PIACERE



# Deliverable D4.1

# Infrastructural model and code verification - v1

| | |
|---|---|
| **Editor(s):** | Michele Chiari<br>Michele De Pascalis<br>Matteo Pradella |
| **Responsible Partner:** | Politecnico di Milano/Polimi |
| **Status-Version:** | Final 1.0 |
| **Date:** | 30.11.2021 |
| **Distribution level (CO, PU):** | PU |

| | |
|---|---|
| **Project Number:** | 101000162 |
| **Project Title:** | PIACERE |

| | |
|---|---|
| **Title of Deliverable:** | Infrastructural model and code verification - v1 |
| **Due Date of Delivery to the EC** | 30.11.2021 |

| | |
|---|---|
| **Workpackage responsible for the Deliverable:** | WP4 Verify the trustworthiness of Infrastructure as Code |
| **Editor(s):** | Politecnico di Milano/Polimi |
| **Contributor(s):** | Michele Chiari – Polimi,<br>Michele De Pascalis – Polimi,<br>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.<br>Two prototype model checkers are presented. The first one targets the TOSCA IaC specification language and has been developed as a proof-of-concept for evaluating different possible model-checking backends. The second one is targeted to the PIACERE DOML specification language in one of its current form, still at an early development stage. It will be further implemented by exploiting the insights learned from the first prototype. The work described in this deliverable contributes to KR4 from WP4. |
| **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)<br>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 | 17.06.2021 | TOC | Matteo Pradella (Polimi) |
| V0.2 | 07.11.2021 | Added introduction and description of the first prototype | Michele Chiari (PoliMi) |
| V0.3 | 08.11.2021 | Added description of second prototype | Michele Chiari (PoliMi), Michele De Pascalis (PoliMi) |
| V0.8 | 09.11.2021 | Final editing | Michele Chiari (PoliMi), Matteo Pradella (PoliMi) |
| V0.9 | 30.11.2021 | Post-review editing | Michele Chiari (PoliMi), Matteo Pradella (PoliMi) |
| V1.0 | 30.11.2021 | Ready for submission | Leire Orue-Echevarria (TECNALIA) |

# Table of contents



# List of figures

# Terms and abbreviations

| | |
|---|---|
| BNF | Backus-Naur Form |
| CSP | Cloud Service Provider |
| DevOps | Development and Operation |
| DoA | Description of Action |
| EC | European Commission |
| EMF | Eclipse Modelling Framework |
| GA | Grant Agreement to the project |
| IaC | Infrastructure as Code |
| IEP | IaC execution platform |
| IOP | IaC Optimization |
| KPI | Key Performance Indicator |
| KR | Key Result |
| OCL | Object Constraint Language |
| RMDF | Resource Model Definition |
| SAT | Propositional Satisfiability |
| SMT | Satisfiability Modulo Theories |
| SW | Software |

# Executive Summary

This deliverable describes the work performed on the development of the model checking tool for IaC in the PIACERE project.

In particular, it illustrates two prototype model checkers. One of them targets the TOSCA IaC specification language and has been developed as a proof-of-concept for evaluating different possible model-checking backends. The other one is instead targeted to the PIACERE DOML specification language and is currently at an early development stage. It will be further implemented by exploiting the insights learned from the first prototype.

The work described in this deliverable contributes to Key Result (KR) 5 from WP4.

This deliverable consists of the following sections:

- Section 1 gives a more detailed introduction of the purpose and scope of the work done, also providing a rationale for the choices we made.
- Section 2 describes the first model-checker prototype.
- Section 3 describes the second model-checker prototype.
- Section 4 concludes the document.

The work described in this deliverable has the main purpose of investigating different techniques for model checking IaC.

Its main planned development is the implementation of the DOML-targeted model checker and its integration into the PIACERE framework.

# 1 Introduction

The present deliverable describes the current state of the contribution to WP4 "Verify trustworthiness of Infrastructure as code", with the aim of producing KR5 "PIACERE Model Checking Tool".

## 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. In particular, KR5 contributes to this aim by providing static analysis tools to ensure correctness, safety, performance and data transfer privacy of all application components.

