Tarski: A Platform for Automated Analysis of Dynamically Configurable Traceability Semantics

Ferhat Erata
Information Technology Group
Wageningen University
Wageningen, The Netherlands
ferhat@computer.org

Moharram Challenger
Information Technology Group
Wageningen University
Wageningen, The Netherlands
moharram.challenger@wur.nl

Bedir Tekinerdogan
Information Technology Group
Wageningen University
Wageningen, The Netherlands
bedir.tekinerdogan@wur.nl

Anne Monceaux
System Engineering Platforms
Airbus Group Innovations
Toulouse, France
anne.monceaux@airbus.com

Eray Tüzün
Technology & Academy
Directorate
HAVELSAN Inc., Turkey
etuzun@havelsan.com.tr

Geylani Kardas
International Computer
Institute
Ege University, Izmir, Turkey
geylani.kardas@ege.edu.tr

ABSTRACT

Traceability can be defined as the degree to which a relationship can be established among work products of the development process. Traceability is important to support the consistency and likewise to ensure that a system is understandable, maintainable and reliable. Several approaches have been proposed to model traceability elements and reason about them by extending a predetermined set of possible trace links with fixed semantics. Furthermore, they do not cope with the need for dynamic adaptation and configuration of traceability semantics. However, different project types usually require various ways of tracing the system to obtain richer and precise automated traceability analysis. In this paper, we introduce a novel approach with its supporting platform which enables the user to rigorously configure the system based on project-specific needs and interactively specify the semantics of traceability elements. The semantics of traceability elements are formalized using first-order relational logic, which are used to facilitate different form of automated analysis. The use of the proposed approach and the corresponding tool is described within the context of an industrial application lifecycle management process.

Keywords

Traceability; Domain-Specific Modeling; Formal Trace Semantics; Automated Analysis; Alloy; KodKod

CCS Concepts

•**Software and its engineering** → **Consistency; Traceability;** *Specification languages; Formal methods;*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC 2017,April 03-07, 2017, Marrakech, Morocco
Copyright 2017 ACM 978-1-4503-4486-9/17/04. . . $15.00
http://dx.doi.org/10.1145/3019612.3019747

1. INTRODUCTION

Development process of software-intensive systems requires implementation, usage, and maintenance of a huge number of work products such as specifications, models, code, and test cases. These artefacts are usually related in different and complex ways. Tracing the dependencies among them is important to support the consistency and likewise to ensure that a system is understandable, maintainable and reliable.

To support traceability, a number of different types of traceability models have been used with their own tools [10, 11, 17, 19]. A traceability model defines the types of the trace relations, such as *depends-on*, *refines*, *satisfies* etc. For ensuring traceability very often a fixed traceability model is employed by the existing tools, that define a predetermined set of possible trace relations and their corresponding semantics. For homogeneous systems with a predetermined semantics that does not need to change, adopting a single static traceability model is usually not a serious problem. However, in the case of dealing with complex heterogeneous systems, instead of a one-size-fits-all approach, it is required to enable the adoption of different traceability models with their own specific semantics, and herewith the corresponding different traceability analysis approaches. On its turn, this requires a platform in which the traceability model can be easily adapted to support a customized traceability analysis. Traceability analysis can be carried out for different purposes including consistency analysis, impact analysis and repairing broken traces.

In this paper, we introduce a novel approach with its supporting platform, Tarski[1], which enables the user to rigorously configure the system based on project-specific needs and interactively specify the semantics of traceability elements. Their semantics are formalized using first-order relational logic, which are used to facilitate different form of automated analysis such as consistency checking, reasoning about trace-relations and trace-element discovery.

The approach and the corresponding platform are described

[1]The platform's name is inspired by Alfred Tarski's foundational work on the relational calculus

for traceability analysis within the context of application lifecycle management (ALM) process which is a paradigm for integrating and managing the various activities related to the governance, development and operations of software applications [4]. Poor traceability between artifacts which are produced in different stages of software development will lead to inconsistencies and affect the success of the projects. Oftentimes, trace-links get unsynchronized since those artefacts evolves independently and are subject to change during their lifetimes.

The rest of the paper is organized as follows. In section 2, we describe the approach step by step, providing background information and comparing our contributions with the traceability literature. Section 3 briefly describes the platform architecture, explains the platform components and gives some availability details about the open source platform. Section 4 describes the industrial use case on which the approach and the platform is applied. In section 5, we compare Tarski platform with widely-used industrial standards and approaches which provide automated analysis support in Traceability. Finally section 6 presents our conclusions and gives several remarks about the future work.

2. APPROACH

2.1 Traceability Domain Model

In order to understand the ramifications of changes over development cycles, and to obtain the rationale of design decisions, it is necessary to record the semantic relationships between artefacts [18, 26]. To realize the reasoning about traceability systematically and effectively, a proper abstraction is needed. To this end, in the following we describe the basic terms that we adopt from the literature in traceability [10, 11, 17, 19]. Using those terms we constitute a domain-specific model in Figure 1 (using MOF [23] notation) with some minor modifications and several contributions to the literature.

Traceability Information. The term traceability information applies to a wide spectrum, ranging from handcrafted intra-model links, such as stereotyped UML[2] associations, to tool generated inter-model links relating elements of the source and target models of an automatic transformation [19]. In the traceability framework of Tarski platform *Traceability Information* represents a traceability instance of a running platform, being corollary of the set of unique *trace-element*s.

Trace-elements. We use the notion of *Trace-element* to refer to all entities of traceability, which is therefore an abstract concept. Technically speaking, *trace-link* and *trace-location* are the disjoint subsets of the set of *trace-element*.

