PIACERE

Deliverable D4.2

Infrastructural Model and code verification – v2

Editor(s):	Michele Chiari Matteo Pradella
Responsible Partner:	Politecnico di Milano/Polimi
Status-Version:	Final 1.0
Date:	24.11.2022
Distribution level (CO, PU):	PU

Project Number:	101000162
Project Title:	PIACERE

Title of Deliverable:	Verify the trustworthiness of Infrastructure as Code – Infrastructural model verification
Due Date of Delivery to the EC	30.11.2022

Workpackage responsible for the Deliverable:	WP4 Verify the trustworthiness of Infrastructure as Code
Editor(s):	Politecnico di Milano/Polimi
Contributor(s):	Michele Chiari – Polimi, Matteo Pradella – Polimi
Reviewer(s):	Adrián Noguero - Go4IT
Approved by:	All Partners
Recommended/mandatory readers:	WP3, WP7

Abstract:	This deliverable describes the development of the model checking tool for IaC in the PIACERE project. The DOML Model Checker (KR5) performs consistency checks on DOML models provided by the user, highlighting common mistakes and issues that might prevent the specified infrastructure from being deployed successfully. KR5 can be run as a stand-alone service that can be accessed through REST APIs and has been integrated with the PIACERE IDE (KR2) to offer a graphical interface to the user. This deliverable describes KR5 in terms of user interface, functionalities, software architecture and implementation choices.
Keyword List:	DOML, Model Checker, Automatic Verification, SMT Solver
Licensing information:	This work is licensed under Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) http://creativecommons.org/licenses/by-sa/3.0/
Disclaimer	This document reflects only the author's views and neither Agency nor the Commission are responsible for any use that may be made of the information contained therein

Document Description

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
v0.1	07.10.2022	TOC	Michele Chiari (Polimi)
V0.2	20.10.2022	Initial draft	Michele Chiari (Polimi)
V0.3	28.10.2022	Post internal review version	Michele Chiari (Polimi)
V0.4	17.11.2022	Appendix	Matteo Pradella (Po-limi)
V1.0	24.11.2022	Ready for submission	Juncal Alonso (TEC-NALIA)

Table of contents

TERMS AND ABBREVIATIONS 6
EXECUTIVE SUMMARY 7
1 INTRODUCTION 8
1.1 ABOUT THIS DELIVERABLE 8
1.2 DOCUMENT STRUCTURE 8
2 IMPLEMENTATION 9
2.1 PURPOSE 9
2.2 APPROACH 9
2.2.1 SMT SOLVERS AND Z3 9
2.2.2 THE OBJECT CONSTRAINT LANGUAGE AND ITS ISSUES 10
2.3 CHANGES IN V2 12
2.4 FUNCTIONAL DESCRIPTION 12
2.4.1 VALIDATION OF THE COMPONENT 14
2.4.2 FITTING INTO THE OVERALL PIACERE ARCHITECTURE 14
2.5 TECHNICAL DESCRIPTION 15
2.5.1 INTERFACE WITH THE IDE AND WORKFLOW 15
2.5.2 SOFTWARE ARCHITECTURE 16
2.5.3 TECHNICAL SPECIFICATIONS 17
3 DELIVERY AND USAGE 19
3.1 PACKAGE INFORMATION 19
3.2 INSTALLATION INSTRUCTIONS 20
3.2.1 RUNNING FROM SOURCES 20
3.2.2 BUILDING AND RUNNING THE DOCKER IMAGE 20
3.2.3 BUILDING THE DOCUMENTATION 20
3.3 USER MANUAL 21
3.4 LICENSING INFORMATION 22
3.5 DOWNLOAD 22
4 FUTURE PLANS 23
5 CONCLUSIONS 24
6 REFERENCES 25
7 APPENDIX 27

List of tables

TABLE 1. REQUIREMENTS FOR KR5 12
TABLE 2. COMMON REQUIREMENTS CHECKED BY THE DMC 14
TABLE 3. RESPONSE FORMAT OF THE API ENDPOINT 21

List of figures

FIGURE 1. SEQUENCE DIAGRAM OF THE EXTERNAL AND INTERNAL BEHAVIOUR OF THE DMC AT A HIGH-LEVEL. 15
FIGURE 2. COMPONENT DIAGRAM OF THE DMC 16
FIGURE 3. INTERNAL SEQUENCE DIAGRAM OF THE DMC 17
FIGURE 4. RESPONSE OF THE DMC TO A REQUEST CONTAINING AN ERRONEOUS DOMLX MODEL 22

Terms and abbreviations

BNF	Backus-Naur Form
CSP	Cloud Service Provider
DevOps	Development and Operation
DMC	DOML Model Checker
DoA	Description of Action
DOML	DevSecOps Modelling Language
DOMLX	DOML XMI format
EC	European Commission
EMF	Eclipse Modelling Framework
GA	Grant Agreement to the project
HTML	HyperText Markup Language
IaC	Infrastructure as Code
IEP	IaC execution platform
IM	Intermediate Model
IMC	Intermediate Model Checker
IOP	IaC Optimization
KPI	Key Performance Indicator
KR	Key Result
MC	Model Checker
OCL	Object Constraint Language
RMDF	Resource Model Definition
SAT	Propositional Satisfiability
SMT	Satisfiability Modulo Theories
SW	Software
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

Executive Summary

This deliverable is the second in a series of three and supersedes D4.1. It describes the work that the POLIMI contributors have done on the development of the DOML Model Checker (DMC, KR5, previously called the Verification Tool), the model checking tool for IaC in the PIACERE project. This is the result of task T4.1.

The DMC is part of the PIACERE verification tools, together with KR6 and KR7. While KR6 and KR7 focus on issues in the IaC generated by the ICG, the DMC works directly on user-supplied DOML models. It is part of the PIACERE design-time workflow, and it has been integrated with the design-time tools, so that it can be invoked by the graphical user interface offered by the IDE.

The DMC's main purpose is to check DOML models for consistency and completeness issues. It checks models against a set of pre-defined common requirements, and reports violations to the user through error messages. It helps users in developing DOML models that can be used to successfully deploy cloud applications on an appropriate infrastructure.

In this deliverable, we first present the purpose of the DMC in terms of the features it is required to offer. We assess the level of fulfilment of such features by analysing the requirements laid out within WP2.