In this deliverable we present prototype model checking tools that we developed with the aim of exploring the possible solutions to the challenges that WP4—and KR5 in particular—should address. Since the number of potentially suitable verification and model checking technologies available on the literature is considerable, we decided to search for a satisfactory solution by means of *rapid prototyping.*

**Rapid prototyping** consists of the creation of simple proof-of-concept prototypes for the candidate solutions, in order to be able to quickly assess their suitability for the purposes of WP4 by testing them directly on sample IaC applications of varying size and complexity [1] [2]. Rapid prototyping requires the use of tools and technologies that allow for a fast development of the target solutions, favouring the quick delivery of working solutions. Once one of such prototypes is deemed to be satisfactory in terms of functionality, it is polished and refined to fit into the requirement framework of the whole project, addressing the possible integration and user-experience issues due to its quick development—possibly also refactoring or redeveloping it in the framework that meets best the requirements set by the project.

The need for fast development of prototypes lead to our decision of using the Python programming language to implement them.

Thus, we first pruned the set of candidate solutions by only picking those:

1) offering the greatest expressive power in terms of checkable requirements, and
2) being 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 for whose expressiveness has also 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) is the one that restricts the search-space of model checking techniques the most. The DOML language, which is one of the planned key results of WP3, may also be largely declarative, according to the current state of development of KR1 [3]. Thus, a tool's capability of modelling relational data is essential for fulfilling such a requirement.

The two techniques that we deemed best-suited according to the above requirements are Prolog and Z3.

Prolog is a programming language featuring a declarative programming model, based on the *logic programming paradigm.* A Prolog program is a knowledge base containing *facts* and *rules*, which represent mathematical relations, and its execution is initiated by running a *query*. Thus, Prolog's knowledge bases are ideal for representing IaC. Moreover, Prolog's Turing

completeness ensures that all computable queries are expressible, and practically interesting requirements can be checked.

*Satisfiability modulo theories (SMT)* solvers are software tools that solve constraint systems expressed as first-order logic formulas containing predicates from several decidable theories, such as real numbers, linear integer arithmetic, lists, bit vectors, etc. When the quantifier-free fragment of first-order logic is used, the decidability of the underlying theories makes the whole satisfiability problem for such formulas decidable. Thus, SMT can be seen as an extension of propositional satisfiability (SAT), allowing for much more expressive queries thanks to the supported theories. Moreover, many SMT solvers also support satisfiability checking of formulas with quantifiers, although the undecidability of such problem may cause the non-termination of the solution process in some cases. 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.

Since several tools developed by other WPs rely on the Eclipse framework, another possible option for the backend could be 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 first explore Prolog and SMT-solver backends because they allow for more expressive specification languages. Moreover, the other WPs are planning to develop their tools—and the IDE, in particular—in the new web-based Eclipse Theia framework. Unfortunately, support for EMF and OCL in Theia seems to be still insufficiently advanced to build upon. We might reconsider using OCL if its support in Theia improves, and if we find out that it can express all kinds of constraints that are needed for DOML verification.

The first prototype tool works directly on TOSCA code. TOSCA is a widely used state-of-the-art IaC solution and is one of the main competitors of the PIACERE framework. TOSCA was chosen as the target language for this prototype because, at the time of development, the DOML language specification was still at an early stage of development, and thus not complete enough to build upon. Using TOSCA allows us to have a frontend representative of the one that the final DOML-based version of the tool will have. In fact, DOML is planned to be a largely declarative language, similarly to TOSCA. This prototype has been developed in two flavours: the first one uses Prolog as a backend for model checking, and the second one uses the Z3 SMT solver. Both versions of the tool can read TOSCA IaC artifacts as inputs, but they use two different domain-specific languages for specifying requirements, each one of them tailored to the kind of properties that can be most easily proved by the backend.

The second prototype tool, which is still work-in-progress, will instead address one of the first DOML proposals, to assess the feasibility of its model checking with the techniques we identified during the previous prototyping phase. In particular, the tool will use the Z3 SMT solver as a backend. Note that the DOML version addressed by this prototype is not the currently accepted one (presented in Deliverable 3.1), but an earlier one. Adaptation of this prototype to the most recent DOML version is left as future work.