Trace-links. The relationships between artefacts are called trace-links or trace-relations in the literature. We prefer using the term, *trace-link*, since we use the term, *trace-relation* heavily after section 2.3 to denote a *trace-link* provided with formal semantics. Trace-links may be defined between entire artefacts (e.g., a requirements document and a design document) or between parts of artefacts (e.g. model elements, text fragments or code parts); they can be used for many different purposes, such as impact analysis (i.e., to identify the effect of change in one artefact onto related artefacts), code generation, model transformation, visualisation, change control. The intended use of the trace-links dictates the meaning imposed on the link. 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 amenable to formal analysis [19]. In Figure 1, we specify the notion of *trace-link* as a construct that relates *trace-locations*. The main significant difference about *trace-link* from the literature is that our definition of link can represent not only a binary relation but also an n-ary relation which exists between more than two *trace-location*s. In this way, the notion of *trace-link* turns into more expressive and compact entity. Each instance of a *trace-link* is an ordered list of size n, where n is the total size of the elements each of which points to a *trace-location*.

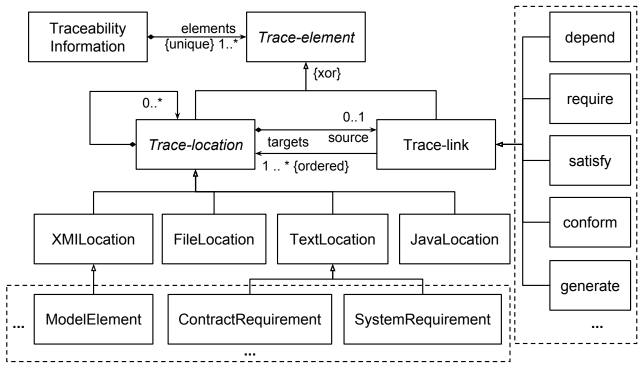


Figure 1: A conceptual model for traceability

The concept of *trace-link* in the domain model solely encapsulates the information necessary to relate *trace-locations*. To assign further meaning to those links, the most common way in literature is to extend the *trace-link* with a predefined semantics. For instance, Goknil et al. [11] include several new link types, that are *requires*, *refines*, *partially refines*, *conflicts* and *contains*, in their requirement metamodel. They additionally provide a well-defined semantics for each of link types by using First-order Logic (FOL). The major drawback of metamodeling approaches is that once an update arises from the changing needs of the project, the metamodel and/or the formalization of *trace-links* are needed to be updated as well, that consequently ends up with reimplementation of the tool.

Trace-locations. There are different traceable elements in complex software intensive systems. These elements need to be specified, specialised and instantiated to represent *traceability information* specific to an organisation or a project. There are some studies focused on requirements traceability [11, 2], traceability between requirements & conceptual models [9], and homogeneous models [20, 6]. However, there are many other artefacts in the process of system development required to be addressed. In this study, to achieve a wider spectrum of artefact types, we introduce the notion of *trace-location* which represents a traceable entity, that is an entire or a part of an artefact.

[2]http://www.omg.org/spec/UML/

A *trace-location* may contain other *trace-location*s in the sense that a requirement specification document is an artefact as a whole in which individual requirements reside. We develop four different type of built-in supported artefacts, namely *TextLocation*, *FileLocation*, *XMILocation* and *JavaLocation*. This can be extended for further needs using the API provided by the platform. A *TextLocation* designates the offset and the length of a text fragment, which usually denotes an important part of a plain-text artefact such as a particular requirement in a specification document. *FileLocation* shows the relative path of the document in the workspace. *XMILocation* is derived to locate the position of a model element in an XML Metadata Interchange (XMI)[3] document, oftentimes created through model-driven engineering approaches to interchange abstract data model of models between MOF-based tools. Finally, *JavaLocation* is introduced to be able to mark a java construct such as *package*, *class*, *method*, *field* and even a *for-each* loop as a *TraceLocation* through parsing the program file and obtaining its abstract-syntax tree (AST).

All these specializations of the *trace-location* capture only the contextual information regarding the location of the element, similar to the case we mentioned in *trace-link*. There is still no semantics assigned to them for traceability analysis yet. On the other hand, one can specialize a location to create a taxonomy or to assign a special meaning on it, e.g. creation of new concepts such as *SystemRequirement* and *ContractRequirement* inherited from *TextLocation* to analyze different part of a specification document individually. However, in a project specific setting of traceability there might be countless number of design choices as depicted inside dashed-line boxes on the figure 1. Drivalos et. al. [6] present a metamodeling approach, called Traceability Metamodeling Language (TML), dedicated to defining traceability models for gaining semantically rich case-specific traceability. Nevertheless, the downside of this approach is that a traceability system created using TML does not facilitate the reconfiguration of the metamodel based on changing needs. To overcome these limitations of similar metamodeling approaches suggested in the literature we propose to externally assign types to *trace-element*s derived from the formal specification and allow them mutate preserving structural integrity. In order to extend a base traceability metamodel without making any modifications on it and to systematically configure the semantics of *trace-link*s, our approach relies on the creation of a *first-order relational model* from the core *Traceability Information*. In the next section, we briefly describe what a first-order model is and why such a conversion is practical to achieve a solid approach for automated analysis of traceability.