We then discuss our choice of the approach underlying the implementation, by comparing it with competing approaches and by taking into account the results of our experiments carried out with the prototypes presented in the first version of this deliverable. The DMC has been implemented by using an SMT solver as its backend. We discuss the benefits of this choice and present the overall architecture of the tool.

The DMC is offered as a web service that can be reached through RESTful APIs. We describe the usage of these APIs, and we provide instructions to install and use the tool.

Finally, we discuss our future plans for the DMC. Development will focus mainly on enhancing coverage of the requirements defined within WP2. We plan to investigate the possibility of providing additional automated error-fixing features.

The third and final version of this deliverable is planned for month 30 of the project, which will be May 2023.

1 Introduction

The present deliverable describes the current state of the contribution by the Politecnico di Milano (POLIMI) partner to WP4 “Verify Trustworthiness of Infrastructure as Code”, with the aim of producing KR5 “DOML Model Checker” (DMC). This deliverable is the second one in a series of three deliverables. It provides updates to the previous deliverable D4.1, describing the activities concerning KR5 performed throughout the second year of the project.

1.1 About this deliverable

The purpose of WP4 is to assess the trustworthiness of IaC artifacts with respect to code quality, and safety and security of the overall architecture and its components. KR5 contributes to this aim by providing static analysis tools to ensure correctness, safety, performance, and data transfer privacy of all application components.

In D4.1 we investigated possible solutions for the implementation of KR5 by developing two software prototypes, one of them based on the Prolog programming language [1], and one based on the Z3 Theorem Prover [2], a SMT (Satisfiability Modulo Theories) solver [3]. Our evaluation of the prototypes and the feedback received from the other partners led to the decision to choose the Z3-based approach. KR5 has been thus developed by following this approach, and we describe it in this deliverable.

1.2 Document structure

The document is divided in the following sections:

- Section 1 presents an overall description of this deliverable;
- Section 2 focuses on the purpose, implementation details, functional requirements along with validation and technical description of KR5;
- Section 3 describes the delivery and usage of the developed tool;
- Section 4 lists the planned improvements to KR5;
- Section 5 summarizes the achievements of this deliverable and draws conclusions;
- Section 6 contains the bibliography.

2 Implementation

2.1 Purpose

One of the main goals of the PIACERE framework is to enable users to easily specify and deploy complex cloud applications and the underlying infrastructure with little effort. This kind of task requires a considerable experience in cloud application design, and inexperienced users may incur in mistakes that prevent them from obtaining a working deployment or expose them to security and privacy risks. The DMC assists the user by identifying the most common mistakes that may prevent a DOML model from describing a functional and safe infrastructural deployment.

In short, the DMC interprets a DOML model received in input by taking its semantics into account and checks it against a collection of pre-defined properties entailing its consistency and correctness.

2.2 Approach

During the first year of the project, we developed two prototypes for KR5 in order to choose between two different approaches: Prolog and SMT solving.

The requirements for the approach implied that the ideal approach should

1) offer the greatest expressive power in terms of checkable requirements, and
2) be the best suited for modelling IaC.

Concerning point 1), the target technologies should allow for expressing the requirement specifications to be checked in languages that are well-known to be expressively powerful, and whose expressiveness has been thoroughly characterized from the theoretical point of view. Moreover, ease-of-use and a not excessively steep learning curve are other desirable features.

Point 2) restricts our choice to tools capable of modelling relational data. In fact, the DOML language is largely declarative, and DOML models contain associations between different elements of the described deployment.

According to our evaluation of the two prototypes presented in D4.1, we decided to choose the approach based on the Z3 Theorem Prover. Thus, KR5 has been developed using this SMT solver as its backend. We give a general description of SMT solving and Z3 in Section 2.2.1.

Since several tools developed by other WPs rely on the Eclipse framework, another possible option for the backend would have been the Eclipse Modeling Framework (EMF), and its validation framework, with the Object Constraint Language (OCL). While this would have the advantage of an easier integration with an Xtext DOML parser, we decided to employ the SMT-solver backend because it allows for more expressive specification languages. One of the reasons for not choosing OCL we reported in D4.1 was the use of the Eclipse Theia framework for the IDE, which still lacks full EMF and OCL integration. However, this decision was revoked, and the IDE is being developed in the classic Eclipse framework. While this would allow us to use OCL, there are other reasons why an SMT solver is a better choice, which we thoroughly analyze in Section 2.2.2.

2.2.1 SMT Solvers and Z3

In this section, we briefly describe SMT solving, the approach we use to develop the backend of the DMC.

Satisfiability Modulo Theories (SMT) solvers [3] have recently been introduced as an extension of SAT solvers. SAT solvers are programs that receive in input a Boolean propositional formula and look for an assignment of its variables that satisfies it. If no such assignment can be found, it means the formula cannot be satisfied. Otherwise, the SAT solver returns a satisfying assignment, which is a model for the formula.

SMT solvers are essentially SAT solvers that integrate solvers for specific first-order theories, such as real numbers, integers, bit-vectors, arrays etc. Their inputs consist of quantifier-free first-order formulas, possibly containing terms in the theories supported by the solver. SMT solvers, too, try to find a satisfying assignment for variables in the formula, and return it if it exists. Modern SMT solver often accept inputs in more expressive fragments of first-order logic, possibly even containing quantifiers (although termination is not guaranteed in this case). Since SMT solvers often support the definition of finite relations in their input languages, they are capable of modeling IaC artifacts, and the rich assortment of available theories allows for very expressive queries.

One of the most successful uses of SAT (and later SMT) solvers is model checking. The idea behind SMT-based model checking is to model the system to be checked as an SMT formula. The requirements against which to check the system are also expressed as SMT formulas, and their negation is added in conjunction to the formula modelling the system. The formula obtained in this way is then checked for satisfiability. If it is unsatisfiable, it means the system satisfies its requirements. If it is satisfiable, the so-obtained variable assignment is a counterexample for the system requirement to be checked.

Several SMT solvers are currently available on the market. We chose Z3 [4] [2] as our reference solver because it is open source and is one of the most popular, which means the community around it is quite large. Moreover, it has been developed within Microsoft Research, which gives even more guarantees of continued support and robustness. The number of theories it supports is also quite large, which gives us the possibility of modelling many aspects of the DOML.

Moreover, most SMT solvers support a unified input language, called SMTLIB, which makes it easy to switch between different solvers (e.g., to pick the best performing one), and avoids vendor lock-in.