## 1.2 Document structure

The document is divided in two main sections.

Section 2 presents our TOSCA model checking prototypes, while Section 3 presents the current state of development of the DOML model checker prototype.

Both sections are divided into two parts. The “implementation” part, which introduces the purposes and functionalities of the prototypes, and gives a high-level overview of the techniques

used to implement them, together with more technical details. The “delivery and usage” part gives more technical details on how to download, install and use the software artifacts.

Finally, Section 4 concludes the deliverable by summarizing the presented results.

# 2 TOSCA Model Checker Prototype

## 2.1 Implementation

In this section, we describe the implementation of the first prototype model checker, which addresses the TOSCA IaC language.

### 2.1.1 Functional description

The purpose of this prototype tool is to check standard consistency and user-defined requirements against TOSCA service templates. Thus, it reads as its input:

- A TOSCA service template in YAML syntax, and
- A list of requirements to be checked expressed in a human-readable formal language.

In the final version of the model-checking tool, the requirements to be checked will be part of a single input file, together with the system model, as DOML will directly support expressing both. However, since TOSCA does not support adding this kind of requirements, this tool reads them from a separate file.

As its output, the tool informs the user on whether the given service template satisfies the requirements. If this is not the case, the tool provides the user with guidance on how to solve the issue, by highlighting the offending components and showing which parts of the requirements they break.

We developed two versions of this prototype. They both use the same format for input TOSCA service templates, but different ones for formulas to be checked. This is a result of our experimentation on different backends for the model checking process. The two different formats are both tailored to the kinds of properties the underlying checker is capable of checking and try to be as user-friendly as possible. As we shall see in the next sections, they have been devised to naturally reason about the elements that compose TOSCA service templates.

In the two versions of this prototype, we respectively employ a Prolog implementation and an SMT solver. As we discuss in Section 1, Prolog implementations comprise a constraint solver, and Prolog is ideal for representing relational data such as TOSCA service templates. SMT solvers also allow for a natural and effective representation of TOSCA service model, while providing a greatly expressive language for expressing requirements, and efficient solving algorithms.

The purpose of this prototype is the sole feasibility evaluation of IaC verification through model checking based on constraint system and SMT solvers. As such, it will not be directly integrated into other parts of the PIACERE software framework. This prototype is a proof-of-concept, upon which we will build a model checking tool targeted at DOML (WP3, KR1) in particular, which will be integrated into the overall PIACERE framework. We discuss this part of the deliverable in Section 3 of this document. Nevertheless, to better serve the purpose of being a proof-of-concept for the tool described in Section 3, we developed it with a possible integration into the PIACERE IDE in mind.

#### *2.1.1.1 Overview of the approach*

In the following figure, we show an UML sequence diagram of a typical workflow intended for the model checking tool.