2.2 First-order Relational Model

The platform uses the "relational logic" of Alloy [15], essentially consisting of a first-order logic (FOL) augmented with the operators of the relational calculus [24]. The inclusion of *transitive closure* extends the expressiveness beyond standard FOL and allows the encoding of common reachability constraints that otherwise could not be expressed, such as preventing cyclic dependencies between *trace-locations*. The *dot join* and *transpose* operators ensure a uniform way of navigation between *trace-locations* through *trace-links* in constraints. In contrast to specification languages (such as B [1], Z [22], and OCL[4]) that are based on set-theoretic logics, Alloy's relational logic was designed to have a stronger connection to data modeling languages (such as ER [5] and SDM [13]), a more uniform syntax, and a simpler semantics [25]. Alloy also supports *n-ary* relations, and thus, instances of traceability patterns emerging from *n* participating artifacts are naturally modeled as *n-tuples*, therefore, sets of such pattern instances are *n-ary* relations. Considering those facts, instead of proposing a domain-specific language for traceability modeling, first-order relational logic (FORL) is chosen to be used as a basic configuration file of the Tarski platform. However, in order to use Alloy as a means of domain-specific language for traceability modeling, without obscuring the language, we had to introduce some syntactic constructs in the form of annotations for users to provide some extra meta-information that guides the reasoning process. Hence, apart from this specification file, the platform does not need any further information to dynamically adapt itself to a new project setting. The details of the annotation mechanism will be given later in section 2.3.

A *first-order relational model* of a specification, expressed as a collection of declarative constraints written in FORL, is a binding of its free variables to values that makes the specification true or false. The values assigned to variables, and the values of expressions evaluated in the context of a given model, are relations. Sets are unary relations which represents a set of atoms and scalars are singleton unary relations. A relation with no tuples is empty. These relations are first order: that is, they consist of tuples whose elements are atoms (and not themselves relations) [16].

All structures in first-order relational models are built from *atoms* and *relations*, corresponding to the basic entities and the relationships between them. A relational *universe* consists of *atoms*, each *atom* is a primitive construct which is uninterpreted, immutable and indivisible. A *relation* is a structure that relates atoms. It's composed of a set of tuples, each *tuple* being a sequence of atoms. The number of atoms in each tuple of a relation must be the same, that is called the *arity* of the relation. Relations with arity one, two, and three are said to be unary, binary, and ternary. At first glance, encoding of traceability instance can be realized practically as an instance of relational model. Earlier version of the platform in fact used the same approach, but technical challenges in traceability forced us to adopt a domain specific model as shown in section 2.1.

2.3 Type Annotations and Trace-Relations

In this section we essentially describe one of our core contributions, creation of a relational universe and a partial model from traceability information. As we discussed in previous section, in order to assign meaning to *trace-elements*, we need to transform *traceability information* into the formal domain to assign precise semantics. On the other side, a traceability instance has higher-order structures formed due to the technical challenges such that a *trace-location* may compose of other *trace-location*s and *trace-link*s. Therefore,

[3]http://www.omg.org/spec/XMI/

[4]http://www.omg.org/spec/OCL/

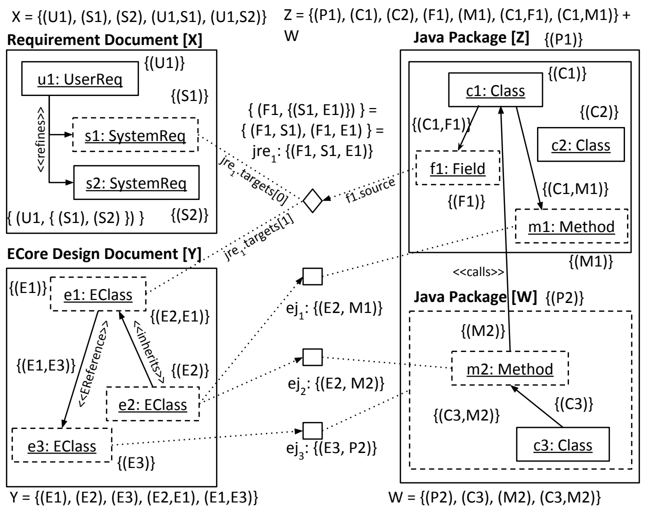


Figure 2: Fragments of a traceability instance

we encode *trace-element*s as first-order (flat) structures so that the *trace-element*s become amenable to manipulation and automated analysis as depicted in figure 2.

A relation specification written in Alloy consists of type declarations (*signature* and *field* definitions) and formulas (*fact*s). In this section we focus on type declarations which is used to annotate *trace-location*s or to form a *trace-link*. Alloy types implicitly represent relations. A *basic type* is introduced for each top level signature and extension signature. The signature *abstract sig A* {} declares a top-level type named A whereas *sig B extends A* {} declares a type B as a subtype of the type associated with A. Since a unary relation represents a set of atoms, this declaration also means that B is a disjoint subset of A. The keyword *abstract* is optional and indicates an *abstract* signature which has no elements except those belonging to its extensions. In this way, basic type declarations constitute a hierarchy that classifies all atoms. Since universe consists of atoms, this hierarchy takes the form of a tree (i.e. *type hierarchy*) where the implicit type *univ* is at its root. A subset signature, such as *sig C in A* {} which introduces a set C that is a subset of A, can be defined. Subset signatures, unlike extension signatures, are not necessarily mutually disjoint. To form a *type lattice*, a signature can be declared as a subset of a union of sets, given *sig C in* $A + B$ {} every element of C belongs to A or to B. Furthermore, A relation whose arity is greater than one can only be declared as *fields* of a signature. A binary *relation type* r, that is *sig A* $\{r : B\}$, is declared as the product of $A \rightarrow B$, *sig A* $\{r' : B \rightarrow C\}$ declares a ternary *relation type* r', the product of $A \rightarrow B \rightarrow C$ and so on. A relation can be constrained by the multiplicity keywords *lone* (at most one), *some* (at least one), *one* (exactly one), and *set* (any number). A declaration $r : A\ m \rightarrow n\ B$ constrains r to associate each element of A with n elements of B, and each element of B with m elements of A where m and n are multiplicity keywords.

