Constraint-based Model Transformation: Tracing the Preservation of Semantic Properties

Thomas Triebsees
Department of Computer Science, Universität der Bundeswehr München, 85577 Neubiberg, Germany
Email: Thomas.Triebsees@unibw.de

***Abstract*— We present and evaluate a novel constraint-based model transformation approach that implements a preservation-centric view. The proposed framework comprises formal preservation constraints that can be used to specify the preservation of invariants that are possibly implemented differently in the source and target model. These invariants are enclosed in concepts, which at the same time serve as grouping mechanism for their different implementations. In that, our framework abstracts from the concrete implementation languages by pre-supposing only a basic set of modeling constructs. To this end, we present two case studies where we apply our approach for the preservation of non-trivial properties and provide some performance analysis where we show that tracking the preservation of a relevant class of complex properties can be done in linear time.**

***Index Terms*— model transformation, semantic preservation, constraints**

I. Introduction

Model transformation has applications in many areas like model-driven software development (MDD) or automated knowledge exchange [1]–[4]. When applied, model transformations usually are designed to *preserve* certain properties. In MDD, e.g., abstract models are transformed into more specific ones while preserving the behavior of the specified software system [5], [6]. As another prominent example, the rapid emergence of the XML technology and the resulting diversity of XML-related domain-specific document formats requires for "content-preserving" document transformations.

Whenever a high degree of reliability of these transformations is desired, formal model transformation approaches, like diverse variants of graph transformation [7]–[9] or stepwise refinement techniques [10], [11] are used and complemented by formal proofs. When the specifications are executable, a higher degree of automation is achieved as an important by-product by approaches like Programmable Graph Transformation [8].

The taxonomy for model transformation introduced in [5], however, identifies some scenarios that inherently decrease the level of automation. Firstly, it may well turn out that proofs are hard to carry out as system complexity grows. Graph transformation methods, e.g., may require termination or confluence proofs. These proofs can be quite challenging and are known to be undecidable in general [7]. As an alternative, approaches dealing with *model checking* of graph transformations can be found in the literature, but scalability remains an open problem [2].

Secondly, scalability with regard to efficiency is still in doubt for formal transformation approaches [5], [6]. As a remedy, expressive control mechanisms have been developed [7], [8]. They help guide transformations but can, in general, not overcome inherent complexity issues of model transformation problems. As a relevant example, take DB-schema to XML mapping. This application has gained more and more importance since the emergence of the XML standard. There, it is, in general, undecidable whether database-related constraints like foreign key conditions can be preserved in XML documents in the presence of DTD's or XML-schemas [12]. Under more restrictive assumptions on the underlying constraints, the satisfiability problem decreases to be NP-complete or – at best – PSPACE hard. In such situations, human interaction can be necessary, which in turn requires alternative approaches that guarantee the preservation of relevant properties.

In all these scenarios, a verification-oriented technique can complement formal transformation approaches by *checking relevant parts* of their output for correctness on the instance level. This would exploit the fact that property checking can often be done in linear or polynomial time where automated model construction may cause an exponential blow-up. In primarily human driven model transformation scenarios (like in early phases of a software development process), this even seems to be the most promising strategy [5], [13], [14]. In [15] we have introduced a declarative, constraint-based approach that can be used to constrain permissible model transformations from a preservation-centric perspective. In particular, we have shown its automated model construction facilities by non-trivial examples like automated preservation of link-consistency in web model transformations [15], [16].

Motivated by the introductory explanations, we will study our approach from another perspective in this article. We will provide two case studies and examine its constraint-*checking* facilities. In particular, these case studies come from significantly different application domains in order to underline applicability of our method. We use our framework to 1) specify model invariants that have to be preserved by the respective transformation process and 2) evaluate adherence to the specified constraints in an automated way. It turns out that the ability

This work has been partially funded by the Universität der Bundeswehr München.

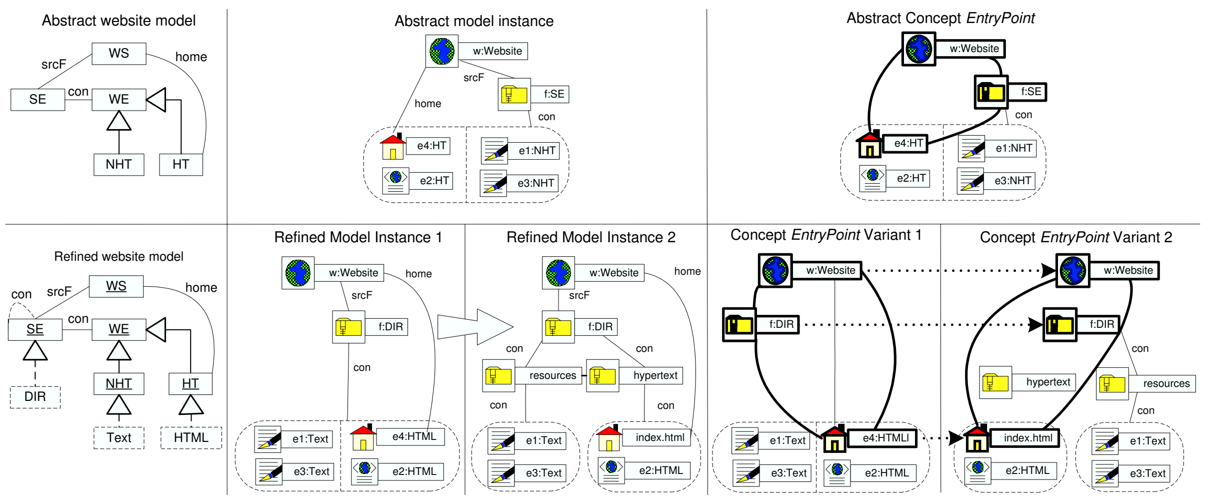


Figure 1. Running example: Transformation of website models

to *trace* model changes w.r.t. relevant model entities is decisive since model invariants can be expressed by different languages in the source and target model. In [15] we introduced a mechanism that requires all implementations of a *concept* (semantic property) to adhere to the concept's interface. We will see that this mechanism – together with our formal notion of preservation – allows us to elegantly relate source model instances to different versions of target model instances and, thus, facilitates efficient constraint-checking.

This article is organized as follows: We present the basic ideas in Sect. II on an informal level. We introduce a running example and use it to motivate the technical parts of this article, which are provided in Sects. III-V. There, we recall some preliminary results published in [15], provide a more profound explanation of our notion of preservation and distinguish our work from other approaches. Moreover, we will introduce a new variant of preservation constraints, the necessity of which will be motivated with the running example. Finally, we employ two case studies ranging from systems specification to the transformation of document models in order to evaluate our approach and analyze computational complexity of the prototypical implementation. We shall see that our approach allows verification of complex preservation requirements in linear time w.r.t. constant model size and the number of found matches, where fully automated model construction adhering to these properties is known to be NP-complete. Technical issues of this article are mostly explained in prose text. The interested reader is referred to the appendix for the formal definitions.

II. Informal Description and Key Ideas

In this section we present the key ideas of our approach and motivate the design decisions with a particular model transformation example. For this purpose, we use the abstract website model depicted on the upper left-hand side of Fig. 1. We will see in the sequel that this seemingly simple example comprises very complex properties.