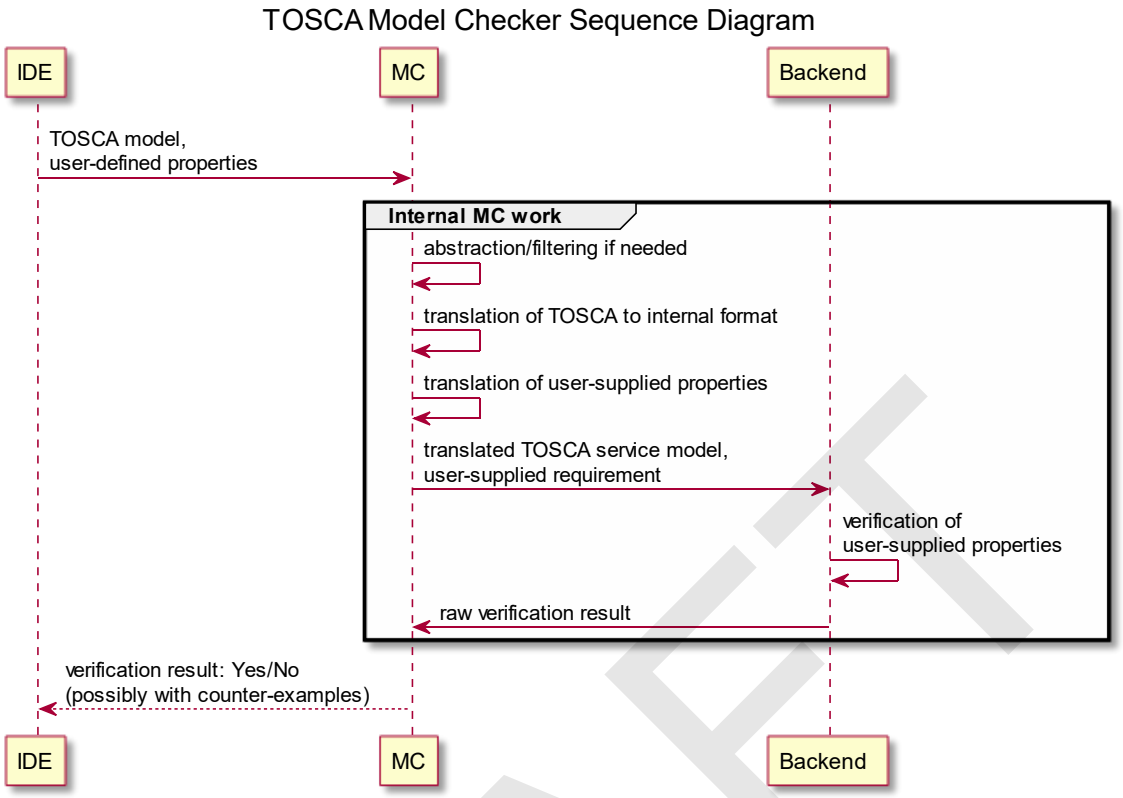


*Figure 1. Sequence diagram of the TOSCA model checker prototypes.*

The user may invoke the model checker directly from the IDE. The IDE gives the TOSCA service template in input to the model checking tool. The tool loads it in an internal in-memory representation, and then further translates it to an input representation for the backend.

The user may then specify complex properties through a YAML-based domain-specific language. Such requirements are parsed by the tool and translated to a representation readable by the backend. The backend solver is then invoked, and its results are reported to the user.

We again note that the interaction with the IDE has not been implemented, but rather the tool features a command-line user interface, upon which an integration with an IDE could be easily built.

### *2.1.1.2 Translation of the TOSCA model into an Internal Format*

First, the TOSCA YAML service models are parsed by an external library, and their contents are stored in a structured map-like data structure offered by the programming language used for the implementation. Then, they are translated to the input languages of the chosen backend.

#### 2.1.1.2.1 Prolog backend

When the Prolog backend is used, the elements composing the service model are represented as fact in a knowledge base. Nodes are represented as predicates that relate:

- An atom representing the type name;
- An atom representing the type's parent type;
- A list of property definitions;
- A list of capability definitions;
- A list of requirements.

Properties, capabilities, and requirements are represented as predicates on appropriate atoms and other predicates, all including an atom to uniquely identify them.

The resulting knowledge base is an accurate representation of the TOSCA service model and its semantics and can be directly consulted by the Prolog engine.

#### 2.1.1.2.2 SMT solver backend

Service models for the SMT solver backend are represented as quantifier-free first-order formulas, with predicates from the theories offered by the SMT solver we use—which, as we shall see in Section 2.1.2.3, is Z3.

Each element in the service model is represented by a unique name in an enumeration. Different classes of such names are defined for nodes, note types, and properties. Each node is related to its node type by a function. Another function relates nodes to their properties, and properties to their values, which may be of several different types offered by the solver, namely integers, floating-point numbers, strings, or lists thereof.

### *2.1.1.3 Checking of User-defined Properties*

Users can enter arbitrary properties defined in two domain-specific languages, each one tailored to one of the solvers used in the backend. Such requirement specifications are then parsed and translated into the input language of the target solver, which is run on them.