As it is seen, even this fragment of Alloy formalism is expressive enough to be used as an object modeling notation (e.g. ECore [23]). Herewith, in Tarski, each *trace-location* which is subject to formal analysis must be annotated with a type from the hierarchy obtained from the *signature* declarations. Once the type annotation is done for a given *trace-location*, the platform introduces an *atom* to the traceability *universe*, and then adds the atom into the set, associated with that type. This means that, in Tarski, each atom is mutable and interpreted. In fact, since the interpretations of atoms are the same across *universe*, that is *trace-location*, it makes no difference for the analysis. However, mutability brings about several challenges in the analysis and therefore we investigate each state of traceability instance in the system independently.

On the other hand, there are two ways to establish a *trace-relation*. In cases, where a *trace-link* already exists and is subject to formal analysis, the platform try to approximate a suitable *relation type* obtained from the *field* declarations. If a type is successfully assigned, the *trace-link* becomes a *trace-relation*, then the platform creates a *tuple* and adds it into the relation associated with the type. For instance, if the *source* end of a *trace-link* has a type which conforms to a *relation type* declaration, a type for the *trace-link* can be determined based on the other ends of the *trace-link*. In the other cases, when a *trace-link* does not exist, user can create a legitimate *trace-relation* directly using the wizards presented to the user in the same order of *relation type* declaration, accordingly the *trace-link* that corresponds to *trace-relation* is automatically created. As a result of these operations, an instance of the relational model is being constructed from the traceability instance using *base* and *field type* declarations.

Figure 2 depicts an arbitrary snapshot of traceability instance created among three types of artefact, that includes some fragments of a requirement document, an ECore model and a Java code file which consists of two packages. Each box shows an entity in its own formalism, a box with dashed outline denotes a *trace-location*. For instance, the element, R_1 represents the serialization of an *EObject* in the document, that is an instantiation of an *EClass* of an *ECore* model. It is associated with an *XMILocation*, detected from the document's context. The dotted lines represents *trace-link*s and their combination shows a *trace-relation*. For each relation, an arrow is put on one of *trace-locations* to emphasize the *source* end. In this figure, it appears that a trace definition is probably declared from design entities *EClass* to Java language constructs *Package, Class, Method* to represent the relationship between the design and implementation in the forward direction (R_{EJ}). On the other side, in the backward direction, a java developer might relate his implementation commit with a system requirement and a design element (R_{JRE}). The *universe* of traceability of the current state is $D_T : \{S_1, E_1, E_2, E_3, F_1, M_1, M_2, P_2\}$, and the relational model under the signature $\Sigma_T : \{R_{EJ} \sqsubseteq E \rightarrow C \sqcup M \sqcup F,\ R_{JRE} \sqsubseteq F \rightarrow S \rightarrow E\}$ is $M_t : \{S = \{\langle S_1 \rangle\}, E = \{\langle E_1 \rangle, \langle E_2 \rangle, \langle E_3 \rangle\}, J = \{\langle F_1 \rangle, \langle M_1 \rangle, \langle M_2 \rangle, \langle P_2 \rangle\}, R_{EJ} = \{ \langle E_2, M_1 \rangle, \langle E_2, M_2 \rangle, \langle E_3, P_2 \rangle\}, R_{JRE} = \{\langle F_1, S_1, E_1 \rangle\}\}$.

2.4 Formal Semantics & Automated Analysis

Tarski platform provides three types of automated analysis. If a model derived from a given universe satisfies the constraints in the specification, it is called a valid model. It is worth emphasizing that we are not interested in checking the satisfiability of a specification, that is finding a valid model for verification purposes. Instead, the user elaborates the model and according to his needs he changes the specification. Subsequently, the system checks whether the model satisfies the specification or not, that is called by the tool as *consistency checking*. If the model is a partial (incomplete) model, the platform tries to complete the model with respect to the semantics declared in the specification inferring new trace-relations on the model, which is what we call as the process of *reasoning about trace-relations*. If a desycnronization occurs on one or more ends of a *trace-link* probably caused by a change such as deletion of a trace-location, we try to repair the broken link based on the semantics and the current snapshot of the system, that is called *trace-location discovery* by us.

The decision procedure of KodKod model finder is integrated instead of using Alloy Analyzer since Alloy has no notion of partial models. If a partial traceability instance is available for a set of Alloy constraints which is usually the case, it can only be provided to the analyzer in the form of additional constraints. Since KodKod, that is the back-end reasoner of Alloy Analyzer, is essentially forced to rediscover the partial model from the constraints, the approach is limited in scale [25]. To overcome this limitation, we use KodKod's API, extracting lower bounds from each snapshot of the trace information of running instance on the platform.

Consistency Checking. As each interaction of the user with the visualization view of the platform mutates tuples in the model and adds new atoms to the universe (Figure 4 Part 3), it is not guaranteed that the model satisfies the FORL *facts*. However, this allows the user to manipulate traces in a free-hand manner while the structural integrity preserved by the type hierarchy. Due to these mutations, a mechanism is needed to check the satisfiability of the changed model [17, 20]. In Tarski platform, the model generated from the user's traceability information is automatically encoded in KodKod using the exact bounds to check the possible inconsistencies and the satisfiability of the given model is reported to the user.