An abstract website (entity type WS) is connected to a structuring element (entity type SE) by means of the association $srcF$ (source folder), which at the same time contains website elements of type WE. Association multiplicities are omitted for clarity since they are not important for this introduction. Website elements are further distinguished by non-hypertext (NHT) and hypertext (HT) elements. One hypertext element is the "home" page (or index page) of the website, which is modeled by the $home$ association. A model instance of this (quite abstract) website representation is shown on the upper middle part of Fig. 1. It comprises one website-entity $w : WS$, one entity of type SE, two non-hypertext, and two hypertext entities. The home page element $e4$ is represented by the special icon. This abstract website instance already indicates that we require the home page element of a website to be contained in the corresponding structure element. This semantic property, however, cannot be expressed in the abstract model using associations. Therefore, the *concept* $EntryPoint$ is explicitly introduced on the instance level and shown on the upper right-hand side of Fig. 1. This concept is defined on a triple of elements comprising a website, a structure element and a hypertext element. We say that the concept $EntryPoint$ has the interface $WS \times SE \times HT$. In Fig. 1 we deliberately omitted all details in the area covered by $EntryPoint$ in order to indicate that it is *implemented* somehow using some appropriate language. In this case, First-Order Logic (FOL) is adequate. Other models may require *domain-specific* languages as they are used in MDD, for example.

On its bottom left-hand part, Fig. 1 depicts a *refined* website model. Model extensions are marked by dashed lines. In particular, the refined model introduces specific formats for non-hypertext ($Text$), hypertext ($HTML$), and structure elements (DIR, directories). Moreover, it allows for recursive structuring of directories. This refinement is, thus, a step from abstract navigatible structures towards a computer-based implementation and typical for all software engineering approaches.

The lower middle part depicts two model instances. Instance one refines the abstract instance by refining the entity types. The "home" page $e4$, e.g., is of type

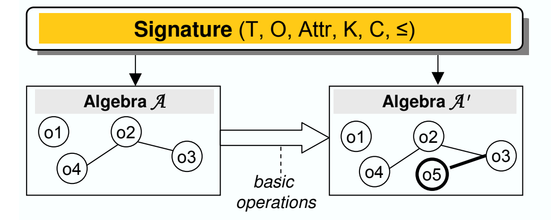


Figure 2. Stated-based semantics of basic operations.

$HTML$. This model instance is to be transformed, where the second model instance depicts the desired output of this transformation. It must adhere to substantially more constraints than the source instance. In particular, it contains two directories "resources" and "html", respectively, which separate non-hypertext and hypertext content. Subdirectories are permitted. This specific constraint is also implemented using a concept, which is, however not shown for clarity. The specific directory structure of the target model instance already indicates that preservation of link-consistency is challenging. Link consistency is a complex property that is difficult to handle solely by formal proofs.

The bottom right-hand part of Fig. 1 shows that both instances also implement the entry point concept, the preservation of which we use for demonstration purposes in the following. The specific structure of the transformation result affects the implementation of the $EntryPoint$ concept. In particular, we have to incorporate the "html" directory, which is indicated on the outer right-hand part by placing the corresponding icon inside the concept's area. Similar to graph transformation approaches we speak of different implementations in different *contexts*. In our approach, concept interfaces can be implemented by different contexts. Hence, concepts at the same time serve as grouping mechanism.

At this point, it is important to notice two facts in order to understand the notion of *preservation*. Firstly all elements in the abstract model instance have a unique correspondence in the refined instances. This is indicated by using equal element IDs. Moreover, and although the implementation changes, both refining implementations for the concept $EntryPoint$ still have the same interface. In our view, preserving the abstract concept $EntryPoint$ corresponds to a possible change of the concrete implementation while adhering to the abstract property itself. The interface objects serve as tracing points that connect the source and target model. This tracing is done using transformation operations, sequences of which construct partial model homomorphisms between the concrete representations of the corresponding abstract elements and their counterparts in the target context (dotted arrows). In other words: If the source elements satisfy the implementation of the concept $EntryPoint$ in the source context, then the transformation results must satisfy the $EntryPoint$, but for the implementation in the target context.

III. Basic Entity Model

After the informal survey, we now introduce the basic formal environment in which our approach is settled. We principally review those parts of [15], [16] that are relevant to clarify our ideas. The major notions introduced here comprise *signature*, *system states*, and *state changes*. Their relations are visualized in Fig. 2. Compared to [15] we go into deep at some points with informal explanations where this helps understanding the framework.

A. Syntactic Modeling Elements

Our approach only requires a minimal set of modeling elements including objects (atomic model entities), types, and concepts. That way we achieve a high degree of implementation independence on the one hand, but still maintain the ability to identify and trace concrete objects on the other hand. Types and sub-typing provide us with a structuring principle for objects, which is important for incorporating abstraction and refinement [10]. Notice, that application domains may require extensions, which at the same time can introduce side-effects. In [15], [16] we, e.g., use functions as well that can themselves cause changes to the model.

More formally a signature $\Sigma := (\mathcal{T}, O, Attr, K, C, \leq)$ comprises a set $\mathcal{T}$ of type symbols, elements of which uniquely type objects $o \in O$ (denoted by $type(o) = \tau$). Sub-typing is denoted by $\leq$. We abbreviate $type(o) \leq \tau$ by $o : \tau$, which is especially used for quantifier scopes later on. We further distinguish *static* types $\mathcal{T}_S \subseteq \mathcal{T}$ and class types $\mathcal{T}_C \subseteq \mathcal{T}$, which model static and dynamic parts of the models, respectively. Collection types $\mathcal{T}_{Coll} \subseteq \mathcal{T}_C$ are subsumed to the set of class types. Constants like the integer number 1 cannot have a static type. In contrast, object symbols are place holders for objects that can be created, transformed, and deleted. Class type specifications provide objects with attributes. We regard object symbols as IDs that are dynamically attached to objects on creation. Once used, an object's ID stays unchanged until this object is deleted. For better *syntactic* tracing of object life-cycles, objects are immutable. In particular, changing the value of an object's attribute is a transformation and yields a new object with a new ID.

K denotes the set of *concept symbols*, the elements of which each have a set of implementing *context symbols* $contexts(\mathcal{K}) \subseteq C$. Concepts and contexts are worth being introduced on their own right in Sect. IV.

B. System States

As already stated in the introduction, our approach constrains permissible model transformations by restricting valid combinations of the previous and subsequent *state*. Hence, we need notions for states and state change. We recall the modeling approach of [15], which has proven to be adequate for this purpose (see Sect. VI).

All model elements are interpreted in algebras $\mathcal{A}$, which at the same time serve as system states. This is very similar to algebraic graph transformation approaches [7].

Interpretations of objects and constants are called their *values* and must be an element of the *type domain* of their type. More precisely, an object's value comprises the values of all its attributes and the object's ID. This leads to the important consequence that no two objects have the same interpretation. This property is particularly important when carrying over changes on the interpretation level back to the syntactic level. In our implementation, which we will shortly introduce in Sect. VI-A, we interpret all syntactic entities by JAVA objects. These class instances serve as connector to real-world applications. With this interpretation, changes on the implementation level can always be carried over uniquely to the symbolic level since there is a unique object symbol for each JAVA class instance. As a by-product we get a convenient notion of object equality. Two objects are equal if they both have the same ID. More precisely, equality means identiy and equality checking can solely be done on the syntactical level. This considerably improves efficiency.

Notice that the special symbol $\perp$ (*undef*) is required to be part of all type domains. If an object symbol has semantics $\perp$, we say that o does not *exist* in the system.

Finally, the interpretations of concepts together with their contexts yield semantic predicates. We will, however, explain this in more detail in Sect. IV.

