A Tool for Automated Reasoning about Traces Based on Configurable Formal Semantics

Ferhat Erata
Wageningen University, the Netherlands
ferhat@computer.org

Arda Goknil
University of Luxembourg, Luxembourg
arda.goknil@uni.lu

Bedir Tekinerdogan
Wageningen University, the Netherlands
bedir.tekinerdogan@wur.nl

Geylani Kardas
Ege University, Turkey
geylani.kardas@ege.edu.tr

ABSTRACT

We present Tarski, a tool for specifying configurable trace semantics to facilitate automated reasoning about traces. Software development projects require that various types of traces be modeled between and within development artifacts. For any given artifact (e.g., requirements, architecture models and source code), Tarski allows the user to specify new trace types and their configurable semantics, while, using the semantics, it automatically infers new traces based on existing traces provided by the user, and checks the consistency of traces. It has been evaluated on three industrial case studies in the automotive domain (https://modelwriter.github.io/Tarski/).

CCS CONCEPTS

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

KEYWORDS

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

ACM Reference format:
Ferhat Erata, Arda Goknil, Bedir Tekinerdogan, and Geylani Kardas. 2017. A Tool for Automated Reasoning about Traces Based on Configurable Formal Semantics. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 5 pages.
https://doi.org/10.1145/3106237.3122825

1 INTRODUCTION

The complexity of software systems in safety critical domains (e.g. avionics and automotive) has significantly increased over the years. Development of such systems requires various phases which result in several artifacts (e.g., requirements documents, architecture models and test cases). In this context, traceability [29, 32] not only establishes and maintains consistency between these artifacts but also helps guarantee that each requirement is fulfilled by the source code and test cases properly cover all requirements, a very important objective in safety critical systems and the standards they need to comply with [25, 30]. As a result, the engineers have to establish and maintain several types of traces, having different semantics, between and within various development artifacts.

We present a tool, Tarski[1], which supports specifying configurable trace semantics to facilitate multiple forms of automated trace reasoning. Tarski is developed for environments, requiring maintenance of various artifacts, within the context of our research [13, 14] in collaboration with Ford-Otosan [15], Airbus [1] and Havelsan [24]. The motivation behind Tarski is to provide a way to interactively specify trace types and semantics, which vary for different artifacts, to be used in automated trace reasoning.

There are approaches and tools [2, 16, 17, 21, 22] that use a predetermined set of possible trace types and their semantics for automated reasoning. However, in the case of dealing with complex software systems, instead of a one-size-fits-all approach, it is required to enable the adoption of several trace types and their semantics, and herewith the various forms of automated reasoning about traces. To do so, Tarski provides the following features: (i) specifying trace semantics which can be configured due to project and artifact types, (ii) deducing new traces based on the given trace semantics and on the traces which the engineer has already specified, and (iii) identifying the traces whose existence causes a contradiction according to the trace semantics. The tool provides a traceability domain model which describes the basic concepts of traceability such as *Trace-link* and *Trace-location.* The notion of trace-location refers to traceable elements in an artifact, while the notion of trace-link refers to traces between trace-locations. The user defines new trace types by extending *Trace-link* and *Trace-location.* The user specifies the semantics of new trace types in a restricted form of Alloy [26], i.e., First-Order Logic (FOL) augmented with the operators of the relational calculus [33]. We employ Kodkod [35, 36], an efficient SAT-based constraint solver for FOL with relational algebra and partial models, for automated trace reasoning using the trace semantics. Our tool is integrated with Eclipse [8] platform.

In the remaining sections, we outline Tarski's features and components. We highlight the findings from our evaluation of Tarski over multiple industrial case studies with one of our industrial partners.