Reasoning about Trace-relations. In the scope of Tarski platform, one type of inference is the activity of deducing new *trace-relation*s. To realize this, user guides the platform, providing annotations in the specification with marking the *fact(s)* which holds during reasoning and indicating the relation on which the reasoning is targeted. The platform synthesizes a new specification in which the unary relations become singletons, undesired relations become empty, and the marked relations are not constrained. This forces the analyzer to find new tuples on the model if exists, which is reported back to the user as inferred relations.

Trace Elements Discovery. In addition to the reasoning about relations, the proposed framework can reason on the provided trace-elements to suggest missing trace-locations. In this case, the n-ary relations are fixed and the unary relations are not constrained to generate tuples on the model.

When the user accepts a suggested unary or n-ary relation on the visualization view of Tarski platform, the relation will be encoded as a new constraint on the model and further reasoning will be done on this new model. This mechanism will provide an evolutionary approach to elaborate the traceability information. To specify the semantics of the traceability, we have provided an enhanced text editor, see Figure 4 part 1, with which the user defines the system rules using relational logic.

3. PLATFORM ARCHITECTURE

The platform architecture of the proposed approach is depicted in Figure 3. Considering the functionality of the system, the first row of figure 3 shows the configuration process. In the second row, the traceability processes are realized including the user-interaction, persistence, and interpretation of trace information. Finally, in the third row, the reasoning mechanism is maintained over the provided model and AST of specification. The figure shows the interaction of the Tarski platform with the Traceability Framework and Alloy back-end. From the user point of view, the formal specification is loaded or updated into the system, then user carries out interactions with the system such as creating *trace elements* and assigning types to them. If the extra information is provided by the user in the specification using special annotations, the Tarski platform is able to determine the types of trace elements and automatically creates trace-links. User can analyze the traceability based on his/her provided locations and links, and the specified rules. The Tarski platform interacts with Traceability Framework to persist trace instance. Also, Tarski interacts with Alloy compiler to reuse the type system [7] generated from the user specification. While assigning relation names to trace-links or creating relations between trace-locations, the system suggests only the legitimate trace-locations based on semantic analysis of the specification. As a result, the traceability information is adapted to a first-order relational model by the user's type assignments to *trace-location*s (unary relations) and *trace-links* (binary. ternary and n-ary relations). Each user function has a counter-part API method in order to create automatically those trace-elements especially in model-based development, e.g trace creation while generating code from a domain-specific model. Furthermore, Tarski platform provides functions such as *create*, *delete*, *update* and *change type* of the relations with respect to type hierarchy and multiplicity constraints to enable users to elaborate further on the formal instance.

Tarski is being exploited and extended in different industrial use case scenarios primarily as part of two projects, namely ModelWriter[5], and ASSUME[6]which are labeled by the European Union's EUREKA Cluster programme ITEA (Information Technology for European Advancement). Additional details about the platform including data set, source codes and screen-casts are available on the project repository[7].

4. INDUSTRIAL USE CASE

The approach introduced in this paper has been applied on the Application Lifecycle Management platform of HAVEL-

[5]https://itea3.org/project/modelwriter.html
[6]https://itea3.org/project/assume.html
[7]https://github.com/ModelWriter/WP3