#### 2.1.1.3.1 Expressing and Verifying User-defined Properties (Prolog backend)

User-defined properties for the Prolog-based version of the prototype can be expressed in a YAML-based domain-specific language which allows for querying node types, properties, capabilities, and requirements. Each requirement to be checked is assigned a unique name and a natural-language description. Then, the actual requirement is a propositional formula (i.e., a combination of propositions with *and*, *or,* and *not* operators) on predicates regarding capabilities offered by a node, node properties, and TOSCA requirements. Node variables can also be defined.

A specification of the language in BNF form with regular expressions is shown below. Terminal symbols are in red and non-terminal symbols in bold face.

```
SpecDocument →
(- Query)*
```

A specification document (i.e., the document containing the requirements to be checked) is composed of a list of queries.

```
Query →
name: ID
description: string
check: Formula
```

A query is composed of a name, a description that should explain the meaning of the query in natural language, and a formula to check.

```
ID → [a-z_][A-Za_z0-9_]*
Variable → $ID
Atom → [^\'\n]+
```

Identifiers (ID) can be any alpha-numeric string starting with a lower-case letter. A variable is an ID prefixed with a `$`. Variables are used to match terms in the knowledge base and possibly report them in the output.

```
Term → Atom | integer | Variable | ListTerm | CompoundTerm

ListTerm →
(- Term)*

CompoundTerm →
ID:
(- Term)*
```

Terms are the building blocks of the knowledge base. Atoms are elementary terms that are only equal to themselves: they stand for the names of nodes, node types, and other elements in the topology. The specification can refer directly to components defined in the TOSCA model by using atoms with the same name. Terms can be grouped into lists or compound terms.

```
Formula → NodeFormula | NodeTypeFormula |
CapabilityTypeFormula | PolicyFormula | MatchFormula |
PredicateFormula | NotFormula | AndFormula | OrFormula
```

Formulas are used to check the existence of topology elements and their contents. They can be combined into more complex formulas using logical operators.

```
NodeFormula →
node:
(Atom | Variable): (NodeDef | {})

NodeDef →
(type: (Atom | Variable))?
(properties: (Properties | Variable))?
(capabilities: (Capabilities | Variable))?
(requirements: (Requirements | Variable))?

Properties →
((Atom | Variable): Term)+

Capabilities →
((Atom | Variable): Capability)+

Capability →
properties: Properties

Requirements →
(- (Atom | Variable): (Atom | Variable))+
```

A node formula describes the existence of a node template and can refer to its contents. Type, properties, capabilities and requirement specifications are optional. A node formula matches a node template if the type matches and all properties match some property contained in the node template, and the same is true for capabilities and requirements.

```
NodeTypeFormula →
node_type:
(Atom | Variable): (NodeTypeDef | {})

NodeTypeDef →
(derived_from: (Atom | Variable))?
(properties: (PropDefs | Variable))?
(capabilities: (CapDefs | Variable))?
```

```
(requirements: (ReqDefs | Variable))?


PropDefs →
((Atom | Variable):
(type: (Atom | Variable))?
(required: (true | false)?))+

CapDefs →
((Atom | Variable):
type: (Atom | Variable))+

ReqDefs →
(- (Atom | Variable): ReqDef)+

ReqDef →
(capability: (Atom | Variable))?
(node: (Atom | Variable))?
(relationship: (Atom | Variable))?
(occurrences: [unsigned_integer, (unsigned_integer |
UNBOUND)])?

CapabilityTypeFormula →
capability_type:
(Atom | Variable): (CapTypeDef | {})

CapTypeDef →
(derived_from: (Atom | Variable))?
(properties: (PropDefs | Variable))?

PolicyFormula →
policy:
(Atom | Variable): (PolicyDef | {})

PolicyDef →
(type: (Atom | Variable))?
(targets: [((Atom | Variable) (, (Atom | Variable))*)?])?
```

Node type, capability type and policy formulas are defined in a similar fashion.

```
MatchFormula →
match:
- Term
- Term
```

A match formula tries to unify two terms, assigning values to the variables to make them equal. For terms that contain no variable, this is the same as checking equality.

```
PredicateFormula →
ID:
(- Term)*
```

A predicate formula invokes a Prolog predicate defined outside of the specification language.

```
NotFormula →
not:
Formula

AndFormula →
```

```
and:
(- Formula)+

OrFormula →
or:
(- Formula)+
```

The language also supports Boolean combinations of requirements. The “not” formula succeeds if the contained formula fails. “and” and “or” formulas work as one would expect.