2.2.2 The Object Constraint Language and its issues

The Object Constraint Language (OCL) is a modelling language that was added as an extension to UML [5]. Whereas UML contains comprehensive features for describing the objects and classes that compose a system, the purpose of OCL is to augment system descriptions with business rules, well-formedness rules and other semantic requirements about such objects and classes. In particular, some (but not all) of the kinds of rules and requirements that OCL was devised to express are [6]:

- Class invariants
- Initialization of class fields
- Specification of derived models
- Query operations
- Specification of constraints on operations and business rules

OCL has been devised as a mostly declarative language. It supports interacting with elements of UML class diagrams, and it features a complex type system based on classes and inheritance.

While OCL has been quite successful and has reached a rather wide adoption, some of its shortcomings have been highlighted throughout the years.

2.2.2.1 General issues

Vaziri and Jackson [7] examine some of such shortcomings, especially with respect to syntax and ease-of-use. While the main purpose of their paper is to "advertise" the Alloy framework, developed by their group at MIT, some of the issues they identify are still relevant for evaluating the use of OCL within the DMC in PIACERE. We list some of such issues in the following.

1. First, OCL is very tightly integrated into UML, from which it inherits a very complex type system relying on classes, objects and inheritance. DOML has been specified based on UML and, under the hood, it is object-oriented and uses inheritance. However, a DOML user only needs to be familiar with the specific class hierarchy used in DOML to write correct deployment models. On the other hand, the OCL type system is significantly more general and, thus, more complex. This complexity and the necessity to directly refer to UML concepts might be an obstacle to the definition of an independent requirement specification language for the DMC. If OCL was used, such a language would have to be tightly coupled with UML concepts, being potentially confusing to a user that has only seen the DOML syntax and knows very little of its intermediate EMF representation.
2. Although OCL has been devised as a declarative language, its syntax and semantics are still quite "operational". In particular, OCL requirements may sometimes need to be written in the form of loops. Instead, we would like to rely on a completely declarative language, since we want the DMC input language to also be as declarative as possible, in order not to require programming knowledge to be used.

The Z3 SMT solver suffers from none of such shortcomings. With particular reference to no. 2, Z3 uses fragments of first-order logic, which can be seen as one of the most declarative languages and is very close to human reasoning. This could also be an advantage when devising the requirement specification language for the DMC.

2.2.2.2 Decidability and termination

Another issue with OCL is decidability of its constraints and queries. Due to the presence of recursion and loops, OCL is undecidable in general, and evaluation of its expressions may not terminate [6]. This may happen not only when analysing infinite models, but also when finite models contain relations with loops [7].

This can be an issue in terms of user experience because it would require imposing a timeout on model checking, rendering it inconclusive in certain cases. While a workaround for this issue could be to specify all pre-defined constraints so that their evaluation always terminates, the issue would remain if we allowed the user to specify custom requirements. In this case, it could be very difficult for the user to understand when their requirements' evaluation may incur in non-termination.

This issue is much easier to control when using Z3. In fact, Z3 relies on well-defined fragments of first-order logic, for which it is easier to ensure termination. In particular, if we constrain ourselves to finite models and theories, termination is ensured. Note that there is no reason to think that this would be an excessive limitation, as the purpose of DOML is to model cloud deployments, which are unlikely to contain an infinite number of objects.

Although this does not rule out the possibility of timeouts for model checking, as Z3 could also take an excessive time to solve constraints, the assurance of termination is still a useful improvement.

2.2.2.3 Synthesis and automated repair

Another advantage of Z3 (and of SMT solvers in general) is that it natively supports model synthesis. Suppose we have a DOML model with some issue which is detected by the DMC. We would like to suggest to the user a possible way of fixing this issue, e.g., by changing some property, or by adding some component. This would be quite arduous to achieve with OCL, as it has been mostly devised as a "query" language, i.e., a language that only allows to check if some requirements have been satisfied, or to identify system components that satisfy some property. It is unclear whether it could be possible to also synthesize new model components by exploiting OCL's feature of defining derived models. Certainly, this would require separate work to specify ways on correcting each class of model defects and would be hardly generalizable to user-specified requirements.

SMT solvers, on the other hand, natively support model synthesis. In fact, it suffices to describe the model and the desired property, and an SMT solver can be queried for a new model satisfying the property. If no such model can be found without modifying the initial one, it is possible to identify the offending component and remove it, so that the SMT engine can infer a new, correct version of the component.

2.3 Changes in v2

The first version of this deliverable only consisted of two proof-of-concepts of the model checker, developed with the aim of performing experiments to identify the most suitable approach to design its engine. These prototypes did not target the DOML because it was at a too early stage of development, so one of them targeted TOSCA, and the other one an older—and later discarded—version of the DOML [8].

The current version of the deliverable (v2) contains the final prototype of the model checker, which can process IaC models in the most current DOML version, described in deliverable D3.2. We describe its features, intended user interaction, integration with the PIACERE design-time tools, and implementation in terms of architecture and employed techniques.

2.4 Functional Description

Partners of the PIACERE consortium have defined a set of requirements that KR5 must satisfy, in terms of core features aimed at verification of DOML models and integration with other tools developed within the PIACERE framework (the IDE in particular). These requirements are also available in the PIACERE architectural specification [8], and we report those related to KR5 in Table 1.

Table 1. Requirements for KR5

#REQ	Description		Status	Requirement coverage at M24
REQ95	VT tools (model checker) must be able read DOML language (ex REQ56)	MUST HAVE	advanced	Full
REQ103	Verification Tool (model checker) must verify the structural consistency of the DOML models	MUST HAVE	advanced	Checks for common inconsistencies in DOML models
REQ104	Verification Tool (model checker) must verify the correctness of DOML models, with respect to some	MUST HAVE	none	

	correctness properties provided in DOML			
REQ105	Verification Tool (model checker) must verify the completeness of DOML models	MUST HAVE	basic	Checks for some commonly missing components in DOML models

The requirements of Table 1 can be summarized as follows:

- The model checker should offer APIs to allow other PIACERE tools (namely, the IDE) to interact with it to check the correctness of DOML models. In particular, the DMC must be able to parse and convert DOML models to a meaningful internal representation that allows for reasoning on their semantics (REQ 95).
- The DMC must then verify the correctness of DOML models in three ways:
    - It must verify the *structural consistency* of DOML models (REQ103), i.e., the absence of contradicting statements, duplicated or conflicting elements, and other issues that may prevent the ICG to generate IaC code for a working deployment.
    - It must verify the *completeness* of DOML models (REQ104), i.e., it must check whether they contain all elements needed to obtain a deployment that actually works at runtime. For instance, it should check that the model contains networks properly configured to let nodes communicate when needed by the applications deployed on them.
    - It must be able to verify other requirements expressed by the user in a domain specific language (REQ105). Thus, the user must be able to assert that their DOML model has some feature of interest, and the model checker must consequently check whether this is the case.