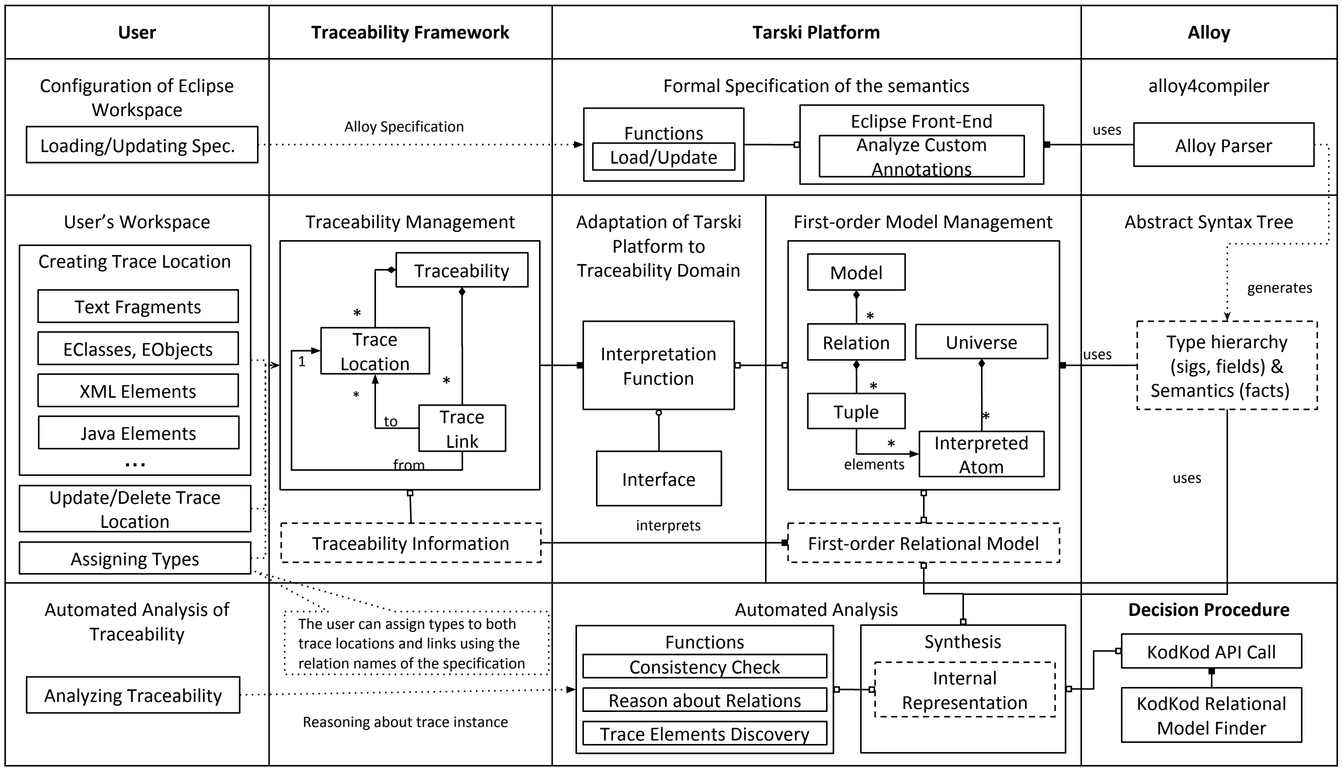


Figure 3: Overview of the approach and the platform

SAN[8], one of the largest systems and software company in defense industry of Turkey, to demonstrate the usability of the approach and its supporting platform. Basically, tracing for a typical project forms a complex directed acyclic graph. A contract requirement should be traceable forward into the system requirements that have been elaborated from it, and on into the code modules that implement it, or the test cases verify that code and even a given section on the user manual which describes the actual functionality [3]. Similarly, a contract-based project in HAVELSAN consists of a set of *Artifacts* which can be partitioned into *Specification*, *Implementation* and *Verification*. An artefact actually is an abstraction in this particular use case to denote an element which is traceable in the user's workspace such as a text fragment, model element, java method or class etc. A requirement *Specification* is formed of different *ContractRequirements* which probably *contains* different requirements to create a part-whole hierarchy. Each *SystemRequirement* might be decomposed into different system requirements through *refines* relation. Each system requirement is satisfied by one or more implementation methods such as *Model*, *Component*, and *Code* that are verified by *Simulation*, *Analysis* or *Test* method. An *Implementation fulfills* a *ContractRequirement*. The classification for type annotations are given in the following:

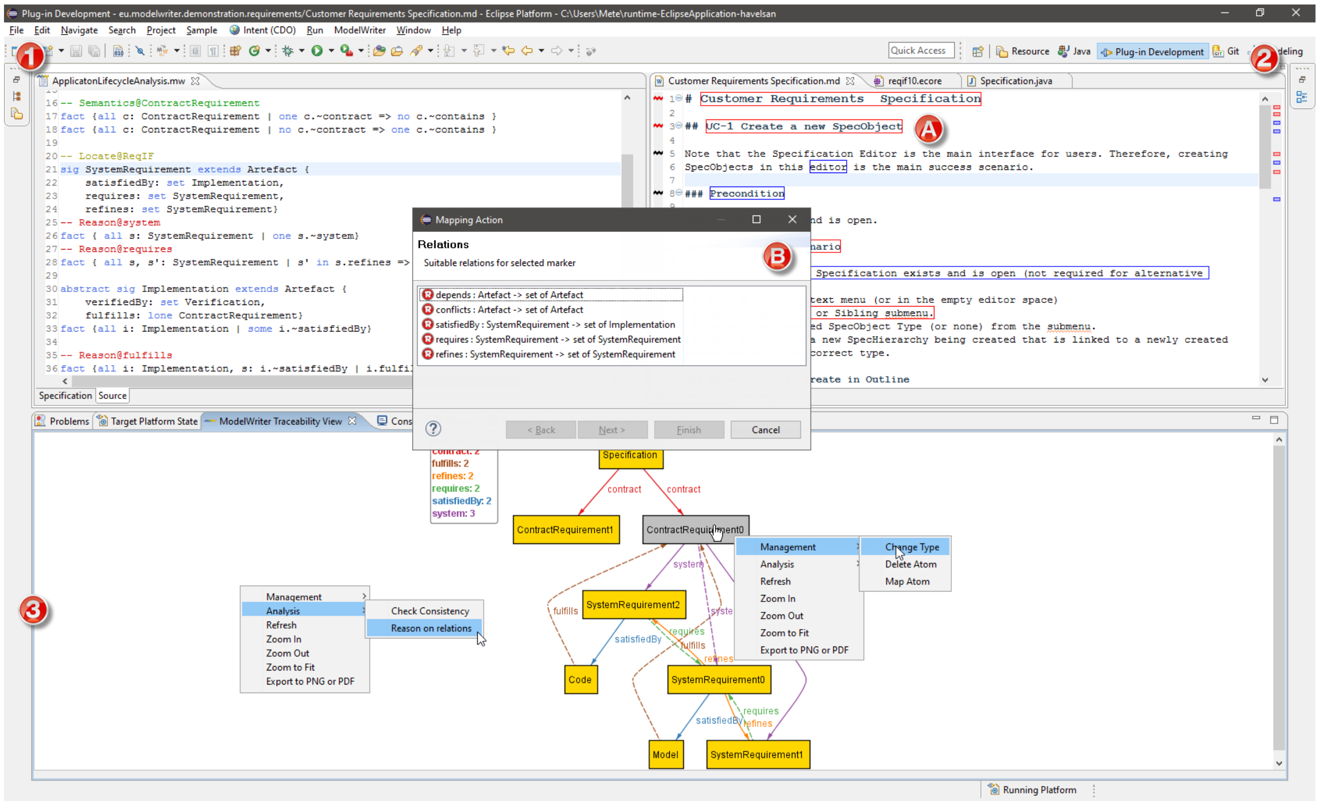
```
abstract sig Artefact {
    depends: set Artefact}

-- Locate@File
one sig Specification extends Artefact {
    contract: some ContractRequirement}

-- Locate@Text
sig ContractRequirement extends Artefact {
    system: set SystemRequirement,
    contains: set ContractRequirement}

-- Locate@ReqIF
sig SystemRequirement extends Artefact {
    satisfiedBy: set Implementation,
    requires: set SystemRequirement,
    refines: set SystemRequirement}

abstract sig Implementation extends Artefact {
    fulfills: lone ContractRequirement}

-- Locate@Java
sig Code, Component extends Implementation {}

-- Locate@EMF
sig Model extends Implementation {
    transforms, conforms: set Model,
    generates: set (Code ∪ Component)}
```