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3122825

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

2 RELATED WORK

Several approaches and tools have been proposed for automated trace reasoning using the trace semantics [2, 6, 7, 9–11, 16, 18–22, 27, 28, 31, 34]. These approaches employ a predefined set of trace types and their corresponding semantics. For instance, Goknil et al. [22] provide a tool for inferencing and consistency checking of traces between requirements using a set of trace types (e.g., *refines*, *requires*, and *contains*) and their formal semantics. Similarly, Egyed and Grünbacher [11] propose a trace generation approach. They do not allow the user to introduce new trace types and their semantics for automated reasoning. In the development of complex systems, it is required to enable the adoption of various trace types, and herewith automated reasoning using their semantics.

Tarski does not encode any predefined trace type or semantics. It allows the user to interactively define new trace types with their semantics to be used in automated reasoning about traces. Using the semantics specified by the user, Tarski deduces new traces and checks the consistency of traces.

3 TOOL OVERVIEW

Tarski is the tool supporting our approach for automated reasoning about traces based on configurable trace semantics, recently described in [12]. Fig. 1 presents an overview of our tool. In Step 1, the user specifies new trace types and their semantics in First-Order Logic (FOL) augmented with the operators of the relational calculus [33]. To do so, Tarski employs a restricted form of Alloy [26] with a custom text editor. New trace types are defined by extending *Trace-link* and *Trace-location* in *Traceability Domain Model*.

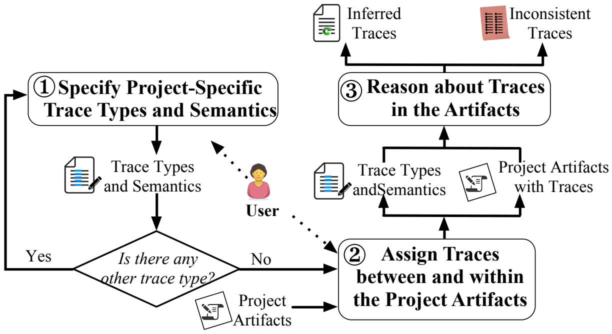


Figure 1: Tool Overview

Once the user specifies the trace types and their semantics, Tarski allows the user to assign traces between and within the input project artifacts (e.g., requirements specifications, architecture models, and test cases) using the trace types (Step 2). After the traces are manually assigned, the tool proceeds to Step 3 with automated trace reasoning. In the rest of the section, we elaborate each step in Fig. 1 using the Electronically Controlled Air Suspension (ECAS) System of Ford-Otosan [15], a safety-critical system in automotive domain, as a case study.

3.1 Specification of Trace Types and Semantics

As the first step, for the artifacts, the user specifies trace types and their semantics in FOL using a restricted form of Alloy. First, the user extends the *traceability domain model* with new trace and artifact types. Fig. 2 shows part of the extended traceability domain model for the ECAS case study.

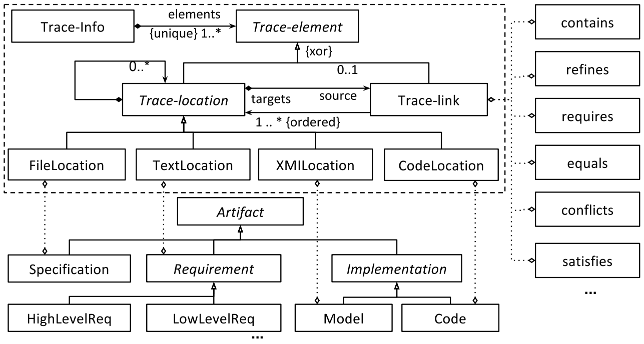
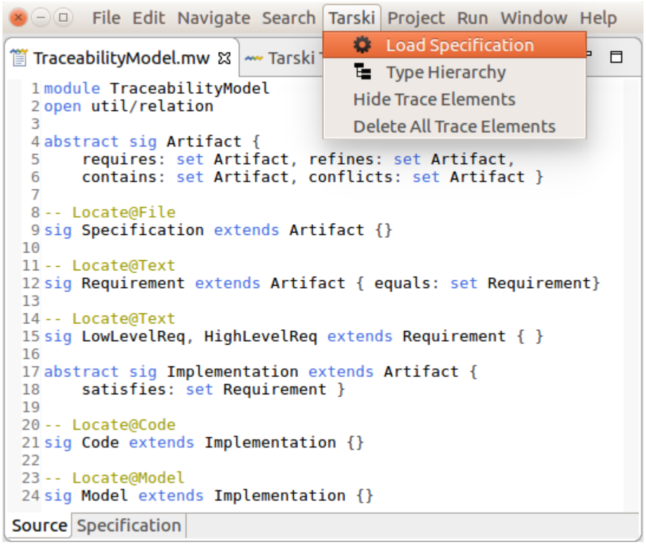