C. State Changes

In our preservation-centric environment, object histories are decisive for tracing the preservation of semantic properties. Therefore, we support three basic object-related state changes. Notice, however, that additional state changes can be reasonable if the basic entity model is extended (cf. [15]). In particular, we see the entity model as an algebra that evolves over time where object creation, object transformation, and object deletion can change states. An object may not be deleted if it is the attribute value of another object. This restriction avoids "dead" references. Moreover, the value of the host object would change, which is forbidden in our setting. More complex transformations are modeled as *sequences* of basic operations. The semantics of all basic operations is provided in the appendix and illustrated in Fig. 2. There, the algebra $\mathcal{A}'$ is generated from $\mathcal{A}$ by creating $o5$, the latter having $o3$ as an attribute. We consider the effect of all operations w.r.t. a preceding and a subsequent system state where the basic operation causes the minimal necessary model change w.r.t. its post-condition. In this case we say that a state $\mathcal{A}'$ is *subsequent to* $\mathcal{A}$. Post-conditions may constrain attributes of the created object, only. This avoids unwanted side-effects.

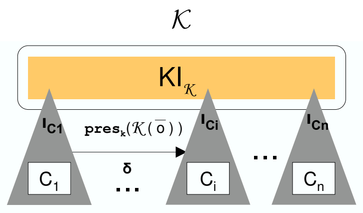
Example: Using graphs as objects and derivation steps as transformations we can model graph transformations. Post conditions can only constrain the produced graphs, which is typical for rule-based graph transformation [7].

This example shows the generality of our approach. We can trace graph transformations and their results w.r.t. the preservation of semantic properties. This very much corresponds to the approach in [17], which has been developed as a uniying theory for the structuring of graph transformations. We, however, extend it in certain ways since we allow the tracing of several transformations at a time that may be carried out using different graph transformation approaches. This is supported by our simple view of transformations being characterized through their object-to-object-mapping only. In contrast, [17] presupposes a notion of rule application and introduces ID-environments as structuring principle for transformations. In our approach, the notions of preservation and immutability of objects provide an inherent structuring and control mechanism. In [15] we have shown that these facilities are powerful enough to guide model construction for non-trivial examples.

At this point, we have defined the basic framework in which to execute model transformations. The next sections provide adequate language constructs for expressing semantic object properties (apart from attributes) and constraining model transformations.

IV. Specifying Semantic Properties: Contexts and Concepts

As explained in the introduction, our approach and – in particular – our view on preservation is to incorporate the definition of semantic properties (concepts) as well as support their refinement and implementation in different contexts. These concepts then can be specified to be preserved when models are transformed. This connection between concepts and their preservation is sketchily depicted in Fig. 3. Similar to programming languages, a concept $\mathcal{K}$ defines a *concept interface* KI that has to be implemented by all contexts $C_1, ..., C_n$ in $contexts(\mathcal{K})$. This implementation is called *embedding* and denoted by ι_{C_i}. Thus, contexts can be seen as "semantic plug-ins" for concepts. When transforming objects (in this case from context C_1 to context C_i), the concept $\mathcal{K}$ can be specified to be preserved using a *preservation constraint*. In our concrete example, we want to preserve properties like link-consistency or the entry point relation between a web-entity, a directory-entity, and a hypertext-entity. We support arbitrary implementation languages for



Concept name:	$EntryPoint$
Concept interface:	$\langle w : \alpha_1, d : \alpha_2, h : \alpha_3 \rangle$
	$\alpha_1 \leq WS \wedge \alpha_2 \leq SE \wedge \alpha_3 \leq HT$
Context name:	C^{eP}_{abstr}
embedding:	$srcF(w, d) \wedge home(w, h) \wedge con(d, h)$
Context name:	C^{eP}_{src}
embedding:	$srcF(w, d) \wedge home(w, h) \wedge \mathcal{K}^{con}(d, h)$
Context name:	C^{eP}_{trg}
embedding:	ϕ

Figure 3. Concept definition and concept preservation

the specification of concepts and only require that they adhere to the basic entity model explained in Sect. III. An example using graph- query languages and automata-based techniques for the preservation of link-consistency can be found in [16]. Typical other examples comprise FOPL formulae (e.g., in the form of OCL constraints) or tree languages based on tree automata. The latter can be used to constrain tree structures in a short and elegant way. We have used them to describe the desired directory structure in the target model. Other applications are DTD conformity of XML documents or constraints on class hierarchies in UML specifications.

The *concept interface* in Fig. 3 defines arity and typing constraints. The interface is the starting point for concept matching, where the typing can be seen as a pre-filter and speeds up the matching process. If this pre-filter is passed, an object tuple $\overline{o_i}$ satisfies the concept $\mathcal{K}$, if and only if one of the context embeddings evaluates to $true$ for these objects. The formal definitions for matching and concept satisfaction can be found in the appendix. Concept interfaces are used to trace object histories w.r.t. satisfaction of the corresponding concept $\mathcal{K}$ in possibly different contexts. This is possible only because all contexts of $\mathcal{K}$ have to implement this interface.

Example: The bottom part of Fig. 3 shows excerpts of the specification for the concept $EntryPoint$. Its interface is given by the sequence $\langle w : \alpha_1, d : \alpha_2, h : \alpha_3\rangle$ together with a type constraint. This is very much related to interfaces in programming languages. The three contexts implement this interface, where we omit the implementation for the target context C^{eP}_{trg} due to its complexity. The implementation for context C^{eP}_{src} exhibits the facility to structure concept definitions. This has two advantages. Firstly, it provides us with a means to reason over concept subsumption on the syntactic level. Secondly, it facilitates re-use and supports composition and decomposition of definitions. This becomes important when complexity grows (cf. [5]).

In the following sections, the language construct $\mathcal{K}[C]$ will be important. It reduces the evaluation of the concept $\mathcal{K}$ to the context C and can significantly speed up concept satisfaction checking. More specifically, a validity check of $\mathcal{K}$ for an object tuple $\overline{o_i}$ in Fig. 3 would require to test all n contexts if their embedding formula evaluates to $true$ for these objects. The just-mentioned language construct avoids this by fixing one of these n contexts.

V. Preservation of Model Properties

The notion of preservation can be found in different variations throughout the relevant literature [5], [7], [18]–[20]. Refinement, e.g., usually means a reformulation of system properties in the same language such that the newly specified system inherits all properties of the former system [5]. In node replacement approaches of graph transformation, authors usually speak of preservation w.r.t. those nodes and edges of the source graph that are not affected by a rule application. In [20] preservation of FOPL formulae under model homomorphisms

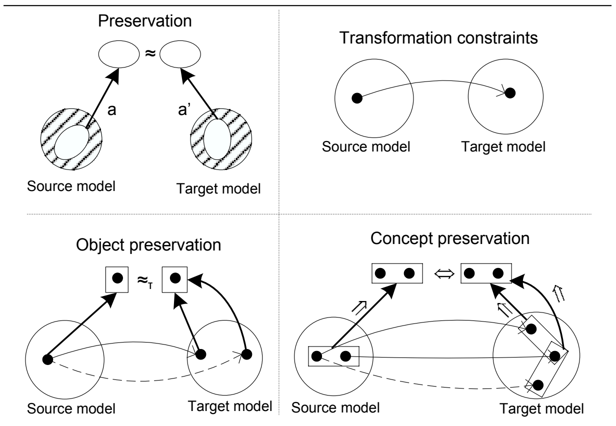


Figure 4. Preservation and effect of basic constraints

is studied with applications to database queries. In all these cases, however, preservation can be traced back to our explanations in Sect. II and the abstract preservation scheme shown in the upper left-hand part of Fig. 4. According to this scheme, preservation means to preserve a model property under abstractions a, a' w.r.t. the source and target model. Clearly, the real effect of this notion depends on the chosen abstraction functions and $\approx$ operator. If we replace the abstractions a, a' by implication $\Rightarrow$ and comparison by $\Leftrightarrow$, we have exactly the preservation requirement for the $EntryPoint$ concept of our running example. However, the concept of identity used in some rule-based graph transformation methods is too weak [7].