At this time, most of the requirements have been fulfilled, and the remaining ones are work-in-progress.

REQ95 has been fulfilled by using an intermediate XML-based representation of DOML models generated by the IDE. Thus, the actual parsing of DOML models only happens in the IDE, which then communicates with the DMC through a machine-readable format. The DMC's architecture has been designed to make it flexible with respect to changes to the class hierarchy underlying the DOML. Thus, it is possible to quickly adapt the DMC's frontend to parse new versions of the DOML and to support multiple DOML versions at the same time.

Requirements REQ103 and REQ 105 are being fulfilled by equipping the DMC with a collection of default requirements that check properties of DOML models concerning their structural consistency and completeness. This collection is continuously augmented with new requirements motivated by new features added in new versions of the DOML and, most importantly, by the feedback received from use case owners. We describe this collection of default (or *common*) requirements in Table 2.

Table 2. Common Requirements checked by the DMC

Requirement	Description
All virtual machines must be connected to at least one network interface.	Virtual machines can communicate with other components of a deployment or with external clients only through an appropriately configured network. This check makes sure no virtual machines are locked in.
All software packages can see the interfaces they need through a common network.	This check makes sure all exposed and consumed software interfaces at the application layer level have been concretized through a network connection that allows the involved components to communicate.
There are no duplicated interfaces.	Checks whether two or more interfaces have been assigned the same IP address.
All software components have been deployed to some node.	Makes sure that all software components specified in the application layer have been associated to at least one node in the abstract infrastructure layer through the currently active deployment.
All abstract infrastructure elements are mapped to an element in the active concretization.	Makes sure all abstract infrastructure nodes are concretized by the currently active concretization layer.
All elements in the active concretization are mapped to some abstract infrastructure element.	Makes sure each concrete infrastructure element is mapped to a node in the Abstract Infrastructure Layer.

2.4.1 Validation of the Component

The DMC has been validated both internally to WP4 and externally by the use case providers.

The internal validation has been carried out by creating an extensive set of regression tests, at least one for each one of the requirements reported in Table 2: for each requirement, we created a DOML model that violates it. All such DOML models are distributed together with the tool, to allow for regression testing.

The DMC has also been formally validated by the use case providers within Task 7.3 from WP7. This activity has yielded valuable feedback, resulting in both bug fixes and addition of new features, mostly in terms of new default requirements.

2.4.2 Fitting into the overall PIACERE Architecture

The DMC is part of the tools developed within WP4, together with the *IaC Security Inspector* (KR6) and the *IaC Component Security Inspector (KR7),* with the aim of verifying the correctness and trustworthiness of the IaC generated by the ICG. While KR6 and KR7 operate directly on the final IaC code, KR5 analyzes DOML models before the resulting IaC is generated.

In the overall architecture, the DMC is part of the *design-time tools,* a set of software tools that help the user in designing application and infrastructural deployments and in modeling them through DOML. All these tools are integrated with the IDE, which provides the main graphical interface with the PIACERE toolset for the users. After writing their DOML model with the help of the syntax checking performed by the IDE, users can invoke the DMC through a right-click menu entry in the IDE, to check it for errors. The IDE communicates with the DMC through a RESTful API. The integration of the DMC with the IDE will be explained in more detail in the next sections.

2.5 Technical Description

In this section, we report in detail how the DMC interacts with other design-time tools in the PIACERE framework and describe its internal software architecture.

2.5.1 Interface with the IDE and Workflow

Figure 1 shows the sequence diagram of the DMC. In this section we comment on its external behaviour, which consists of the interactions between the IDE and the DMC, while we describe its internal behaviour in detail in Section 2.5.2.

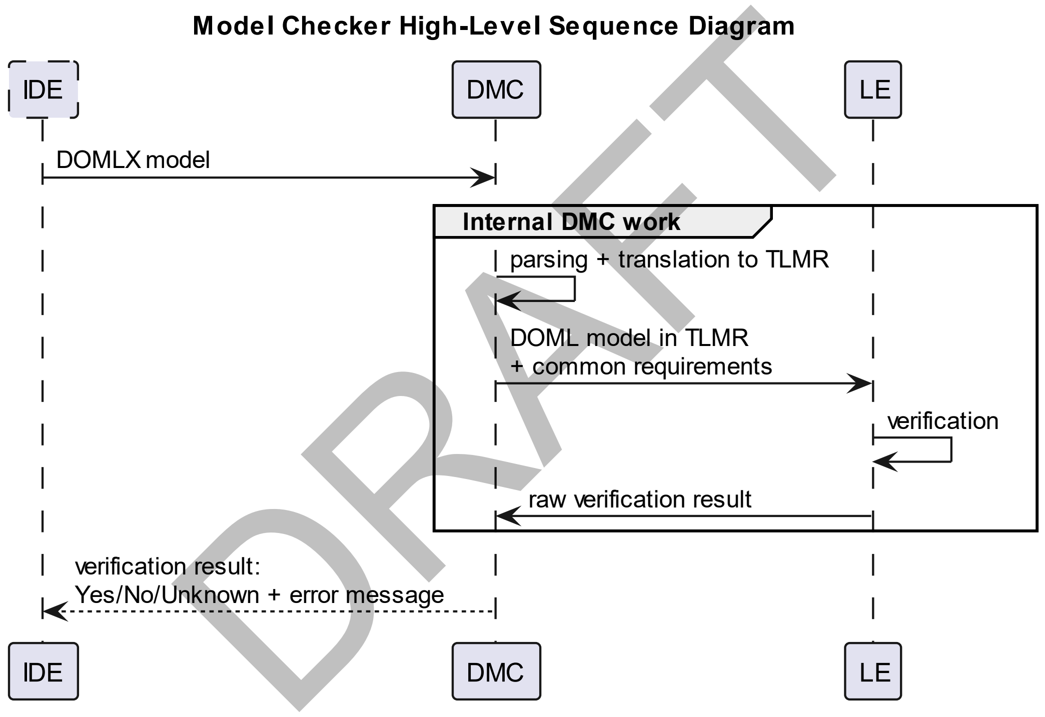


Figure 1. Sequence diagram of the external and internal behaviour of the DMC at a high-level

The interaction between the IDE and the DMC is carried out through the DMC's RESTful APIs. The IDE initiates the verification process by sending to the DMC a request containing the DOML model to be checked in DOML XMI (DOMLX) format, a machine-readable representation. The DOMLX file is generated automatically by the IDE, which parses DOML files created by the user. This allows the DMC to avoid the overhead due to DOML parsing, because the DOMLX format is based on XML, which makes it relatively easy to parse.