Figure 2: Traceability Model with User-defined Trace Types

We extend *Trace-link* in Fig. 2 with new trace types (e.g., *contains*, *refines*, and *satisfies*), while *Text-location* is extended with new types of elements of the artifacts to be traced in the case study (e.g., *Requirement*, *HighLevelReq*, and *Code*). Fig. 3 shows some of the extensions of *Trace-link* and *Trace-location* in Fig. 2.

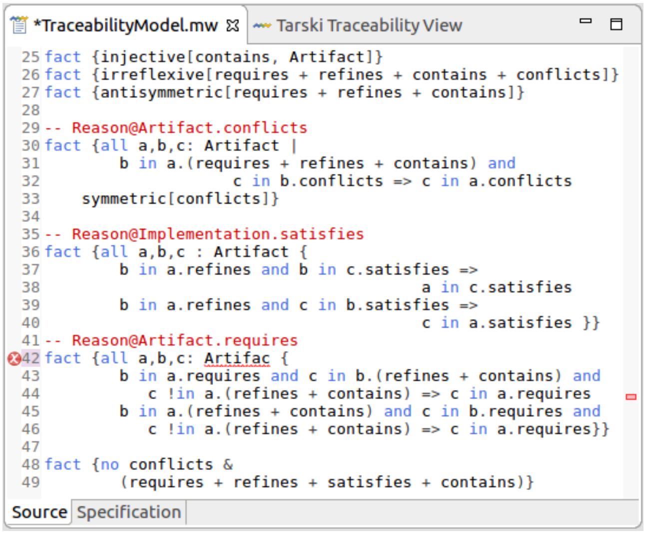
In the following, we briefly explain the restricted Alloy notation Tarski employs for declaring trace types and their semantics. Signatures define the vocabulary of a model in Alloy (see keyword *sig* in Fig. 3). We use them to extend *Trace-location* for declaring artifact element types (see Lines 4, 9, 12, 15, 17, 21 and 24 in Fig. 3). Tarski employs some special annotations to specify artifacts' location types (Lines 8, 11, 14, 20 and 23). The location type information is later used by Tarski to create the Eclipse workspace fields to save traces assigned in Step 2 in Fig. 1 (see Section 3.2). For instance, *Requirement* is given as a text location in Line 11 (see *Requirement* and *Text-location* in Fig. 2), while *Code* is given as a source code location in Line 20. For a trace between a textual requirement and a code fragment, using the location information in Fig. 3, Tarski creates a *resource* field as a path referring to the location of the requirement, while the *resource*, *offset*, and *length* fields are created to refer to the code fragment where *resource* gives the path of the source code file, *offset* gives the start index of the code fragment in the code file, and *length* gives the length of the code fragment.



```
 1 module TraceabilityModel
 2 open util/relation
 3
 4 abstract sig Artifact {
 5     requires: set Artifact, refines: set Artifact,
 6     contains: set Artifact, conflicts: set Artifact }
 7
 8 -- Locate@File
 9 sig Specification extends Artifact {}
10
11 -- Locate@Text
12 sig Requirement extends Artifact { equals: set Requirement}
13
14 -- Locate@Text
15 sig LowLevelReq, HighLevelReq extends Requirement { }
16
17 abstract sig Implementation extends Artifact {
18     satisfies: set Requirement }
19
20 -- Locate@Code
21 sig Code extends Implementation {}
22
23 -- Locate@Model
24 sig Model extends Implementation {}
```