In the following we shortly recall the tripartite approach proposed in [15], where we have used (1) transformation constraints, (2) object preservation constraints, and (3) concept preservation constraints to restrict permissible model transformations. In particular constraints of type (2) and (3) implement different variants of abstraction and comparison. After that we introduce some extension and variations of this basic set of constraints that truly increase the expressive power of our approach. We will motivate these extensions out of our running example.

A. Transformation constraints

Basic transformation constraints of the form $o \mapsto \tau$ simply *enforce* the transformation of the object o to the target type τ. Notice that the other constraint types may well introduce transformation *obligations* [15]. Hence, one transformation constraint together with other preservation constraints can enforce whole transformation processes under suitable circumstances. The semantic effect of transformation constraints is visualized in the upper right-hand part of Fig. 4. The formal semantics can be found in the appendix. Satisfaction of transformation constraints and all other constraint types is evaluated w.r.t. subsequent states and a transformation sequence. In particular, this sequence must contain a transformation sequence leading from o to another object o' of type τ.

B. Object preservation constraints

Basic object preservation constraints focus on object contents. Read constraints $\mathtt{pres}_o(o \mapsto \tau, o[\tau'])$ as follows: “Whenever transforming o to τ ($o \mapsto \tau$), preserve the content of o arising when o is abstracted to type τ' ($o[\tau']$)”. The first parameter in parentheses is the *transformation assumption* and relates transformation constraints and object preservation constraints. An object preservation constraint holds, if the underlying transformation assumption holding implies that the source and target objects have similar contents when abstracted to τ'.

Example: In Sect. VI we will refine the entity model of our running example such that the type SE (structure element) has an attribute $name : SE \rightarrow String$. Since DIR (directory) is a sub-type of SE, it inherits this attribute. Furthermore, we introduce an attribute $subDirs : DIR \rightarrow Set\langle DIR\rangle$, which implements directory structuring. When transforming a directory, we will require the preservation of its name while possibly changing the set of sub-directories. This is assured by an object preservation constraint $\texttt{pres}_o(d \mapsto DIR, d[SE])$.

Regarding the abstract classification scheme of Fig. 4, abstraction in the context of object preservation yields type abstraction. Comparison $\approx$ yields an equivalence relation w.r.t. the values of those attributes that are defined on the respective type. A formal definition of this notion of *undistinguishability* is given in the appendix.

Notice that we use traces *of maximum length* from o to type τ in the constraint semantics provided in the appendix; we permit temporary inconsistencies. Consider, e.g., the evolution of a class specification in a UML class diagram through the software development process. Using our constraints, we can assure that the *final* version of the class specification meets the requirements. The phrase “Whenever” from above is also worth explicit notice. In Fig. 4 it is illustrated by two different transformation traces of o to the target type. With the semantics here, *all* traces have to satisfy the preservation requirement. In terms of the object’s history, all future paths that lead to a new object of the respective type satisfy this preservation requirement. Later on, we will introduce existential variants and show where they are necessary in the running example.

C. Concept preservation constraints

Basic concept preservation constraints

$$\texttt{pres}_k(\{\overline{o_{I_j} \mapsto \tau_{I_j}}\}, \mathcal{K}(\overline{o_i}), (C_s, C_t)),$$

incorporate the semantic preservation of object *relationships*. Read this constraint as follows: “Whenever transforming o_{I_1} to τ_1, ..., and o_{I_m} to τ_m, then the transformation result must 1) match the target context C_t and 2) satisfy $\mathcal{K}$ in C_t, if $\mathcal{K}(o_1, ..., o_n)$ was valid in the source context C_t”. In other words, preserving a concept means preserving a semantic relationship but possibly in different contexts (cf. Fig. 4).

Example: Recall that we want to preserve the $EntryPoint$ concept in our running example whenever we transform website entities. In Sect. II we have introduced the corresponding concept $\mathcal{K}^{eP}$ with different source and target contexts C^{eP}_{src} and C^{eP}_{trg}, respectively. In particular the embedding formula of the target context imposes a restricted directory structure on the target model. The constraint

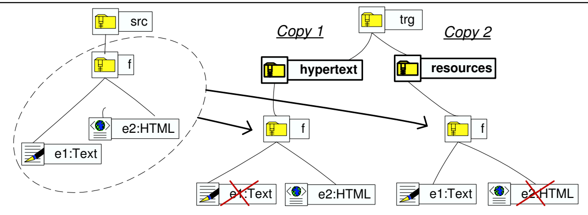


Figure 5. Use case for existential preservation constraints

$$\texttt{pres}_k(\{w \mapsto WS\}, \mathcal{K}^{eP}(w, d, h), (C^{eP}_{src}, C^{eP}_{trg}))$$

assures this preservation requirement.

Regarding the classification scheme of Fig. 4, abstraction is defined by upwards implication $\Rightarrow$, and comparison is defined by equivalence $\Leftrightarrow$. The semantics is given in the appendix. The strong correspondence $\Leftrightarrow$, however, is deliberately introduced instead of the weak form $\Rightarrow$. Consider the concept preservation constraint above. With our semantics it at the same time assures that all objects that do not satisfy the entry point concept will also not satisfy it in the target context. A weaker implicative variant of concept preservation can, however, still be specified using *extended* constraints (see Sect. V-E).

At this point, we have a basic set of constraints that is expressive enough in cases where object properties have to be preserved equally for all different and possibly branching traces. In the next section, we will give an example where variant building is important. This motivates the introduction of an existential variant of preservation constraints that exactly meets those requirements.

D. Existential Preservation constraints

Fig. 5 illustrates the necessity for an existential variant of concept preservation in the running example. Directories in the source model can contain non-hypertext as well as hypertext entities. This is forbidden in the target model, where all non-hypertext has to reside in “resources” and all hypertext has to reside in “html”. The right-hand part of Fig. 5 depicts this pattern. Suppose we want to preserve the source directory structure in both, the “resources” and “html” directory. Our semantics provided in the last sections, does not capture this. The main reason is that a source directory f has to be transformed twice – one copy for the folder “resources” and one copy for “html”. Now, suppose we specify the preservation of the file containment property

$$\texttt{pres}_k(\{f \mapsto DIR\}, \mathcal{K}^{con}(f, h), (C^{eP}_{src}, C^{eP}_{trg})).$$

In this case, the preservation semantics would require all suitable transformation results of f and h to satisfy this property in the target model (cf. Fig. 4). This is, however, not satisfiable if h is hypertext because hypertext must not reside in “html”. The transformed hypertext document can never be a sub-element of the copy of f in the “resource” directory.

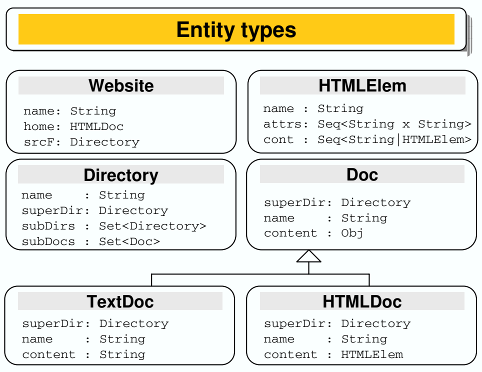


Figure 6. Entity types and example website model

Therefore, we introduce *existential* object- and concept preservation constraints. In fact, they exactly can express our preservation intention in this situation. The formal semantics is provided in the appendix. We explicitly annotate the existential variants, which yields the notations $\texttt{pres}^{\exists}_{o}(o \mapsto \tau, o[\tau'])$ and $\texttt{pres}^{\exists}_{k}(\{\overline{o_{I_j} \mapsto \tau_{I_j}}\}, \mathcal{K}(\overline{o_i}), (C_s, C_t))$.