Then, the DMC verifies the DOML model against a set of common, pre-defined requirements, and sends the result back to the IDE. The result consists of a summary of the verification result, which can be one of the following:

- **Yes:** the DOML model satisfies all requirements
- **No:** the DOML model violates at least one requirement
- **Unknown:** verification was inconclusive (e.g., because it timed out)

If the result is “No”, the response also contains an error message that helps the use in understanding the issues with their DOML model and fix them. If more than one requirement is violated, the error message is actually the concatenation of multiple messages, one for each violated requirement.

In case the DMC incurs in an error while reading the DOMLX model, for instance because it is malformed or because it is based on an unsupported DOML version, its response to the IDE contains both an error and a debug message.

2.5.2 Software Architecture

In this section, we describe in detail the internal architecture of the DMC. A high-level overview of its components is shown in Figure 2.

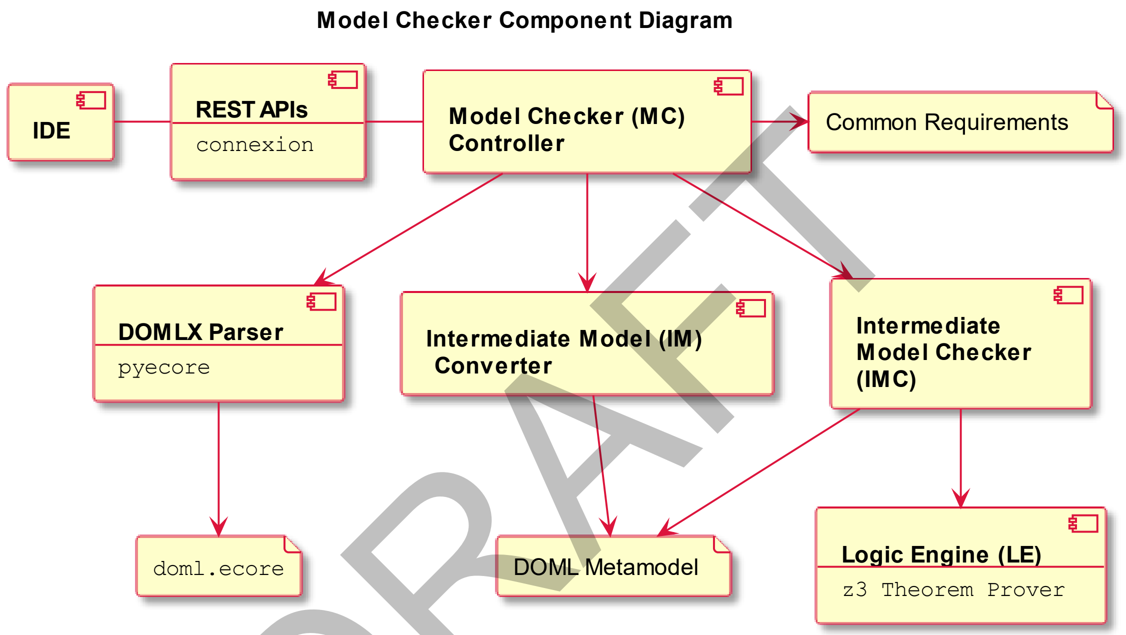

Model Checker Component Diagram
IDE
REST APIs
connexion
Model Checker (MC) Controller
Common Requirements
DOMLX Parser
pyecore
Intermediate Model (IM) Converter
Intermediate Model Checker (IMC)
doml.ecore
DOML Metamodel
Logic Engine (LE)
z3 Theorem Prover

Figure 2. Component Diagram of the DMC

The DMC accepts requests from the IDE through a RESTful API specified in OpenAPI [10], a self-documenting industry standard for specifying APIs. This enables its use by frontends other than the PIACERE IDE, if needed. A request sent to the main endpoint of the APIs triggers the model checking process, which is orchestrated by the Model Checker Controller.

The main verification process is performed by the Logic Engine, which consists of an SMT solver. Thus, the input DOMLX model must be translated into an input format compatible with the software APIs of the LE, called *Target Logic Model Representation (TLMR)*. This activity is carried out in several stages, each one managed by a different component:

- The **DOMLX Parser** parses the input DOML XMI files through an external library, translating them to custom Python objects. Since the DOML has been developed within the Eclipse Modelling Framework, this component needs an Ecore specification of the classes, attributes and references defined for the DOML language. The resulting Python objects are, however, specific to each DOML version. To make it easier to support different DOML versions within the DMC, they are translated to a more generic intermediate format.
- The **Intermediate Model Converter** converts DOML objects generated by the DOMLX parser into a simpler data structure based on Python dictionaries, called *Pythonic*

Intermediate Representation. This translation is guided by DOML Metamodels, which provide a machine-readable description of classes, attributes, and associations available in the DOML (this is partially redundant with Ecore files), together with some additional information on how to represent them internally.

- The **Intermediate Model Checker (IMC)** receives in input DOML models in the Pythonic intermediate representation and translates them into the TLMR, which consists of First-Order formulas employing some of the theories supported by the SMT solver implementing the LE. The IMC also initiates and manages the proper verification process by executing the LE and gathering its results. The common requirements against which the model is checked are supplied by the MC Controller already in TLMR.

This architecture allows the DMC to be more general with respect to its input format: it is parametric on the description of the DOML supplied by the Ecore file and the DOML Metamodel. Thus, to support different DOML versions, the DMC switches between different Ecore and metamodel files, and to add support for a new DOML version it suffices to provide new versions of these files, possibly with minimal changes to the code.

Figure 3 shows in more detail the interactions between the components described above.