Figure 3: Some Example Trace Types in Tarski

New trace types are defined as binary relations in the signature fields (see Lines 5, 6, 12, and 18 in Fig. 3). Tarski automatically extends *Trace-link* for those binary relations (see Fig. 2). For instance, in Line 18, *Satisfies* is declared as a new trace type between *Implementation* and *Requirement*. Trace semantics is given as *facts* in Alloy (see Fig. 4). Facts are constraints that are assumed to always hold. They are used as axioms in constructing examples and counterexamples [26]. The *Refines*, *Requires* and *Contains* trace types are defined *irreflexive* and *antisymmetric* (see Lines 26 and 27 in Fig. 4). In addition, *Contains* is *injective* (Line 25).



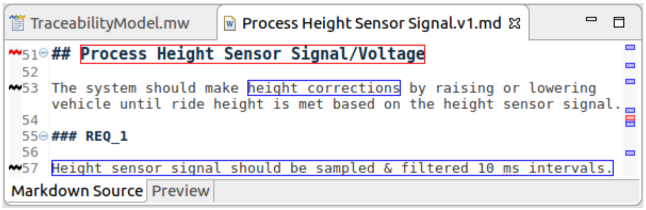
```
25 fact {injective[contains, Artifact]}
26 fact {irreflexive[requires + refines + contains + conflicts]}
27 fact {antisymmetric[requires + refines + contains]}
28
29 -- Reason@Artifact.conflicts
30 fact {all a,b,c: Artifact |
31         b in a.(requires + refines + contains) and
32                     c in b.conflicts => c in a.conflicts
33     symmetric[conflicts]}
34
35 -- Reason@Implementation.satisfies
36 fact {all a,b,c : Artifact {
37         b in a.refines and b in c.satisfies =>
38                                 a in c.satisfies
39         b in a.refines and c in b.satisfies =>
40                                 c in a.satisfies }}
41 -- Reason@Artifact.requires
42 fact {all a,b,c: Artifac {
43         b in a.requires and c in b.(refines + contains) and
44            c !in a.(refines + contains) => c in a.requires
45         b in a.(refines + contains) and c in b.requires and
46            c !in a.(refines + contains) => c in a.requires}}
47
48 fact {no conflicts &
49         (requires + refines + satisfies + contains)}
```

Figure 4: Example Trace Semantics in Tarski

As part of the semantics, we define how trace types are related to each other (Lines 30-49). For instance, according to the fact in Lines 30-33 where *a*, *b* and *c* are artifact elements, if *a refines*, *requires* or *contains b*, while *b conflicts* with *c*, then *a* also *conflicts* with *c*.

3.2 Trace Assignment in Project Artifacts

Tarski guides the user in assigning traces between and within the input artifacts (see Step 2 in Fig. 1). The user manually assigns traces for the input artifacts using the trace types. The main challenge is that the traceable parts of textual artifacts (e.g., requirements in a requirements specification) need to be determined before assigning traces. To address this challenge, Tarski employs a semantic parsing approach [23] that automatically maps natural language to Description Logic (DL) axioms. The mappings between the natural text and the DL axioms are used by Tarski to automatically identify the traceable parts of textual artifacts. Fig. 5 shows part of the ECAS requirements specification after semantic parsing in Tarski.



```
## Process Height Sensor Signal/Voltage

The system should make height corrections by raising or lowering
vehicle until ride height is met based on the height sensor signal.

### REQ_1

Height sensor signal should be sampled & filtered 10 ms intervals.
```

Figure 5: Part of the ECAS Requirements Specification

The blue colour indicates the traceable parts of the ECAS requirements specification which do not yet have any trace. When the user wants to assign a trace from/to these blue coloured parts, Tarski automatically suggests the possible trace types using the type hierarchy encoded in Step 1 (see Fig. 3). After the trace is assigned, the blue colour automatically becomes red, which indicates having at least one trace.

