActorClassTraceLink* class, the *TaskOperationTraceLink* class, the *ResourceClassTraceLink* class, the *HardGoalAttributeTraceLink* class and finally the *SoftGoalAttributeTraceLink* class. Each one of these classes specifies two references, one in the SD metamodel and one in the sample OO metamodel. These trace-links as well as the associated references are illustrated in figure 6. By explicitly specifying these typed links, we manage to capture information which can be used to prevent the establishment of illegitimate links or for performing analysis over the traceability information throughout the development lifecycle.

Additionally, the following constraints must be satisfied by the trace models.

- **(C3)**: For every *HardGoalAttributeTraceLink*, the Attribute end of the link must be of type *Boolean*. This is because hard goals are well defined, hence it is always possible to establish if one has been fulfilled or not.

– **(C4)**: For every *SoftGoalAttributeTraceLink*, the Attribute end of the link must be an *Enumeration*. This is due to the fact that soft goals are not well defined. They can only be *satisfied* to some degree and the different values of the *Enumeration* represent the different degrees of soft goal fulfillment.
– **(C5)**: For every *ActorClassTraceLink*, the name of the Actor reference and the name of the Class reference should be the same. This constraint will not produce an error when it is violated but a *critique*. That is, this constraint checks for a non-critical issues that should nevertheless be addressed by the user.
– **(C6)**: Similarly to **C5**, for every *ResourceClassTraceLink*, the name of the Resource reference and the name of the Class reference must be identical.

Similarly to the previous example, we use EVL to impose the aforementioned constraints. In Listing 3, constraint **C3** applies to all instances of the *HardGoalAttributeTraceLink* in an *IStarOOTraceModel*. In line 3, a *guard* is defined, which checks if an attribute end is defined for the instance of the *HardGoalAttributeTraceLink*. In EVL, a *guard* limits the applicability of a constraint to a narrower subset of instances of its specified type. Hence, if the guard fails, namely no attribute end is defined for the particular link instance, then the contained invariant will not be evaluated. Line 4 returns true if the attribute reference of the *HardGoalAttributeTraceLink* link is of type *Boolean* and false otherwise. If line 4 returns false, that is if the attribute reference of the *HardGoalAttributeTraceLink* link is not of the type *Boolean*, then an appropriate error message is produced in line 5.

Listing 3 Constraint C3 expressed in EVL

```
1     context IStarOOTraceModel!
          HardGoalAttributeTraceLink {
2       constraint C3 {
3         guard : self.attribute.isDefined()
4         check : self.attribute.type = OOMetamodel!
              PrimitiveType#"'Boolean'"
5         message : 'Attribute ' + self.attribute.
              name + ' should be of type Boolean'
6       }
7     }
```

Constraint **C4** works in a similar manner. The EVL code for constraint **C4** is illustrated in Listing 4. Constraint **C4** applies to all instances of the *SoftGoalAttributeTraceLink* in an *IStarOOTraceModel* . Line 3 defines a *guard*, which checks if an attribute end is defined for the instance of the *SoftGoalAttributeTraceLink*. If line 5 returns false, that is if the attribute end of a *SoftGoalAttributeTraceLink* is not an Enumeration, then an error message is produced in line 7.

Listing 4 Constraint C4 expressed in EVL

```
1     context IStarOOTraceModel!
          SoftGoalAttributeTraceLink {
2       constraint C4 {
3         guard : self.attribute.isDefined()
4
5         check : self.attribute.isTypeOf(
              OOMetamodel!Enumeration)
6
7         message : 'Attribute ' + self.attribute.
              name + ' should be of type Enumeration
              '
8       }
9     }
```

In Listing 5, a critique is evaluated against all instances of the *HardGoalAttributeTraceLink* in a *IStarOOTraceModel*. It checks if the attribute reference of an instance of the *HardGoalAttributeTraceLink* has the same name as the hard goal reference of the same link. If they do not have the same name, a warning message is produced by line 7. In Listing 6, a similar critique is evaluated against all instances of the SoftGoalAttributeTraceLink in an *IStarOOTraceModel*.

Listing 5 Critique C5 expressed in EVL

```
1     context IStarOOTraceModel!ActorClassTraceLink
          {
2       critique C5 {
3         guard : self.attribute.isDefined() and
              self.hardGoal.isDefined()
4
5         check : self.attribute.name.toLowerCase
              () = self.hardGoal.name.toLowerCase
              ()
6
7         message : ' The name of the attribute '
              + self.attribute.name + ' and the
              name of the hard goal ' + self.
              hardGoal.name + ' do not match'
8 }
9 }
```