Example: In our running example, we can specify the preservation of the source directory structure in both, the “resources” and “html” directory using the existential concept preservation constraint $\texttt{pres}^{\exists}_{k}(\{d \mapsto WE\}, \mathcal{K}^{con}(dir, d), (C^{con}_{src}, C^{con}_{src}))$. Since NHT and HT are both sub-types of WE, we do not need a separate constraint for each of those types.

E. Extended Preservation constraints

Since our constraints are related to objects, large models will imply large constraint sets. Moreover, users usually do not want to provide a single constraint for every simple preservation intention. They rather group objects to equivalence classes w.r.t. a certain preservation requirement. Therefore, we introduce *extended preservation constraints* of the form $\forall \overline{x_i : \tau_i} \bullet \phi(\overline{x_i}) \Rightarrow c(\overline{x_i})$ where c is a basic constraint (no matter if it is existential or not) and ϕ is called the *guard*. In particular, the initial part $\forall \overline{x_i} \bullet \phi(\overline{x_i})$ selects those objects of the source state the constraint has to be applied to. Semantically, an extended constraint can be thought of as a *set* of basic constraints. The full semantics is provided in the appendix. In the examples provided in the next sections we will solely use extended constraints, which shortens specifications in a nice way.

VI. Case Studies and Results

In this section, we evaluate our approach using two case studies stemming from different application domains. Along with these case studies, we present parts of the respective preservation specification. As main result of this section, we will see that our approach can be used to check the preservation of complex recursive properties in linear time w.r.t. to the amount of found matches and a constant number of objects in the system. We start by introducing the prototypical system implementation and the test setup that have been used to run the case studies.

A. Prototypical implementation and test configuration

We have implemented an experimental system that facilitates to specify significant object properties and preservation requirements of the fashion described in this paper. In particular, all entity types are interpreted as JAVA-classes, objects of the respective types are mapped to instances of the corresponding JAVA-class. Preservation constraints can be specified in the system, and transformations are executed using registered transformation classes. In particular, transformations run inside an own environment such that the system is aware of newly created or transformed objects. During the transformation process, users can activate the constraint checking at any time. The system then reports constraint violations.

Tests in the case studies have been carried out on a Windows XP - workstation with 1 GB RAM and an Intel P4 CPU of 3.2 GHz. Although it slows down the system, no caching of evaluated formulae etc. has been used. In this way we receive unadulterated results.

B. Case study 1 – Transformation of website models

In this section, we present the first case study. It is directly related to our running example, but has been carried out using the refined entity model shown in Fig. 6. There, the “home” page and source directory of type $Website$ are modeled by corresponding attributes. Directories have a name, a super-directory and two sets distinguishing sub-directories and sub-documents. This implementation corresponds to a double linked list and is a direct realization of a $0..1$ to n relationship. We have added an explicit content type $HTMLElem$ for HTML documents. This type is recursive and implements tree-kinded document content. Type Doc is a super-type of all documents and stores a name, the corresponding super-directory and a content-object.

The necessary concepts have already partially been introduced. We omit the formal definitions because they are not relevant for the understanding of this article. We refer the interested reader to [15], [16] for examples. Notice, however, that many of the intellectual challenges that are induced by complex formal specifications may be settled in the proper definition of concepts in our method. But, this is no inherent problem of our approach. It rather can be found in one or the other form in all formal model transformation methods.

The concept $\mathcal{K}^{con}$ models containment in a directory and can easily be implemented using the attributes shown

(1) $\forall d : Directory \bullet$
$\texttt{pres}_{o}(d \mapsto Directory, d[Directory\{name\}])$

(2) $\forall h : HTMLDoc, e : HTMLElem \bullet$
$\forall d : Doc \ \bullet \mathcal{K}^{el}(e, h) \Rightarrow$
$\texttt{pres}_{k}(\{h \mapsto HTMLDoc\}, \mathcal{K}^{link}(h, e, d))$

(3) $\forall w : Website, h : HTMLElem, d : Directory \bullet$
$\texttt{pres}_{k}(\{w \mapsto Website\}, \mathcal{K}^{eP}(w, h, d), (C^{eP}_{src}, C^{eP}_{trg}))$

(4) $\forall d : Doc, dir : Directory \bullet$
$\texttt{pres}^{\exists}_{k}(\{d \mapsto Doc\}, \mathcal{K}^{con}(dir, d), (C^{con}_{src}, C^{con}_{src}))$

Figure 7. Constraints for case study 1

in Fig. 6. The $EntryPoint$ concept is denoted by $\mathcal{K}^{eP}$. Preservation of link consistency will be specified using the concept $\mathcal{K}^{link}$, which is implemented w.r.t. a source HTML document, a link anchor of type $HTMLElem$, and a target document (cf. [16] implementation details). Excerpts of the formal constraints for this example are listed in Fig. 7. Constraint (1) assures all directory names stay unchanged. Constraint (2) assures link consistency. The guard $\mathcal{K}^{el}(e,h)$ evaluates to true, if the element e is contained in the HTML document h. Constraint (3) assures the target directory structure conforms to the target website model. Finally, we preserve the source folder hierarchy in two variants using constraint (4). Notice that the specification is short due to the usage of extended constraints. This leads to concise specification on the one hand, but on the other hand hides the complexity and potential number of basic constraints that have to be checked. Hence, caution has to be taken when specifying the preservation constraints to avoid duplicates and implications among them.

Fig. 8 shows the test results. We have measured the time for constraint checking (y-axis) w.r.t. three system setups. These system setups had an equal amount of objects in the system ($\approx$ 1700) but were generic in the three parameters given as triples (n_1, n_2, n_3) on the right-hand part of Fig. 8. There, n_1 is the number of directories in the system, n_2 the number of $HTML$ documents, and n_3 contains the number of non- HTML documents. We have tested each of these three system setups with five different degrees of connectivity ranging from 20 to 320 HTML links (x-axis). Concerning the preservation requirements of Fig. 7, this is the major source of inefficiency because the resulting link graph will be highly connected. The largest model comprises 320 links while containing 20 HTML documents and 60 non-HTML documents, only. In contrast, the number of object preservation constraints grows linear with model size by definition. The same is true for the directory hierarchy due to its tree structure.

Fig. 8 shows a linear curve for all three system setups w.r.t. an increasing degree of connectivity. Whenever we double the number of links in the system, the evaluation time for constraint preservation approximately doubles as well. This is not surprising, since we trace the found matches in the source model and automatically know which objects to check for concept satisfaction in the target model. As expected, the time for checking the preservation of the other constraints is a constant factor in each of the setups.

On the other hand, complexity even grows linear w.r.t. the number of objects in the system. This cannot be expected in general, since we, e.g., need to compare the link property between two documents, which usually would let us assume a complexity growth of n^2 at least. Here, the guard, which we have mentioned above, is decisive. It avoids a lot of unnecessary link evaluations. In absence of the guard, the system would check the linking concept for all combinations of $HTML$-documents, link anchors, and link targets even if the link anchor is no part of the $HTML$ document.