3.3 Automated Reasoning about Traces

Inferencing and consistency checking aim at deriving new traces based on given traces and determining contradictions among traces. These two activities enrich the set of traces in the artifacts. They are processed in parallel because the consistency checking uses the machinery for inferencing and also checks the inconsistencies among inferred traces as well as among given traces.

Table 1: Some Requirements and Code Fragments in ECAS

Nr.	Requirements/Code Fragments
r_{11}	The system shall do height corrections using long and short term filtered height sensor signal.
r_{59}	The system shall always use height sensors in the range of 0-5V to avoid long term signal filtering.
r_{60}	The system shall do height corrections using long and short term filtered height sensor signal with 10ms interval.
r_{97}	The system shall filter height sensor signal in short term and long term for height corrections.
r_{98}	The system shall filter height sensor signal in long term for height corrections.
i_{14}	`vehicle::ecas::processHeightSensor::filterSignal`
i_{72}	`vehicle::ecas::processHeightSensor`

3.3.1 *Inferring New Traces.*

Tarski takes the artifacts and their manually assigned traces as input, and automatically deduces, using the user-defined trace types and their semantics, new traces as output. Fig. 6 gives the assigned and inferred traces for some simplified ECAS requirements and source code fragments in Table 1.

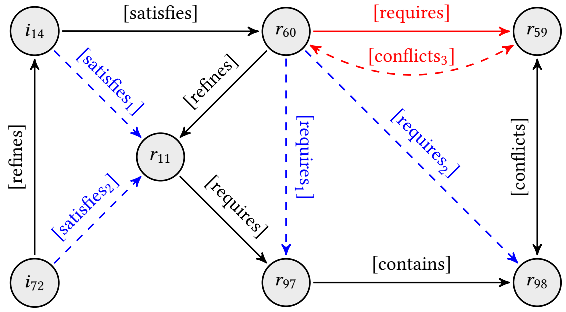


Figure 6: Example Inferred and Inconsistent Traces in ECAS

The solid arrows represent the manually assigned traces, while the dashed arrows are the traces automatically inferred by Tarski. For instance, the user assigns the *refines* traces between i_{72} and i_{14}, and between r_{60} and r_{11}. Using the trace semantics in Fig. 4, Tarski automatically infers two *satisfies* traces, two *requires* traces and one *conflicts* trace in Fig. 6. For instance, i_{14} *satisfies* r_{11} (i.e., inferred) because it *satisfies* r_{60} which *refines* r_{11} (see the *fact* in Lines 36-40 in Fig. 4). The *conflicts* trace between r_{60} and r_{59} is inferred because r_{60} *requires* r_{98} which *conflicts* with r_{59} (see the *fact* in Lines 30-33 in Fig. 4). Please note that the *requires* trace between r_{60} and r_{98} is inferred.

3.3.2 Checking Consistency of Traces. Tarski takes the artifacts and their given and inferred traces as input, and automatically determines, using the user-defined trace types and their semantics, the inconsistent traces as output. Tarski provides an explanation of inconsistent traces by giving all the manually assigned traces causing the inconsistency. In Fig. 6, the *requires* and *conflicts* traces between r_{60} and r_{59} are inconsistent (or contradict each other) because a requirement cannot require another requirement which it conflicts with (see the *fact* in Lines 48-49 in Fig. 4). The inconsistent *conflicts* trace is inferred using two other inferred traces. First, r_{60} *requires* r_{97} (i.e., inferred) because r_{60} *refines* r_{11} which *requires* r_{97}. Second, r_{60} *requires* r_{98} (i.e., inferred) because r_{60} *requires* r_{97} which contains r_{98}. And lastly, r_{60} *conflicts* with r_{59} (i.e., inferred and inconsistent with the *requires* trace) because r_{60} *requires* r_{98} which *conflicts* with r_{59}. Therefore, the manually assigned *refines* trace between r_{60} and r_{11}, *requires* trace between r_{11} and r_{97}, *contains* trace between r_{97} and r_{98}, *conflicts* trace between r_{98} and r_{59}, and *requires* trace between r_{60} and r_{59} actually cause the inconsistency in Fig. 6. When we, together with the Ford-Otosan engineers, analyzed all these assigned traces, we identified that the manually assigned *requires* trace between r_{60} and r_{59} is invalid. We removed it to resolve the inconsistency.