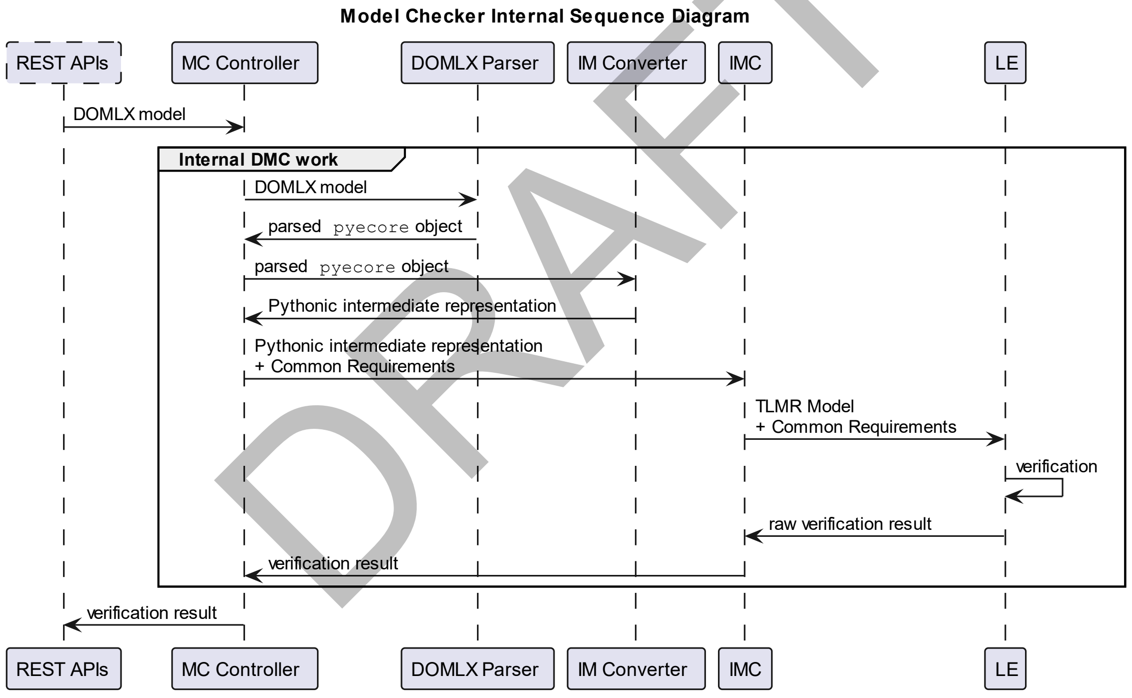


Figure 3. Internal Sequence Diagram of the DMC

2.5.3 Technical Specifications

The DMC has been written in the Python programming language [11], and its library dependencies are managed through the Poetry [12] development tool. The REST APIs, specified in OpenAPI [10], have been implemented with Connexion [12], a Python library that implements REST APIs directly from their OpenAPI specification. The DOMLX parser is based on the pyecore library [14]. The RestAPIs are documented through SwaggerUI [14], which generates online documentation directly from OpenAPI specifications, and the general DMC documentation is written in the *reStructured Text* format [15] and rendered with Sphinx [15]. The pytest [18] library is used for managing regression tests.

The DMC can be both run locally or through a Docker [18] container, whose Dockerfile is provided for easier setup (cf. Section 3.2 for instructions). The Docker image is based on the official Python Debian image [19], and uses the Uvicorn [20] web server to deploy the REST APIs.

3 Delivery and usage

In this section we describe the contents of the package in which the DMC is distributed and give usage and installation instructions.

3.1 Package information

The DMC is distributed as a source package, which allows for easily run the DMC directly or through a Docker container. Since the DMC is written in Python, it does not need to be built to be run.

The source package contains the following **directories** and *files*:

- **docs** – Directory containing the documentation sources as RST files and the configuration files needed for building it
- **mc_openapi** – directory containing the DMC sources including REST APIs
    - **assets** – directory containing files defining different supported versions of the DOML (Ecore and metamodel files)
    - **doml_mc** – sources of the model checker engine
        - **intermediate_model** – type definitions and auxiliary functions for the Pythonic intermediate representation
        - **xmi_parser** – sources implementing the IM Converter
        - **z3encoding** – type definitions and auxiliary functions for the TLMR
        - *common_reqs.py* – common requirements in TLMR
        - *consistency_reqs.py* – other consistency requirements for DOML models in TLMR
        - *imc.py* – IMC sources
        - *mc.py* – sources of the MC Controller
        - *mc_result.py* – class definitions and auxiliary functions for managing raw results of verification and error messages
    - **openapi**
        - *model_checker.yaml* – OpenAPI specification of the REST APIs
    - *app_config.py* – configuration file related to Connexion
    - *bytes_uri.py* – auxiliary file related to the implementation of the REST APIs
    - *handlers.py* – definitions of the REST APIs handlers
- **tests** – pytest test files
    - **doml** – DOML files used in regression tests
    - *test_mc_openapi.py* – definitions of pytest regression tests
- *dev-requirements.txt* – list of Python packages required to run the DMC and regression tests
- *docker-compose.yaml* – Docker-compose file for running the DMC container image and setting up appropriate port bindings
- *Dockerfile* – definition of the Docker image (cf. Section XX for instructions on how to build and run it)
- *poetry.lock* – Poetry configuration file
- *pyproject.toml* – Python project configuration file
- *README.md* – Readme containing basic running and building instructions
- *requirements.txt* – list of Python packages required to run the DMC

3.2 Installation instructions

The DMC can be run directly from its sources or by building and starting a Docker container. In both cases, the source package must be either downloaded directly (cf. Section 3.5) or by cloning the repository:

```
$ git clone https://github.com/michiari/piacere-mc-openapi
```

The repository contains three branches:

- **main** – development is tracked here. May contain unstable versions of the DMC.
- **y1** – DMC version developed in the first year of the project (stable)
- **y2** – DMC version developed in the second year of the project (stable)

The repository also contains tags for DMC releases.

3.2.1 Running from sources

This way of running the DMC is mainly intended for development purposes.

A recent (v3.9 or higher) version of the Python interpreter [22] and the Poetry [23] development tool need to be already installed in the system.

First, all necessary dependencies need to be installed with

```
$ poetry install
```

Then, the server can be run with

```
$ poetry run python -m mc_openapi
```

By default, the server is run on port 8080 of localhost.