When a query is run, for every found result, the values of the variables mentioned in the description are interpolated to generate the output.

Consider the following example requirement:

```
- name: hardcoded_password
  description: Node $x has a hardcoded password.
  check:
    and:
    - node:
        $x:
          type: $nodeType
          properties:
            password: $p
    - predicate:
        type_offers_capability:
          args:
          - $nodeType
          - tosca.capabilities.Endpoint.Database
    - not:
        match:
        - $p
        - get_input:
            args: [$_]
```

The requirement’s name is “hardcoded_password” and, as suggested by its description, its purpose is to identify nodes that contain a hardcoded password (which is bad practice in TOSCA service models, as passwords should always be given as user inputs). The requirement first defines a variable $x representing a node that contains a property called “password”, which is bound to variable $p. This node’s type is bound to variable $nodeType. Then, the requirement asserts that node $x must be of a node type that offers a database capability. Finally, it says that property $p (the password field) should *not* be given as user input.

Requirements like the one above are automatically translated into Prolog rules. The Prolog engine then consults the knowledge base representing the service model to be checked and is queried by the tool for the rule representing the user-supplied requirement. Prolog tries to find models satisfying the requirement by solving the resulting constraint system. If one is found, then a node presents the defect described by the requirement. Such a node is bound to the variables defined in the specification and can be presented to the user as a debugging aid.

#### 2.1.1.3.2 Expressing and Verifying User-defined Properties (SMT solver backend)

User-defined properties for the SMT-solver-based version of the tool can be entered as assertions stating the equality or inequality of arbitrary expressions, which may contain constant literals of any type allowed for node properties, as well as variables referring to nodes. Access

to node properties is supported. Node variables must be declared by means of specific statements.

The grammar for this language is shown below in BNF form with regular expressions:

```
Requirement → (NodeDecl | Assertion)*
NodeDecl → node Var
Assertion → assert( BoolExpr )
BoolExpr → Ref == Ref | Ref != Ref
Ref → TypeRef | AccessRef | INT | STRING
TypeRef → type( Ref )
AccessRef → (Id | Var) (-> Id)*;
```

Non-terminal strings are shown in bold face. Variables, denoted as **`Var`**, can be any alpha-numeric string prepended by '$', while **`Id`** represents any alpha-numeric string.

A requirement can be a list of any number of node declarations and assertions. Node declarations create new variables, that can be used in assertions. Assertions are the actual requirements, and they consist of a Boolean expression that must evaluate to True for the requirement to be satisfied.

Assertions are, in the current version of the prototype, either equality or inequality constraints among Ref expressions. Ref expressions are references to

- Node types, written as `type(Ref)`, which evaluates to the type of the node referenced by `Ref`,
- Accessors, which can refer to variable names or constant objects in the DOML blueprint, as well as fields thereof (obtained by concatenating `->` and the field name to the object identifier), or
- Integer or string constants.

We further explain the syntax by means of the following simple example of a user-defined property:

```
node $x;
assert(type($x) == doml.nodes.SQLDatabaseService);
assert($x->password == "p4ssw0rd");
```

First, a node variable named "x" is declared. Then, the first assertion says that x can only be a node representing a SQL database by constraining its type. Finally, the last assertion checks whether the node contains a property named "password" having a specific constant value. This is a property that nodes should **not** have: if a node satisfying it is found, it means the service model contains an error.

The tool translates this code in the corresponding SMT solver assertions. When the solver is run, it tries to bind variable x with a node satisfying the assertions. If it fails, then no node satisfies the assertion, and the service model is correct. Otherwise, the node bound to x contains the defect described by the user-supplied property, and it can be returned to the user to help them debug the service model.

#### *2.1.1.4 Fitting into overall PIACERE Architecture*

Figure 2 shows a diagram of the overall PIACERE project architecture.