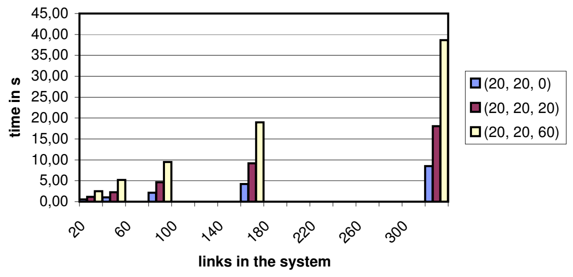


Figure 8. Results for case study 1

C. *Case study 2 – Transformation of component interaction specifications*

This case study describes the preservation of communication flows in component interaction specifications. In particular, we suppose that these specifications are still under development and can be changed, extended etc. by human interaction. This situation usually occurs in software development processes. We will first shortly provide the underlying entity types and explain how we model communication flows. After that we provide the relevant constraints and measure the verification time in differently sized models.

The upper part in Fig. 9 shows the entity types. Component interaction specifications are modeled by type $Specification$ having three attributes – $components$ (the set of components belonging to this specification), $events$ (the set of specified events), and $eventDels$ (a set of event delegation specifications from a hosting component to a sub-component). Type $Event$ models events and comprises an id-attribute. $Component$s have a *name*, an interface $inInterface$ for incoming events (listed by the corresponding event's id), and an interface $outInterface$ including all events that can be issued by the component. Component structure is reflected by the $subcomps$-attribute, which lists all component names that are *directly* below the corresponding component in the hierarchy.

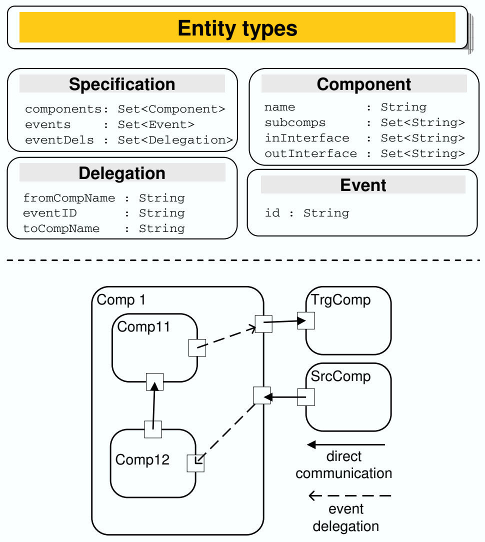


Figure 9. Entity types and example communication flow

(1) $\forall c : Component \bullet$
$\texttt{pres}_o(c \mapsto Component, c[Component\{name\}])$
(2) $\forall s : Specification, c_s : Component \bullet$
$\forall c_t : Component \bullet$
$\texttt{pres}_k(\{s \mapsto Specification\}, \mathcal{K}^{subc}(s, c_s, c_t))$
(3) $\forall s : Specification, c_s : Component \bullet$
$\forall c_t : Component, e : Event \bullet$
$\texttt{pres}_k(\{s \mapsto Specification\}, \mathcal{K}^{comm}(s, c_s, e, c_t))$

Figure 10. Constraints for case study 2

When changing the specification, the component names must not change. Also the component hierarchy must not change in a manner that does not preserve sub-component relationships. Finally, we want to preserve the following: If two components can interact using an event e, this interaction is still possible in the target specification.

The concept $\mathcal{K}^{subc}$ expresses the subcomponent-relationship w.r.t. two components that are part of a given specification. This relationship is directly reflected by the $subcomps$-attribute and, thus, quite easily to express. The concept $\mathcal{K}^{comm}$ is considerably more complex. It expresses that, given a specification, a component can communicate with another component using a given event. These communication flows can be quite complex as is indicated for the example component hierarchy in the lower part of Fig. 9. In particular, events can be delegated hierarchically and event flows must adhere to the different in- and out- interfaces that are offered by the components. We do not go into detail with the specification of this property but mention that we have used automata-based techniques as described in [16].

Fig. 10 shows the preservation constraints. Constraint (1) assures preservation of all component names for those components that are transformed. Constraints (2) and (3) assure the above-mentioned requirements.

We have chosen a non-trivial refactoring operation in order to test the performance of constraint checking in this example. In particular, we carry out a behavior-preserving event-renaming. When renaming an event, all interfaces accepting or offering this event have to be adapted. This is easily expressed but causes validity of all communication flows to be re-checked by the system. Caused by the interface updates, the component objects themselves are changed. Therefore, constraint (1) is re-checked as well. Fig. 11 depicts the test results in the same fashion as was shown in the last section.

Analogous to case study one, we have measured the time for constraint checking (y-axis) w.r.t. three system setups. The generic parameters (n_1, n_2) comprise the number of components and events, respectively. The y-axis again indicates the degree of connectivity and comprises all matchings for component communication. We have constructed the models such that constantly 10 percent of the communication paths were related to the event that was renamed. Since the preservation of communication is very much related to the preservation of link consistency, one can expect a similar system behavior to case study one. At first sight, the results of Fig. 11 only partially confirm this assumption. In all system setups we measured linear complexity w.r.t. a constant number of objects in the system and a growing degree of connectivity. We, however, have a considerably worse behavior when the number of components and events grows. This can be traced back to our "inefficient" specification. We have used no guards in the specification, which causes the system to check *all* combinations of components and events whether they satisfy the preservation requirements. This shows that – up to now – specifications have to be expressed carefully w.r.t. efficiency. This at the same time points up an important direction for future research where static analysis can detect inherent efficiency problems or implications among constraints.

VII. Conclusion and Outlook

In this article, we have presented the preservation-centric model transformation approach of [15], [16] from a verification-oriented perspective. This has been motivated by two application domains, where either automated and still formally correct model transformations are too complex for being carried out efficiently in the large, or model changes are induced by human interaction that does not adhere to pre-defined rules. The latter is a typical case in software engineering where the evolution of UML models from analysis to design and implementation models cannot be automated.

We have provided two case studies drawn from different domains and have shown that we can trace the preservation of complex properties like link-consistency or communication flows in component interaction specifications in linear time w.r.t. the number of matches and constant model size. Apart from these evaluation results, we have introduced a new existential variant of preservation constraints and motivated this extension with the running example. This type of constraint allows one to specify different preservation requirements for different branches of an object's history. In this way, the expressive power of the constraints introduced in [15] is increased.

We have already pointed up some research directions in [15], [16]. Those of [15] address technical issues related to our constraint language. We are still working on a formal theory for static constraint analysis to support the detection of implications among constraints. This is likely to accelerate constraint checking and potential model construction w.r.t. given constraint sets. As stated in [16], integration of additional implementation languages

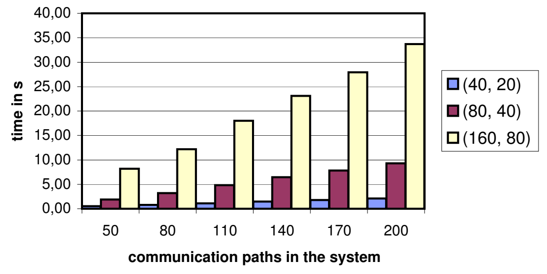


Figure 11. Results for case study 2

beyond FOPL is being done on a stepwise basis. So far, we can handle FOPL, automata, and tree automata. All of these languages have been applied in the case studies presented here.

These technical research issues, however, will be complemented by further practical studies. In particular, it will be interesting to find and study more practical applications of our existential constraints. We think of objects that are subject to branching workflows as they, e.g., occur in software configuration management or version control systems. Apart from that, we have a running Master's project in which we test our approach with a document-related model transformation. We have implemented a transformation from parts of the XHTML standard to the Open Document Format (ODF) standard. We use Cascading Style Sheets (CSS) to support layout for both the source and target documents. Since ODF has been developed as a document exchange format and combines the diverse existing XML-related and non-XML-related document formats, this case study is particularly interesting. Complex properties in this case study comprise link-consistency, preservation of layout, schema-conformity, and preservation of "reproducibility" of the source document. Overall, we have identified around fifty extended preservation concstraints that have to be met. Initial tests have shown that our approach can directly support the development of the transformations because we are able to pinpoint inconsistencies and their reasons.