Regression tests may be run with

```
$ poetry run python -m pytest
```

3.2.2 Building and running the Docker image

A Dockerfile is provided in the root of the source tree that allows to quickly generate a Docker image for running the DMC. First, the image must be built with

```
$ docker build -t wp4/dmc .
```

Then, the container must be run with

```
$ docker run -d -p 127.0.0.1:8080:80/tcp wp4/dmc
```

The Uvicorn server is bound to port 80 of the container. The command above binds this port to port 8080 of *localhost* (of course the user may choose their preferred port configuration).

3.2.3 Building the documentation

The documentation is written in RST format and can be built with Sphinx. To do so, run

```
$ poetry shell
```

And then

```
$ cd docs
$ make html
```

The documentation will be rendered in HTML format in directory `docs/_build`.

3.3 User Manual

The RESTful API offered by the DMC is very simple, as it consists of one single endpoint. It is documented through SwaggerUI, which can be reached at endpoint `ui` (e.g., `http://127.0.0.1:8080/ui`).

The main endpoint for the DMC is `/modelcheck`. It supports a POST request whose body contains the DOML model to be checked in XMI format. Additionally, an optional string parameter called `requirement` can be specified, containing a user-defined requirement for the DOML model to be checked against. However, the functionality behind the requirement parameter has not been implemented yet, so only requests containing a DOMLX file to be checked against the default common requirements are supported. This POST request triggers the verification process, and its result is sent back to the user as its response. The response format is documented in Table 3.

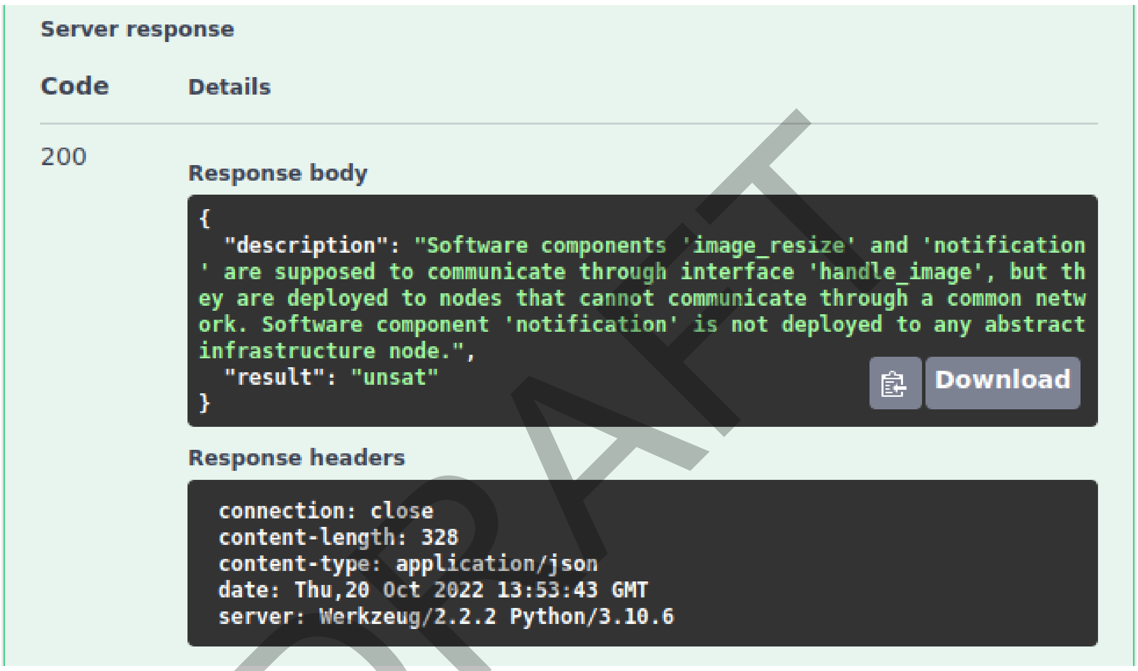
A Gherkin scenario for the usage of the model checker is presented in the appendix.

Table 3. Response format of the API endpoint

HTTP Code	Description	Parameters	
		Name	Description
200	**OK** The verification process was successful.	result	One of **sat** (the DOML model satisfies the requirements), **unsat** (the model violates at least on requirement), **dontknow** (the verification process was inconclusive, e.g. because it timed out)
		description	Explanation of the result. When some requirements are violated, an error message containing violating components is reported.
400	**Malformed Request** Usually because the supplied DOMLX model is malformed or of an unsupported DOML version.	message	A user-friendly message describing the error.
		debug_message	An error message useful to the developers to understand the cause of the error.
		timestamp	Time and date of the erroneous request.
500	**Internal error**	message	A user-friendly message describing the error.

	Other kind of error internal to the webserver or the DMC.	debug_message	An error message useful to the developers to understand the cause of the error.
		timestamp	Time and date of the erroneous request.

For example, sending a POST request to the /modelcheck endpoint containing the DOMLX file test/doml/faas.domlx provided in the repository triggers the response shown in Figure 4. The result is “unsat” because the model violates two requirements. The “description” field explains which requirements are violated, also reporting the violating components by name.



```
{
  "description": "Software components 'image_resize' and 'notification
' are supposed to communicate through interface 'handle_image', but th
ey are deployed to nodes that cannot communicate through a common netw
ork. Software component 'notification' is not deployed to any abstract
infrastructure node.",
  "result": "unsat"
}
```

```
connection: close
content-length: 328
content-type: application/json
date: Thu,20 Oct 2022 13:53:43 GMT
server: Werkzeug/2.2.2 Python/3.10.6
```

Figure 4. Response of the DMC to a request containing an erroneous DOMLX model.

3.4 Licensing information

The DMC is licensed under the open-source **Apache License 2.0** [24].

3.5 Download

The most updated versions of the DMC can be downloaded by cloning the following GitHub repository:

https://github.com/michiari/piacere-mc-openapi

Stable versions can be downloaded from the PIACERE public repository, which is updated less frequently:

https://git.code.tecnalia.com/piacere/public/the-platform/doml-model-checker

4 Future Plans

In the remaining six months allocated to development of the DMC we plan to extend its features to further enhance the support of requirements listed in Table 1.

In particular, more standard requirements will be added to further advance support of REQ103 and REQ105. Moreover, a domain-specific language for defining custom requirements will be defined to support user-supplied requirements, which will be directly included in DOML models to satisfy REQ104. Although it is not requested by formal requirements, verification of user-supplied requirements may be supported also through the optional "requirement" parameter of the API endpoint.