Listing 6 Critique C6 expressed in EVL

```
1     context IStarOOTraceModel!
          ResourceClassTraceLink {
2       critique C6 {
3         guard : self.attribute.isDefined() and
              self.softGoal.isDefined()
4
5         check : self.attribute.name.toLowerCase
              () = self.softGoal.name.toLowerCase
              ()
6
7         message : ' The name of the attribute '
              + self.attribute.name + ' and the
              name of the soft goal ' + self.
              softGoal.name + ' do not match'
8       }
9     }
```

As with the previous example, to establish traceability links between the SD metamodel of the i* framework and a sample OO metamodel, we have introduced a new

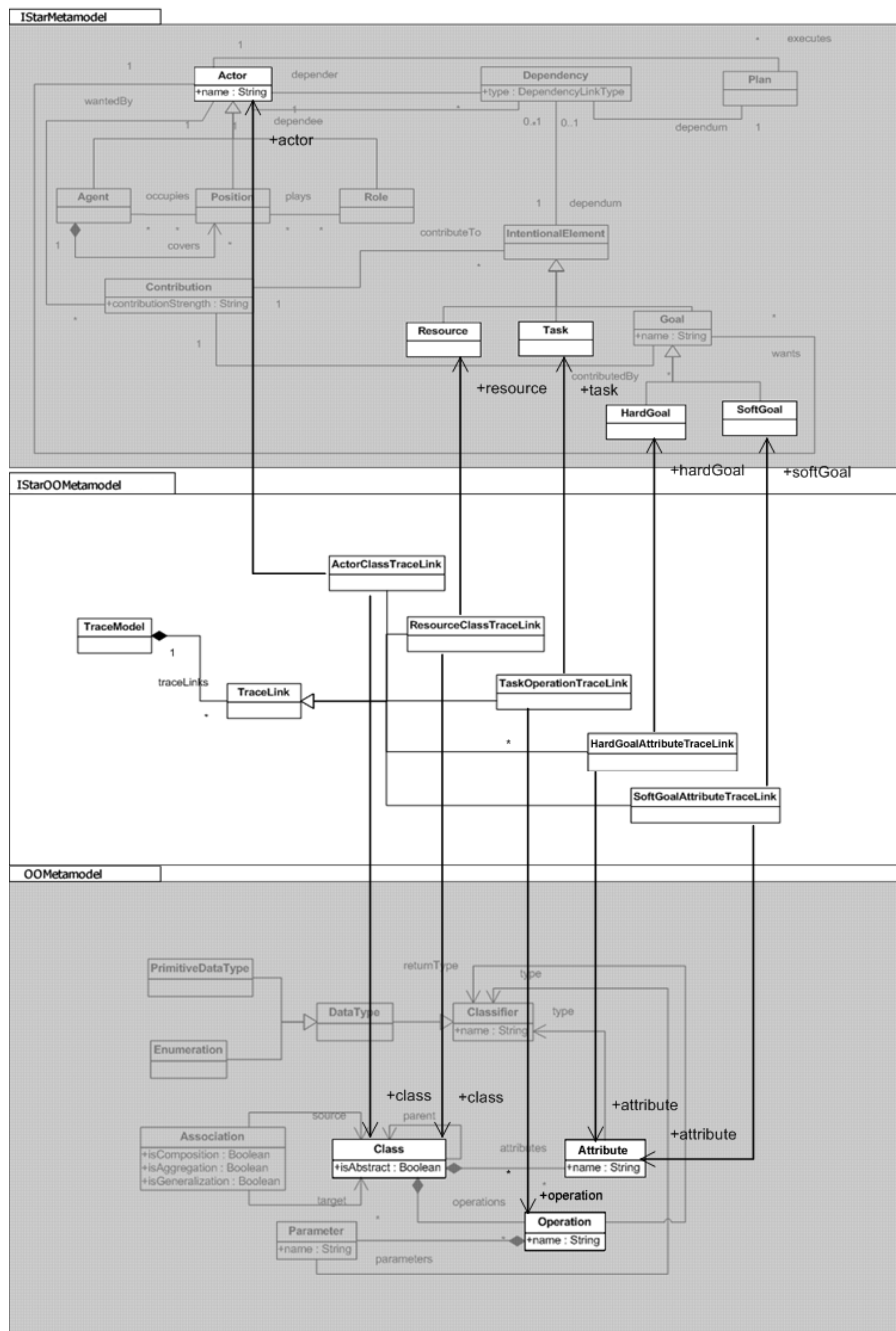

Fig. 6 *IStarOOTraceMetamodel*

case specific metamodel, namely the *IStarOOTraceMetamodel*. In this metamodel, every legitimate type of traceability link is represented as a metaclass, which contains references to the types of elements it can link. Additionally, constraints which cannot be captured by the metamodel are represented using EVL.