4 EVALUATION

Our goal was to assess, in an industrial context, the feasibility of using Tarski to facilitate automated trace reasoning using user-defined trace types and semantics. For this assessment, we selected three industrial case studies which are subsystems of the ECAS system developed by different teams at Ford-Otosan [15]. They are relatively mid-sized systems with multiple artifacts (e.g., requirement specifications, SysML models, Simulink models, test suites and C code) requiring various trace types (see Table 2).

Before conducting the case studies, the Ford-Otosan engineers were given presentations illustrating the Tarski steps and a tool demo. The engineers held various roles (e.g., senior software engineer and system engineer) and all had substantial experience in software development. For each case study, we asked the engineers to identify trace types and assisted them in specifying trace types and their semantics in Tarski (the 1st and 2nd columns in Table 2). The artifacts in each case study had already some typeless traces (i.e., *trace to/from*) manually assigned by the engineers. We asked them to reassign those traces using the trace types they specified using Tarski (the 3rd and 4th columns).

Table 2: Number of Trace Types, Facts, Assigned & Inferred Traces, and Inconsistent Parts in the Case Studies

	Trace Types	Facts	Traced Elements	Manual Traces	Inferred Traces	Inconsis. Parts
#1	7	11	125	138	502	3
#2	11	20	47	102	145	5
#3	10	14	16	21	53	1

To evaluate the output of Tarski, we had semi-structured interviews with the engineers. All the inferred traces and the found inconsistencies in the case studies were confirmed by the engineers to be correct (the 5th and 6th columns). The engineers considered the automated generation of new traces and the consistency checking of traces to be highly valuable. The restricted Alloy Tarski employs was sufficient to specify all the trace types and their semantics for the case studies. The engineers agreed about the useful guidance provided by Tarski for specifying trace types and semantics. They stated that it was intuitive to specify trace types and semantics using Tarski although more practice and training were still needed to become familiar with the tool.

5 IMPLEMENTATION & AVAILABILITY

Tarski has been implemented as an Eclipse plug-in. This plug-in activates the user interfaces of Tarski and provides the features *specifying trace types and their semantics*, *assigning traces in the artifacts using user-defined trace types*, and *reasoning about traces* (i.e., *deducing new traces* and *checking consistency of traces*). We use Kodkod [35, 36], an efficient SAT-based finite model finder for relational logic, to perform automated trace reasoning using the user-defined semantics. Trace types and their semantics are specified in the restricted form of Alloy, while the artifacts containing manually assigned traces are automatically transformed into Alloy specifications. Using the trace semantics and the artifacts in Alloy, we directly call KodKod API [5] to reason about traces.

Tarski relies upon (i) a customized Eclipse editor to specify trace types and their semantics in FOL, (ii) another customized Eclipse editor to assign traces between and within the artifacts (including textual artifacts such as requirements specifications) using user-defined trace types, and (iii) *alloy4graph* [3] and *alloy4viz* [4], the Alloy API packages for performing graph layout and displaying Alloy instances, to visualize the output of automated trace reasoning.