We plan to further explore the possibilities enabled by the SMT solver backend by exploring the possibility of synthesizing new components in input DOML models. In fact, it is possible to instruct the SMT solver to include additional components in a model. The SMT solver will then automatically configure them in order to satisfy the requirements. The main purpose of this activity is fixing erroneous or incomplete DOML models, or at least provide the user with suggested fixes.

5 Conclusions

In this deliverable, we presented the work carried out by PoliMi on the KR5 within WP4. While in the first version of this deliverable we only presented proof-of-concept prototypes that had the only purpose of exploring possible solutions by experimenting with rapid prototyping, in the second version we present a functioning version of the DMC, capable of reading and verifying DOML models in the intermediate format offered by the EMF framework.

The DMC is open-source and can be run both directly from sources and through a Docker image, which facilitates its deployment. The main way of using the DMC is through its RESTful APIs. However, also been integrated with the PIACERE IDE, providing a user-friendly way of using it within the PIACERE design-time tools.

This version of the DMC fulfils a considerable part of the requirements for KR5, and plans for implementing all requirements completely have been laid out. Moreover, we plan to explore the possibility of adding model synthesis features, to provide users with suggested fixes for errors in their DOML models.

6 References

[1] W. F. Clocksin and C. S. Mellish, Programming in Prolog, Springer-Verlag, 2003.

[2] L. Mendonça de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008*, Budapest, Hungary, 2008.

[3] C. W. Barret and C. Tinelli, "Satisfiability Modulo Theories," in *Handbook of Model Checking*, Springer, 2018, pp. 305-343.

[4] Microsoft Research, "The Z3 Theorem Prover," [Online]. Available: https://github.com/Z3Prover/z3. [Accessed 6 April 2022].

[5] The Object Management Group, *Object Constraint Language,* 2014.

[6] J. Cabot and M. Gogolla, "Object Constraint Language (OCL): A Definitive Guide," in *Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012*, Bertinoro, Italy, Springer, 2012, pp. 58-90.

[7] M. Vaziri and D. Jackson, "Some Shortcomings of OCL, the Object Constraint Language of UML," in *TOOLS 2000: 34th International Conference on Technology of Object-Oriented Languages and Systems*, Santa Barbara, CA, USA, 2000.

[8] T. Mendez Ayerbe, Design and development of a framework to enhance the portability of cloud-based applications through model-driven engineering, Milano: Politecnico di Milano, 2021.

[9] A. Černivec, A. De La Fuente Ruiz, A. Motta, C. Nava, C. Bonferini, E. Di Nitto, E. Morganti, E. Osaba Icedo, J. López Lobo, G. Novakova Nedeltcheva, G. Benguria Elguezabal, I. Torres Boigues, L. Blasi, C. Matija, P. Skrzypek and R. Piliszek, "PIACERE DevSecOps Framework Requirements specification, architecture and integration strategy – v2," The PIACERE Consortium, 2022.

[10] SmartBear Software, "OpenAPI Specification," 2020. [Online]. Available: https://swagger.io/specification/. [Accessed 19 10 2022].

[11] Python Software Foundation, "Python.org," 2022. [Online]. Available: https://www.python.org/. [Accessed 19 10 2022].

[12] The Poetry Developers, "Poetry - Python dependency management and packaging made easy," 2022. [Online]. Available: https://python-poetry.org/. [Accessed 19 10 2022].

[13] Zalando SE, "Swagger/OpenAPI First framework for Python on top of Flask with automatic endpoint validation & OAuth2 support," 2022. [Online]. Available: https://github.com/spec-first/connexion. [Accessed 19 10 2022].

[14] V. Aranega, "A Python(nic) Implementation of EMF/Ecore (Eclipse Modeling Framework)," 2022. [Online]. Available: https://github.com/pyecore/pyecore. [Accessed 19 10 2022].

[15] SmartBear Software, "REST API Documentation Tool Swagger UI," 2022. [Online]. Available: https://swagger.io/tools/swagger-ui/. [Accessed 19 10 2022].

[16] The Sphinx Developers, "reStructuredText," 2022. [Online]. Available: https://www.sphinx-doc.org/en/master/usage/restructuredtext/index.html. [Accessed 19 10 2022].

[17] The Sphinx Developers, "Sphinx Python Documentation Generator," 2022. [Online]. Available: https://www.sphinx-doc.org/en/master/index.html. [Accessed 19 10 2022].

[18] The pytest Developers, "pytest: helps you write better programs," 2022. [Online]. Available: https://docs.pytest.org/. [Accessed 19 10 2022].

[19] Docker Inc., "Docker: Accelerated, Containerized Application Development," 2022. [Online]. Available: https://www.docker.com/. [Accessed 19 10 2022].

[20] The Docker Community, "python - Official Image | Docker Hub," 2022. [Online]. Available: https://hub.docker.com/_/python. [Accessed 19 10 2022].

[21] Encode OSS, "Uvicorn," 2022. [Online]. Available: https://www.uvicorn.org/. [Accessed 19 10 2022].

[22] Python Software Foundation, "BeginnersGuide/Download - Python Wiki," 2022. [Online]. Available: https://wiki.python.org/moin/BeginnersGuide/Download. [Accessed 20 10 2022].

[23] The Poetry Developers, "Poetry Installation Instructions," 2022. [Online]. Available: https://python-poetry.org/docs/#installation. [Accessed 20 10 2022].

[24] The Apache Software Foundation, "Apache License, Version 2.0," 2004. [Online]. Available: https://www.apache.org/licenses/LICENSE-2.0.html. [Accessed 20 10 2022].

7 Appendix

Here the scenario of DOML verification is represented in *Gherkin*, a cucumber specification format.

```
Scenario: DOML Verification - Model Checker (KR5)

      Given A DOMLX document

        And a check configuration is prepared

      When a user navigates to the DOMLX document

        And right-clicks on it

        And selects "Piacere"

        And selects "Validate DOML"

      Then a KR5 model checker is invoked

        And a <response> is returned


Examples: <response>

    | Response | Type    | Content             |

    | Correct  | Success |                     |

    | Wrong    | Error   | <Error description> |

    | Fail     | Error   | <Failure reason>    |


# Wrong means the provided DOML document contains mistakes

# Fail means verification failed for some reason (e.g., malformed model syntax, or time-
out)
```