6 Conclusions and future work

We have provided insights into an approach for identifying, defining and implementing semantically rich trace-links in MDE, and have illustrated the approach with examples. Our contributions are two-fold. Firstly, we showed a means for identifying trace-links in an MDE process through a traceability elicitation process. The process was validated by illustrating how it has been used in two-large scale projects, to determine and clas-

sify trace-link requirements. Secondly, we illustrated a way to define and implement such trace-links. Our approach is based on providing typed trace-links that conform to case-specific metamodels. The semantics is enriched by provision of case-specific validity constraints. We implemented the overall approach using the Epsilon model management platform.

We are continuing our work in two main directions. We are exploring larger, richer case studies, particularly in the requirements domain – where we are analysing and implementing trace-link support for different goal-oriented modelling languages. These case studies will also give us the opportunity to validate trace metamodels with respect to user requirements. As well, we are working on providing richer tool support for trace-link definition and implementation, partly through our Traceability Metamodelling Language [34]. Further, we are providing tool support for managing very large collections of trace-links, which may arise in managing large heterogeneous models.

Acknowledgments

The work in this paper was supported by the European Commission via the MODELPLEX project and the AMPLE project, co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2006-2009).

References

1. N. Aizenbud-Reshef, B. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45(3), 2006.
2. N. Aizenbud-Reshef, R. Paige, J. Rubin, Y. Shaham-Gafni, and D. Kolovos. Operational semantics for traceability. In *Proc. ECMDA Workshop on Traceability*, 2005.
3. F. Alencar, J. Castro, G. Cysneiros, and J. Mylopoulos. From Early Requirements Modeled by i* Technique to Later Requirements Modeled in Precise UML. In *In Proc. of the III Workshop de Engenharia de Requisitos, Rio de Janeiro, Brasil*, 2000.
4. I. Alexander. Towards automatic traceability in industrial practice. In *Proc. Workshop on Traceability in Emerging Forms of Software Engineering*, 2002.
5. G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10), 2002.
6. H. Asuncion, F. François, and R. Taylor. An end-to-end industrial software traceability tool. In *ESEC-FSE*, pages 115–124, New York, NY, USA, 2007. ACM.
7. S. Atran. *Cognitive Foundations of Natural History.* Cambridge University Press, 1993.
8. C. Nentwich and L. Capra and W. Emmerich and A. Finkelstein. XLinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151–185, May 2002.
9. Z.E. Chan and R.F. Paige. Designing a domain-specific contract language. In *ECMDA*, volume 3748 of *LNCS*. Springer, 2005.
10. J. Cleland-Huang, C. Change, and J. Wise. Supporting event-based traceability with high-level recognition of change events. In *COMPSAC*. Springer, 2002.
11. G. Cysneiros, A. Zisman, and G. Spanoudakis. A traceability approach for i* and uml models. In *Proc. Workshop on Software Engineering for Large-Scale Multi-Agent Systems*, 2003.
12. J. Dick. Rich traceability. In *Proc. Workshop on Traceability in Emerging Forms of Software Engineering*, 2002.
13. D.S. Kolovos and R.F. Paige and F. Polack. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In *Proc. Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis*, 2007.
14. A. Egyed and P. Gruenbacher. Automatic requirements traceability. In *ASE*. ACM, 2003.
15. A. Espinoza, P. Alarcón, and J. Garbajosa. Analyzing and systematizing current traceability schemas. In *SEW*. IEEE, 2006.
16. M. Didonet Del Fabro and P. Valduriez. Towards the efficient development of model transformations using model weaving and matching transformations. *Software and System Modeling*, 8(3):305–324, 2009.
17. Eclipse Foundation. Eclipse model-to-model transformation, last accessed: June 2009. http://www.eclipse.org/m2m/.
18. H. Giese and R. Wagner. Incremental model synchronization with triple graph grammars. In *MoDELS*, volume 4199 of *LNCS*. Springer, 2006.
19. O. Gotel and A. Finkelstein. An analysis of the requirements traceability problem. In *RE*. ACM, 1994.
20. O. Gotel and A. Finkelstein. Contribution structures (requirements artifacts). In *RE*. ACM, 1995.
21. G. Grau, C. Cares, X. Franch, and F. J. Navarrete. A comparative analysis of i* agent-oriented modelling techniques. In *SEKE*, pages 657–663, 2006.
22. IEEE. IEEE Standard Glossary of Software Engineering Terminology. IEEE Standard 610.12-1990, 2004.
23. F. Jouault. Loosely coupled traceability for ATL. In *Proc. ECMDA Workshop on Traceability*, 2005.
24. F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like transformation language. In *OOPSLA Companion*, pages 719–720, 2006.
25. D.S. Kolovos and R.F. Paige. Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon). http://www.eclipse.org/gmt/epsilon.
26. D.S. Kolovos, R.F. Paige, and F. Polack. On-demand merging of traceability links with models. In *Proc. ECMDA Workshop on Traceability*, 2006.
27. D.S. Kolovos, R.F. Paige, and F. Polack. Detecting and repairing inconsistencies across heterogeneous models. In *ICST*. ACM, 2008.
28. D.S. Kolovos, R.F. Paige, and F. Polack. The epsilon transformation language. In *ICMT*, volume 5063 of *LNCS*. Springer, 2008.
29. I. Kurtev, M. Dee, A. Goknil, and K. van den Berg. Traceability-based change management in operational mappings. In *Proc. ECMDA Workshop on Traceability*, June 2007.