Tarski is approximately 50K lines of code, excluding comments and third-party libraries. Additional details about Tarski, including executable files and a screencast covering motivations, are available on the tool's website at:

https://modelwriter.github.io/Tarski/

6 CONCLUSION

We presented a tool, Tarski, to allow the user to specify configurable trace semantics for various forms of automated trace reasoning such as inferencing and consistency checking. The key characteristics of our tool are (1) allowing the user to define new trace types and their semantics which can be later configured, (2) deducing new traces based on the traces which the user has already specified, and (3) identifying traces whose existence causes a contradiction. Tarski has been evaluated over three industrial case studies. The evaluation shows that our tool is practical and beneficial in industrial settings to specify trace semantics for automated trace reasoning. We plan to conduct more case studies to better evaluate the practical utility and usability of the tool.

ACKNOWLEDGMENTS

This work is conducted within ModelWriter[13] and ASSUME[14] projects and partially supported by the Scientific and Technological Research Council of Turkey (TUBITAK) under project #9140014, #9150181, and by the Luxembourg National Research Fund (FNR) (FNR/P10/03). We acknowledge networking support by European Cooperation in Science and Technology Action IC1404 "Multi-Paradigm Modelling for Cyber-Physical Systems".

REFERENCES

[1] Airbus. 2017. http://www.airbus.com/. (2017).

[2] Netta Aizenbud-Reshef, Richard F. Paige, Julia Rubin, Yael Shaham-Gafni, and Dimitrios S. Kolovos. 2005. Operational Semantics for Traceability. In *ECMDA Traceability Workshop (ECMDA-TW'05)*. 8–14.

[3] Alloy4graph. 2017. http://alloy.mit.edu/alloy/documentation/alloy-api/. (2017).

[4] Alloy4viz. 2017. http://alloy.mit.edu/alloy/documentation/alloy-api/. (2017).

[5] Kodkod API. 2017. http://emina.github.io/kodkod/release/current/doc/. (2017).

[6] Jane Cleland-Huang, Carl K. Chang, and Mark J. Christensen. 2003. Event-Based Traceability for Managing Evolutionary Change. *IEEE Transactions on Software Engineering* 29, 9 (2003), 796–810.

[7] Nikolaos Drivalos, Dimitrios S. Kolovos, Richard F. Paige, and Kiran J. Fernandes. 2008. Engineering a DSL for Software Traceability. In *1st International Conference on Software Language Engineering (SLE'08)*. 151–167.

[8] Eclipse. 2017. https://eclipse.org. (2017).

[9] Alexander Egyed. 2003. A Scenario-Driven Approach to Trace Dependency Analysis. *IEEE Transactions on Software Engineering* 29, 2 (2003), 116–132.

[10] Alexander Egyed and Paul Grünbacher. 2002. Automating Requirements Traceability: Beyond the Record & Replay Paradigm. In *17th IEEE International Conference on Automated Software Engineering (ASE'02)*. 163–171.

[11] Alexander Egyed and Paul Grünbacher. 2005. Supporting Software Understanding with Automated Requirements Traceability. *International Journal of Software Engineering and Knowledge Engineering* 15, 5 (2005), 783–810.

[12] Ferhat Erata, Moharram Challenger, Bedir Tekinerdogan, Anne Monceaux, Eray Tüzün, and Geylani Kardas. 2017. Tarski: A Platform for Automated Analysis of Dynamically Configurable Traceability Semantics. In *Proceedings of the Symposium on Applied Computing (SAC '17)*. ACM, New York, NY, USA, 1607–1614. https://doi.org/10.1145/3019612.3019747

[13] ITEA (Information Technology for European Advancement). 2014. ModelWriter: Text & Model Synchronized Document Engineering Platform. https://itea3.org/project/modelwriter.html. (Sep 2014).

[14] ITEA (Information Technology for European Advancement). 2015. ASSUME: Affordable Safe & Secure Mobility Evolution. https://itea3.org/project/assume.html. (Sep 2015).

[15] Ford-Otosan. 2017. http://www.fordotosan.com.tr. (2017).

[16] Arda Goknil, Ivan Kurtev, and Klaas Van Den Berg. 2014. Generation and Validation of Traces between Requirements and Architecture based on Formal Trace Semantics. *Journal of Systems and Software* 88 (2014), 112–137.