The DOML mode checker is part of KR4 in WP4. The purpose of this WP is to ensure the safety, security, and correctness of DOML models. The model checking tool allows for checking the model against arbitrary user-defined requirements.

The tools described in this section are not intended to be directly included in the PIACERE infrastructure but have only been developed as proof-of-concepts to guide further developments of KR4.

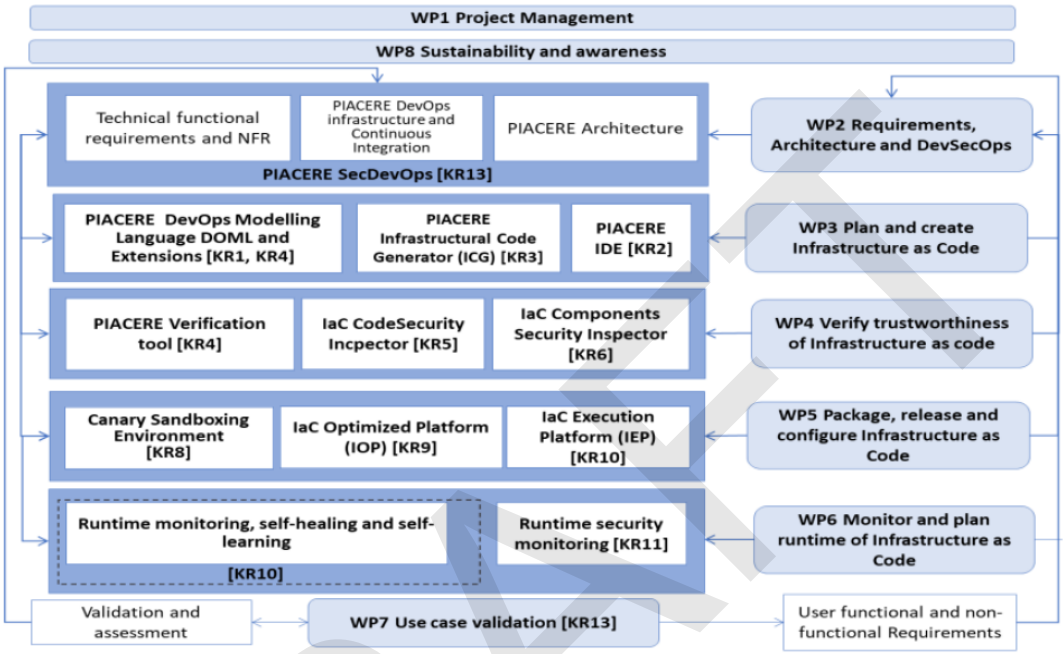


*Figure 2. PIACERE WPs and KRs, with their relationships.*

## 2.1.2 Technical description

In this section, we focus on the technical details of the prototype implementation.

### *2.1.2.1 Prototype architecture*

The architecture of the tool is fairly simple. The prototype consists of:

- A library that reads YAML-based TOSCA service models;
- A stand-alone solver (either a Prolog implementation or an SMT solver);
- The main program, which loads the TOSCA service models through the parsing library, parses the user-supplied requirements, and invokes the solver.

### *2.1.2.2 Technical specifications*

We used the Python programming language, version 3.9, to develop the prototype.

The library used to parse YAML-based TOSCA service templates is called “tosca-parser” and has been developed in the OpenStack project (https://opendev.org/openstack/tosca-parser). The latest publicly available version of the library presents some bugs that we have to solve for our prototype to work. The fixed version of the library can be found at https://github.com/mikidep/tosca-parser

The Prolog-based version of the tool uses the PyYAML library (https://pyyaml.org/) for parsing user-supplied requirements written in the YAML-based domain-specific language.

We use SWI-Prolog (https://www.swi-prolog.org/) as the Prolog implementation for our tool. This choice is based its being one of the most widely used and better supported Prolog implementations. The Python-based core of the prototype uses the library PySwip (https://github.com/yuce/pyswip) to interact with SWI-Prolog.