REFERENCES

[1] P. Fraternali and P. Paolini, "Model-driven development of web applications: the AutoWeb system," *ACM Trans. Inf. Syst.*, vol. 18, no. 4, pp. 323–382, 2000.

[2] A. Rensink, Schmidt, and D. Varró, "Model checking graph transformations: A comparison of two approaches," in *Int. Conf. on Graph Transformations (ICGT)*, ser. Lecture Notes in Computer Science, vol. 3256. Berlin: Springer Verlag, 2004, pp. 226–241.

[3] T. Szemethy, G. Karsai, and D. Balasubramanian, "Model transformations in the model-based development of real-time systems," in *Proc. 13th IEEE Int. Symp. on Eng. of Comp. Based Sys. (ECBS'06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 188–196.

[4] S. Paunov, J. Hill, D. Schmidt, S. D. Baker, and J. M. Slaby, "Domain-specific modeling languages for configuring and evaluating enterprise dre system quality of service," in *Proc. 13th IEEE Int. Symp. on Eng. of Comp. Based Sys. (ECBS'06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 196–208.

[5] T. Mens, K. Czarnecki, and P. V. Gorp, "04101 discussion – a taxonomy of model transformations," in *Language Engineering for Model-Driven Software Development*, ser. Dagstuhl Seminar Proc. no. 04101. IBFI, Schloss Dagstuhl, Germany, 2005.

[6] T. Mens, S. Demeyer, and D. Janssens, "Formalising behaviour preserving program transformations," 2002.

[7] G. Rozenberg and et.al., *Handbook of Graph Grammars and Computing by Graph Transformation*. New Jersey: World Scientific Publishing, 1997.

[8] A. Schürr, A. Winter, and A. Zündorf, "The PROGRES approach: language and environment," pp. 487–550, 1999.

[9] K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, G. Taentzer, D. Varró, and S. Varró-Gyapay, "Model transformation by graph transformation: A comparative study," in *MTiP 2005, Int. Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005)*, 2005.

[10] J. Woodcock and J. Davies, *Using Z: Specification, Refinement, and Proof.* New Jersey: Prentice Hall, 1996.

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

[12] M. Arenas, W. Fan, and L. Libkin, "What's hard about xml schema constraints?" in *DEXA '02: Proc. of the 13th Int. Conf. on Database and Expert Systems Applications*. London, UK: Springer-Verlag, 2002, pp. 269–278.

[13] J. Scheffczyk, U. M. Borghoff, P. Rödig, and L. Schmitz, "Managing inconsistent repositories via prioritized repair actions," in *Proc. of the ACM Symp. on Doc. Eng. (DocEng 2004)*, 2004, pp. 137–146.

[14] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein, "xlinkit: a consistency checking and smart link generation service," *ACM Trans. Inter. Tech.*, vol. 2, no. 2, pp. 151–185, 2002.

[15] T. Triebsees and U. M. Borghoff, "A theory for model-based transformation applied to computer-supported preservation in digital archives," in *Proc. 14th Ann. IEEE Int. Conf. on the Eng. of Comp. Based Systems (ECBS'07)*. Tucson, AZ, USA: IEEE Computer Society Press, March 2007.

[16] T. Triebsees and U. M. Borghoff, "Towards automatic document migration: Semantic preservation of embedded queries," in *Proc. of the Int. ACM Symp. on Doc. Eng. (DocEng07)*. NY: ACM Press, 2007, to appear.

[17] H.-J. Kreowski and S. Kuske, "On the Interleaving Semantics of Transformation Units—A Step into GRACE," in *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*, vol. 1073. Springer-Verlag, 1996, pp. 89–106.

[18] I. Porres, "Rule-based update transformations and their application to model refactorings." *Software and System Modeling*, vol. 4, no. 4, pp. 368–385, 2005.

[19] J. M. Wing and J. Ockerbloom, "Respectful type converters," *IEEE Trans. on Software Eng.*, vol. 26, pp. 579–593, July 2000.

[20] A. Atserias, A. Dawar, and P. G. Kolaitis, "On preservation under homomorphisms and unions of conjunctive queries," *J. ACM*, vol. 53, no. 2, pp. 208–237, 2006.

Thomas Triebsees holds a diploma degree in Information Sciences from the Universität der Bundeswehr München, Germany (2002). After his studies he worked as a Quality Management Officer at the Department for Software Maintenance at the Signal School of the German Army. Since 2004 he has been working for the Department of Information Sciences at the Universität der Bundeswehr München as a research assistant and PhD candidate. His research interests include document management, model transformation, and digital archiving. In these areas he has published in several international conference proceedings.

APPENDIX

For the formal understanding, we owe the semantics for concept validity, interpretations of concept symbols in $\mathcal{A}$, the semantics of basic operations, and the semantics for the different variants of preservation constraints. They have in excerpts been provided in [15]. Here we re-define them using explicit traces for clarity.

A. Concept Semantics

Given a signature Σ. A *basic algebra* $\mathcal{A}^b$ for Σ comprises the sets $\mathcal{T}^{\mathcal{A}^b}$, $O^{\mathcal{A}^b}$, and $Attr^{\mathcal{A}^b}$. An algebra $\mathcal{A}$ for Σ comprises a basic algebra for Σ and interpretations for the concept symbols in Σ. We will not distinguish between interpretations in an algebra or in its basic algebra, respectively, if this leads to no confusion. Hence, $o^{\mathcal{A}^b}$ and $o^{\mathcal{A}}$ have the same meaning. Given a concept symbol $\mathcal{K}$ and a basic algebra $\mathcal{A}^b$ for Σ, $\mathcal{K}^{\mathcal{A}}$ yields

$$\mathcal{K}^{\mathcal{A}} = \bigcup_{C \in contexts(\mathcal{K})} \{\overline{o_i^{\mathcal{A}}} \mid \mathcal{A}^b \models \iota_C[\overline{o_i^{\mathcal{A}}/x_i}]\}$$

where $\overline{x_i}$ is the interface of $\mathcal{K}$.

Concept satisfaction: Given a signature Σ, an algebra $\mathcal{A}$, a concept symbol $\mathcal{K}$, a context C, and n object symbols $o_1, ..., o_n$. Then concept satisfaction of $\mathcal{K}$ in C by $\overline{o_i}$ is defined as follows:

$$\begin{aligned} \mathcal{A} \models \mathcal{K}(\overline{o_i})[C], \text{ iff } & C \in contexts(\mathcal{K}), \\ & type(\overline{o_i}) \in \widehat{\tau}(\overline{x_i}), \text{ and } \overline{o_i^{\mathcal{A}}} \in \mathcal{K}^{\mathcal{A}} \end{aligned}$$

where $\overline{x_i}$ is the interface of $\mathcal{K}$ and $\widehat{\tau}(\overline{x_i})$ is the set of all types that satisfy the corresponding type condition; we check well-typedness already on the syntax level.