30. A. Limon and J. Garbajosa. The need for a unifying traceability scheme. In *Proc. ECMDA Workshop on Traceability*, 2005.
31. P. Mäder, O. Gotel, and I. Philippow. Enabling automated traceability maintenance through the upkeep of traceability relations. In *ECMDA*, volume 5562 of *LNCS*. Springer, 2009.
32. J. Malone. *The Science of Linguistics in the Art of Translation.* State of New York University Press, 1988.
33. L. Mussulman and D. White. Human factors analysis and classification system, 2004. http://www.safetycenter.navy.mil/.
34. N. Drivalos and D. S. Kolovos and R. Paige and K. Fernandes. Engineering a Domain-Specific Language for Software Traceability. In *Software Language Engineering*, volume 5452 of *LNCS*. Springer, 2008.
35. Acceleo Pro Traceability. http://www.acceleo.org/pages/additionnal_products/en, 2009.
36. Object Management Group. UML 2.0 OCL specification. OMG Document, October 2003. URL http://www.omg.org/cgi-bin/doc?ptc/03-10-14.
37. Object Management Group. UML 2.0 infrastructure specification. OMG Document, October 2004. URL http://www.omg.org/cgi-bin/doc?ptc/04-10-14.
38. Object Management Group. MOF models to text transformation language; final adopted specification. OMG Document 08-01-16, http://www.omg.org/spec/MOFM2T/1.0/PDF, 2006.
39. Object Management Group. MOF QVT draft specification. OMG Document ptc/2007/07/07, http://www.omg.org/cgibin/doc?ptc/2007-07-07, 2007.
40. J. Oldevik, T. Neple, R. Gronmo, J. Aagedal, and A. Berre. Toward standardised model-to-text transformations. In *ECMDA*, number 3748 in LNCS. Springer, 2005.
41. G. Olsen and J. Oldevik. Scenarios of traceability in model-to-text transformations. In *ECMDA*, volume 4530 of *LNCS*. Springer, 2007.
42. R.F. Paige, G.K. Olsen, D.S. Kolovos, S. Zschaler, and C. Power. Building model-driven engineering traceability classifications. In *Proc. ECMDA Workshop on Traceability*, 2008.
43. C. Power, H. Petrie, D. Swallow, and R.F. Paige. Pre-requirements traceability in universally accessible e-learning systems (under review). 2009.
44. B. Ramesh and M. Jarke. Toward reference models of requirements traceability. *IEEE Trans. Software Eng.*, 27(1):58–93, 2001.
45. L.M. Rose, R.F. Paige, D.S. Kolovos, and F. Polack. The Epsilon Generation Language. In *ECMDA*, volume 5095 of *LNCS*. Springer, 2008.
46. A. Rummler, B. Grammel, and C. Pohl. Improving traceability in model-driven development of business applications. In *Proc. ECMDA Workshop on Traceability*, 2007.
47. D.C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.
48. A. Sousa, U. Kulesza, A. Rummler, N. Anquetil, R. Mitschke, A. Moreira, V. Amaral, and J. Araújo. A model-driven traceability framework to software product line development. In *Proc. ECMDA Workshop on Traceability*, 2008.
49. G. Spanoudakis and A. Zisman. Software traceability: a roadmap. In *Handbook of Software Engineering and Knowledge Engineering, Vol. III.* World Scientific Publishing, 2005.
50. A. Strens and R. Sugden. Change analysis: a step towards meeting the challenge of changing requirements. In *Proc. Workshop on Engineering Computer-Based Systems*, 1996.
51. M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the use of higher-order model transformations. In *ECMDA*, volume 5562 of *LNCS*. Springer, 2009.
52. D. Vojtisek. Traceability MDK for KerMeta, last accessed: June 2009. http://www.kermeta.org/mdk/traceability/.
53. S. Walderhaug, U. Johansen, E. Stav, and J. Aagedal. Towards a generic solution for traceability in MDD. In *Proc. ECMDA Workshop on Traceability*, 2006.
54. E. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *RE*. ACM, 1997.
55. C. Zirn, V. Nastase, and M. Strube. Distinguishing between instances and classes in the wikipedia taxonomy. In *ESWC*, 2008.
56. S. Zschaler, D.S. Kolovos, N. Drivalos, R.F. Paige, and A. Rashid. Domain-specific metamodelling languages for software language engineering. In *Software Language Engineering*, LNCS. Springer, 2009.