An example of the formalization is provided below for *contains* relation that occurs between *ContractRequirements*:

```
-- Semantics@ContractRequirement.contains
fact {∀ c: ContractRequirement |
    one c.~contract ⟹ no c.~contains }
fact {∀ c: ContractRequirement |
    no c.~contract ⟹ one c.~contains }
```

Several facts are intentionally annotated by the user to guide

[8]http://www.havelsan.com.tr/eng



```
-- Semantics@ContractRequirement
fact {all c: ContractRequirement | one c.~contract => no c.~contains }
fact {all c: ContractRequirement | no c.~contract => one c.~contains }

-- Locate@ReqIF
sig SystemRequirement extends Artefact {
    satisfiedBy: set Implementation,
    requires: set SystemRequirement,
    refines: set SystemRequirement}
-- Reason@system
fact { all s: SystemRequirement | one s.~system}
-- Reason@requires
fact { all s, s': SystemRequirement | s' in s.refines =>

abstract sig Implementation extends Artefact {
    verifiedBy: set Verification,
    fulfills: lone ContractRequirement}
fact {all i: Implementation | some i.~satisfiedBy}

-- Reason@fulfills
fact {all i: Implementation, s: i.~satisfiedBy | i.fulfi
```

```
# Customer Requirements Specification

## UC-1 Create a new SpecObject

Note that the Specification Editor is the main interface for users. Therefore, creating
SpecObjects in this editor is the main success scenario.

### Precondition

nd is open.

nario

 Specification exists and is open (not required for alternative

text menu (or in the empty editor space)
 or Sibling submenu.
ed SpecObject Type (or none) from the submenu.
a new SpecHierarchy being created that is linked to a newly created
correct type.

reate in Outline
```

Mapping Action

Relations

Suitable relations for selected marker

- depends : Artefact -> set of Artefact
- conflicts : Artefact -> set of Artefact
- satisfiedBy : SystemRequirement -> set of Implementation
- requires : SystemRequirement -> set of SystemRequirement
- refines : SystemRequirement -> set of SystemRequirement

< Back | Next > | Finish | Cancel

Figure 4: Screenshot of the running platform configured with HAVELSAN's configuration file.

the reasoning process. *Reason*@ keyword indicates that the fact following the annotation will be excluded from the consistency check, but included in the reasoning to infer the given relations in the annotation following the @ sign. Several examples from the specification are listed in the followings. In this way, backward traceability is automatically constructed based on the semantics defined for each relation.

```
-- Reason@ContractRequirement.system
fact {∀ s:SystemRequirement, s': s.*~refines |
    s'.~system = s.~system}

-- Reason@SystemRequirement.requires
fact { ∀ s, s': SystemRequirement |
    s' in s.refines ⟹ s in s'.requires }

-- Reason@Implementation.fulfills
fact {∀ i: Implementation, s: i.~satisfiedBy
    | i.fulfills = s.~system }
```

In Figure 4 part 3, the inferred relations are shown on the graph with dashed lines. The complete specification of this use case is available online[9].

5. RELATED WORK

There are several studies in the literature and some industrial technologies (some of which became defacto standards) dealing with traceability and its automated analysis. Regarding the former group in Paige et. al. [19], only model elements are considered to be linked with semantically rich traces. These traces are typed and conform to a case-specific trace metamodel accompanied by a set of case-specific constraints, which cannot be captured by the metamodel. As another study, Goknil et al. [11] proposed a requirements metamodel including relation types with formal semantics expressed in FOL. The formalization of relations was used in a tool called TRIC to support for inference and consistency checking. Later in [9], the work was extended to generate and validate traces between requirements and software architectures. Similarly, Drivalos et. al. [6] uses Traceability Metamodeling Language to define traceability metamodels. Sebatzadeh et. al. [20] use formal specification for traceability of homogeneous models. All in all, unlike Tarski, none of the above discussed studies support automated analysis including dynamic configuration of semantics. Also, in Tarski, arbitrary model and text elements are considered to be linked with formal semantics. Regarding the industrial tools and technologies on traceability, modeling tools such as EMF [23] and SysML [21], requirement interchange standard (ReqIF [12]) and management tools such as RMF[10] and IBM Rational DOORS [14] provide some automated or manual means to specify and manage traceability. However, none of them provides configuration with a formal specification and rich semantic support for the artefacts especially on a heterogeneous development and design environment. In addition, there is a lack of a systematic approach to support automated analysis of traceability in such environments.