[17] Arda Goknil, Ivan Kurtev, and Jean-Vivien Millo. 2013. A Metamodeling Approach for Reasoning on Multiple Requirements Models. In *17th IEEE International Enterprise Distributed Object Computing Conference (EDOC'13)*. 159–166.

[18] Arda Göknil, Ivan Kurtev, and Klaas van den Berg. 2008. Change Impact Analysis based on Formalization of Trace Relations for Requirements. In *the ECMDA Traceability Workshop (ECMDA-TW'08)*. 59–75.

[19] Arda Goknil, Ivan Kurtev, and Klaas van den Berg. 2008. A Metamodeling Approach for Reasoning about Requirements. In *European Conference on Model Driven Architecture-Foundations and Applications (ECMDA-FA'08)*. 310–325.

[20] Arda Goknil, Ivan Kurtev, and Klaas van den Berg. 2010. Tool Support for Generation and Validation of Traces between Requirements and Architecture. In *the 6th ECMFA Traceability Workshop (ECMFA-TW'10)*. 39–46.

[21] Arda Goknil, Ivan Kurtev, Klaas van den Berg, and Wietze Spijkerman. 2014. Change Impact Analysis for Requirements: A Metamodeling Approach. *Information and Software Technology* 56, 8 (2014), 950 – 972.

[22] Arda Goknil, Ivan Kurtev, Klaas van den Berg, and Jan-Willem Veldhuis. 2011. Semantics of Trace Relations in Requirements Models for Consistency Checking and Inferencing. *Software and System Modeling* 10, 1 (2011), 31–54.

[23] Bikash Gyawali, Anastasia Shimorina, Claire Gardent, Samuel Cruz-Lara, and Mariem Mahfoudh. 2017. Mapping Natural Language to Description Logic. In *14th European Semantic Web Conference (ESWC'17)*. 273–288.

[24] Havelsan. 2017. http://www.havelsan.com.tr. (2017).

[25] ISO. 2017. ISO-26262: Road vehicles – Functional safety. (2017).

[26] Daniel Jackson. 2012. *Software Abstractions: Logic, Language, and Analysis*. MIT press.

[27] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2008. Detecting and Repairing Inconsistencies across Heterogeneous Models. In *1st International Conference on Software Testing, Verification, and Validation*. 356–364.

[28] Luis C. Lamb, Waraporn Jirapanthong, and Andrea Zisman. 2011. Formalizing Traceability Relations for Product Lines. In *the 6th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'11)*. 42–45.

[29] Balasubramaniam Ramesh and Matthias Jarke. 2001. Toward Reference Models for Requirements Traceability. *IEEE Transactions on Software Engineering* 27, 1 (2001), 58–93.

[30] RTCA and EUROCAE. 2017. DO-178C: Software Considerations in Airborne Systems and Equipment Certification. (2017).

[31] Mehrdad Sabetzadeh, Shiva Nejati, Sotirios Liaskos, Steve Easterbrook, and Marsha Chechik. 2007. Consistency Checking of Conceptual Models via Model Merging. In *15th IEEE International Requirements Engineering Conference (RE'07)*. 221–230.

[32] IEEE Computer Society, Pierre Bourque, and Richard E. Fairley. 2014. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0* (3rd ed.). IEEE Computer Society Press, Los Alamitos, CA, USA.

[33] Alfred Tarski. 1941. On the Calculus of Relations. *The Journal of Symbolic Logic* 6, 03 (1941), 73–89.

[34] David ten Hove, Arda Goknil, Ivan Kurtev, Klaas van den Berg, and Koos de Goede. 2009. Change Impact Analysis for SysML Requirements Models based on Semantics of Trace Relations. In *the ECMDA Traceability Workshop (ECMDA-TW'09)*. 17–28.

[35] Emina Torlak. 2008. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[36] Emina Torlak and Daniel Jackson. 2007. Kodkod: A Relational Model Finder. In *the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*. 632–647.