B. State Change Semantics

Given a signature Σ, two states $\mathcal{A}$, $\mathcal{A}'$ for Σ, and an object symbol o. Then *subsequence of* $\mathcal{A}'$ to $\mathcal{A}'$ *w.r.t. a basic operation* is defined as follows:

Object creation:

$\models (\mathcal{A} \rightsquigarrow \mathcal{A}', \mathtt{cre}(o))$, iff $o^{\mathcal{A}} = \bot \wedge o^{\mathcal{A}'} \neq \bot \wedge$
$\mathcal{A}' \models post(\mathtt{cre}(o'))$ and no other changes to $\mathcal{A}^b$

Object transformation:

$\models (\mathcal{A} \rightsquigarrow \mathcal{A}', \mathtt{tr}(o \mapsto o'))$, iff $o^{\mathcal{A}} \neq \bot \ \wedge \ o'^{\mathcal{A}'} \neq \bot \wedge$
$\mathcal{A} \models pre(\mathtt{tr}(o \mapsto o')) \ \wedge \ \mathcal{A}' \models post(\mathtt{tr}(o \mapsto o'))$
and no other changes to $\mathcal{A}^b$

Object Deletion:

$\models (\mathcal{A} \rightsquigarrow \mathcal{A}', \mathtt{del}(o))$, iff $o^{\mathcal{A}} \neq \bot \ \wedge \ o^{\mathcal{A}'} = \bot \wedge$
$\mathcal{A} \models pre(\mathtt{del}(o))$ and no other changes to $\mathcal{A}^b$

Notice that the changes themselves affect the *basic* algebras. The concept interpretations than are adapted according to the state change. Hence, basic operations can have side-effects on the whole state and our semantics assures basic operations yield the "least fitting" next state.

A sequence $\Delta := \langle op_1, ..., op_n \rangle$ of basic operations is a *transformation algorithm*. Subsequence of states w.r.t. transformation algorithms is defined as follows:

$$\begin{aligned} & \models (\mathcal{A} \rightsquigarrow \mathcal{A}', \langle\rangle), \text{ iff } \mathcal{A} = \mathcal{A}' \\ & \models (\mathcal{A} \rightsquigarrow \mathcal{A}', \langle op_1, ..., op_n \rangle), \text{ iff} \\ & \quad \exists \mathcal{A}_0, ..., \mathcal{A}_n \bullet (\ \mathcal{A}_0 = \mathcal{A} \wedge \mathcal{A}_n = \mathcal{A}' \wedge \\ & \qquad \forall i \in \{1, ..., n\} \bullet \ (\mathcal{A}_{i-1} \rightsquigarrow \mathcal{A}_i, op_i)\ ). \end{aligned}$$

C. Constraint Semantics

Given two objects o and o' and a type τ. Then o and o' can be abstracted to τ, if $type(o) \leq \tau \wedge type(o') \leq \tau$. Let $attrs(\tau)$ denote the set of attributes defined for τ (including inheritance w.r.t. $\leq$). Then *indistinguishability* w.r.t. τ is defined as follows:

1. $type(o) \in \mathcal{T}_S \wedge type(o') \in \mathcal{T}_S : o \approx_\tau o'$, iff $o = o'$
2. $type(o) \in \mathcal{T}_C \wedge type(o') \in \mathcal{T}_C : o \approx_\tau o'$, iff
$\forall a : \tau \rightarrow \tau_a \in attrs(\tau) \bullet a(o) \approx_{\tau_a} a(o')$

Traces: We use traces to keep track of object histories and define them inductively w.r.t. a sequence Δ of basic operations:

1. $\mathtt{traces}_0(\Delta) := \{\langle \mathtt{tr}(o \mapsto o') \rangle | \mathtt{tr}(o \mapsto o') \in \Delta\}$
2. $\mathtt{traces}_{i+1}(\Delta) := \mathtt{traces}_i(\Delta) \cup$
$(\mathtt{traces}_i(\Delta) \otimes \mathtt{traces}_0(\Delta))$

where $\otimes$ denotes the concatenation of all possible traces. It is defined for traces $tr_1 = \langle op_1^1, ..., op_n^1 \rangle$ and $tr_2 = \langle op_1^2, ..., op_m^2 \rangle$, iff $op_n^1 = \mathtt{tr}(o \mapsto o')$ and $op_1^2 = \mathtt{tr}(o' \mapsto o'')$ for some o, o', o''. $\mathtt{traces}(\Delta)$ denotes the reflexive transitive closure of the definition above. For traces $tr = \langle \mathtt{tr}(o_1 \mapsto o'), ..., \mathtt{tr}(o_{n-1} \mapsto o_n) \rangle$ the *trace result* $tr(o)$ yields o_n, if $o = o_1$. Otherwise, $tr(o) = o$.

Basic preservation constraints: The $\forall$-variant is given as follows:

Transformation constraints:
$(\mathcal{A}, \Delta, \mathcal{A}') \models o \mapsto \tau$, iff $(\mathcal{A} \rightsquigarrow \mathcal{A}', \Delta) \wedge$
$\mathtt{trace}^{\mathtt{m}}(o, \Delta, \{\tau' \in \mathcal{T}_C | \tau' \leq \tau\}) \neq \emptyset$

Object preservation constraints:
$(\mathcal{A}, \Delta, \mathcal{A}') \models \mathtt{pres}_o(o \mapsto \tau, o[\tau'])$, iff
$(\mathcal{A}, \Delta, \mathcal{A}') \models o \mapsto \tau \ \Rightarrow$
$\forall tr \in \mathtt{trace}^{\mathtt{m}}(o, \Delta, \{\tau'' \in \mathcal{T}_C | \tau'' \leq \tau\}) \bullet (o \approx_{\tau'} tr(o))$

Concept preservation constraints:
$(\mathcal{A}, \Delta, \mathcal{A}') \models \mathtt{pres}_k(\{\overline{tc_{I_j}}\}, \mathcal{K}(\overline{o_i}), (C_s, C_t))$, iff
$(\mathcal{A}, \Delta, \mathcal{A}') \models \bigwedge \overline{tc_{I_j}} \Rightarrow$
$\forall tr \in \mathtt{trace}^{\mathtt{m}}(\overline{o_i}, \Delta, \widehat{\tau}(\overline{x_i})) \bullet \mathcal{A} \models \mathcal{K}(\overline{o_i}) \Leftrightarrow \mathcal{A}' \models \mathcal{K}(tr(\overline{o_i}))$

where I is an index set, $\overline{x_i}$ is the interface of $\mathcal{K}$, and $\mathtt{trace}^{\mathtt{m}}(o, \Delta, ts)$ contains all *inclusionmaximal* traces in $\mathtt{traces}(\Delta)$ that lead from o to any of the tpyes in ts (including sub-typing). Hence, all traces $tr \in \mathtt{trace}^{\mathtt{m}}(o, \Delta, ts)$ must satisfy the following:

1. $\exists \tau \in ts \bullet type(tr(o)) \leq \tau\}$, and
2. $\neg \exists tr' \neq tr \in \mathtt{trace}^{\mathtt{m}}(o, \Delta, ts) \bullet tr' \subseteq tr$

Transformation constraints are satisfied if there *exists* a matching trace. If all transformation assumptions are satisfied, object and concept preservation constraints assure preservation of this property for *all* traces.

Existential constraints: The semantics for existential object and concept preservation constraints is determined from the definition above by replacing the $\forall$-quantor for traces by an $\exists$-quantor.

Extended Constraints: The semantics of extended constraints is given by

$$\begin{aligned} (\mathcal{A}, \Delta, \mathcal{A}') \models \forall \overline{x_i} \bullet \phi(\overline{x_i}) \Rightarrow c(\overline{x_i}), \text{ iff } (\mathcal{A}, \Delta, \mathcal{A}') \models c \\ \text{holds for all } c' \in \{c[\overline{o_i}/\overline{x_i}] \mid \mathcal{A} \models \phi[\overline{o_i}/\overline{x_i}]\} \end{aligned}$$

where c and c' are basic constraints.