[9]https://github.com/ModelWriter/WP3/wiki/ALM

[10]https://www.eclipse.org/rmf/

6. CONCLUSION AND FUTURE WORK

In this paper a platform called Tarski has been introduced for automated analysis of dynamically configurable traceability semantics. Formal semantics of traceability can be specified interactively based on project-specific needs using first-order relational logic to exploit the automated analysis support of the platform. Traceability between arbitrary artefacts such as model elements and text fragments is also maintained. The use of the approach and the corresponding platform are described within the context of an industrial application lifecycle management process.

Currently we can't analyze the temporal properties of traceability. Thus, we omit ordered relation declarations in Alloy formalism, which are used to model time or state components of a system. To deal with large scale models, as a future work, we will identify essential fragments of first-order relational theory which is expressive enough for traceability to adapt SMT solvers [8] as back-end reasoners to Tarski platform.

7. ACKNOWLEDGMENTS

This work is partially supported by the Scientific and Technological Research Council of Turkey (TUBITAK), Technology and Innovation Funding Programs Directorate (TEYDEB) under project# 9140014, 9150181 and Minister for the Economy, Industry and Digital Affairs of France, Directorate-General for Enterprise (DGE) under contract# 142930204. The authors would like to acknowledge networking support by European Cooperation in Science and Technology (COST) Action IC1404 "Multi-Paradigm Modelling for Cyber-Physical Systems".

8. REFERENCES

[1] J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, 2005.

[2] N. Aizenbud-Reshef, R. F. Paige, J. Rubin, Y. Shaham-Gafni, and D. S. Kolovos. Operational semantics for traceability. In *ECMDA Traceability Workshop (ECMDA-TW)*, pages 8–14, 2005.

[3] P. Bourque and R. E. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK) Version 3.0.* IEEE Computer Society Press, 3rd edition, 2014.

[4] D. Chappell. *What is Application Lifecycle Management?* Chappell & Associates, 2008.

[5] P. P.-S. Chen. The Entity-relationship Model; toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, Mar. 1976.

[6] N. Drivalos, D. S. Kolovos, R. F. Paige, and K. J. Fernandes. *Engineering a DSL for Software Traceability*, pages 151–167. Springer Berlin Heidelberg, 2009.

[7] J. Edwards, D. Jackson, and E. Torlak. A type system for object models. *SIGSOFT Softw. Eng. Notes*, 29(6):189–199, Oct. 2004.

[8] A. A. El Ghazi and M. Taghdiri. Relational reasoning via SMT solving. In *International Symposium on Formal Methods*, pages 133–148. Springer, 2011.

[9] A. Goknil, I. Kurtev, and K. V. D. Berg. Generation and validation of traces between requirements and architecture based on formal trace semantics. *Journal of Systems and Software*, 88:112 – 137, 2014.

[10] A. Goknil, I. Kurtev, K. van den Berg, and W. Spijkerman. Change impact analysis for requirements: A metamodeling approach. *Information and Software Technology*, 56(8):950 – 972, 2014.

[11] A. Goknil, I. Kurtev, K. van den Berg, and J.-W. Veldhuis. Semantics of trace relations in requirements models for consistency checking and inferencing. *Softw. Syst. Model.*, 10(1):31–54, 2011.

[12] A. Graf, N. Sasidharan, and Ö. Gürsoy. *Requirements, Traceability and DSLs in Eclipse with the Requirements Interchange Format (ReqIF)*, pages 187–199. Springer Berlin Heidelberg, 2012.

[13] M. Hammer and D. McLeod. The Semantic Data Model: A modelling mechanism for data base applications. In *ACM SIGMOD International Conference on Management of Data*, pages 26–36, New York, NY, USA, 1978.

[14] IBM. Rational DOORS: A requirements management tool for systems and advanced IT applications, 2011.

[15] D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transaction Software Engineering Methodology*, 11(2):256–290, Apr. 2002.

[16] D. Jackson. *Software Abstractions: logic, language, and analysis.* MIT press, 2012.

[17] D. Kolovos, R. Paige, and F. Polack. Detecting and repairing inconsistencies across heterogeneous models. In *1st International Conference on Software Testing, Verification, and Validation*, pages 356–364, 2008.

[18] S. Nair, J. L. de la Vara, and S. Sen. A review of traceability research at the requirements engineering conferencere. In *21st IEEE International Requirements Engineering Conference*, pages 222–229, July 2013.

[19] R. F. Paige, N. Drivalos, D. S. Kolovos, K. J. Fernandes, C. Power, G. K. Olsen, and S. Zschaler. Rigorous identification and encoding of trace-links in model-driven engineering. *Softw. Syst. Model.*, 10(4):469–487, 2011.

[20] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook, and M. Chechik. Consistency checking of conceptual models via model merging. In *International Requirements Engineering Conf.*, pages 221–230, 2007.

[21] M. Soares and J. Vrancken. Model-driven user requirements specification using SysML. *Journal of Software*, 3(6):57–68, 2008.

[22] J. M. Spivey. *The Z notation: a reference manual.* Prentice-Hall, New York, NY,, 1992.

[23] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework.* Pearson Education, 2008.

[24] A. Tarski. On the calculus of relations. *The Journal of Symbolic Logic*, 6(03):73–89, 1941.

[25] E. Torlak. *A constraint solver for software engineering: finding models and cores of large relational specifications.* PhD thesis, Massachusetts Institute of Technology, 2008.

[26] S. Winkler and J. Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Softw. Syst. Model.*, 9(4):529–565, 2010.