The SMT-solver-based version of the tools, instead, uses the Python library textX (http://textx.github.io/textX/stable/) to parse the domain-specific language used for user-defined requirements.

The SMT solver queried by the tool is the Z3 theorem prover (https://github.com/Z3Prover/z3), developed by Microsoft Research. Z3 offers a Python wrapper which we use to interact with it.

## 2.2 Delivery and usage

### 2.2.1 Package information

We list and briefly describe files contained in the prototype package below.

#### *2.2.1.1 Prolog-based Tool*

- check2swipl.py: source code of the parser for user-defined requirements;
- checks.yaml: file containing example user-defined requirements;
- doml_tosca.yaml: example TOSCA service template;
- poc.py: main part of the tool;
- poetry.lock: file required by the packaging system;
- predicates.pl: file containing Prolog utility code;
- pyproject.toml: settings file for the packaging system;
- README.md: brief user guide for the tool;
- tosca2swipl.py: source code of the module that translates TOSCA service templates into Prolog knowledge bases.

#### *2.2.1.2 SMT-Solver-based Tool*

- doml_tosca.yaml: example TOSCA service template;
- doml_tosca_z3: main module directory
  - __main__.py: an incomplete entrypoint script;
  - tosca_utils.py: convenience functions processing TOSCA entities;
  - vtlang_model.py: source code of the module translating the user-defined specification into Z3 constants and assertions;
  - z3toscamodel.py: source code of the module translating TOSCA service templates into Z3 constants, functions and assertions;
  - z3_utils.py: convenience methods generating Z3 entities;
- poetry.lock: file required by the packaging system;
- pyproject.toml: settings file for the packaging system;
- README.md: brief user guide for the tool;
- setup.cfg: settings file for the Python linter;
- vtlang.tx: grammar description for the specification language;
- vttest.vt: example specification.

### 2.2.2 Installation instructions

First, clone the repository containing the tool (cf. Section 2.2.5) and check out the appropriate branch: master for the Prolog-based prototype, and Z3 for the SMT-solver-based one.

If you want to install the Prolog-based prototype, SWI-Prolog must be installed separately. Please follow the installation instructions available at https://www.swi-prolog.org/ .

Both versions of the prototype require Python 3.9 to be installed on the system (https://www.python.org/downloads/ ), and use the Poetry package manager, which must be installed on the system (https://python-poetry.org/docs/#installation).

Once the above software programs have been installed, run the following command to automatically install all required Python packages:

```
$ poetry install
```

### 2.2.3 User Manual

#### *2.2.3.1 Prolog-based Tool*

The Prolog-based proof-of-concept automatically loads requirements to be checked from the file "checks.yaml". To run the project on the included TOSCA model, run:

```
$ poetry run python poc.py doml_tosca.yaml
```

To further query the generated Prolog model, running

```
$ poetry run python -i poc.py doml_tosca.yaml
```

opens a Python interpreter, where the `prolog` object can be queried using its PySwip interface:

```
$ poetry run python -i poc.py doml_tosca.yaml
>>> results = prolog.query("node(X, T, _, _, _),
extends_type(T, 'tosca.nodes.SoftwareComponent')")
>>> for r in results:
...     print(r)
...
{'X': 'webapp', 'T': 'myapp.nodes.WebApp'}
{'X': 'redis', 'T': 'doml.nodes.Redis'}
>>>
```

#### *2.2.3.2 SMT-Solver-based Tool*

The current version of the tool offers a Python-based interactive interface. To run and inspect the results, run:

```
$ poetry run python -i -m doml_tosca_z3 doml_tosca.yaml
vttest.vt
>>> vtmodel.solver.check()
sat
>>> vtmodel.solver.model()
[x = some_db,
node_prop = [(some_webapp, db_password) ->
              func(get_input,
                   cons(str(ss_5_db_password), nil)),
              (some_webapp, db_user) ->
              func(get_input, cons(str(ss_6_db_user), nil)),
              (some_db, user) ->
              func(get_input, cons(str(ss_6_db_user), nil)),
              (some_db, password) -> str(ss_2_p4ssw0rd),
              else -> none],
```

```
node_type = [some_webapp -> some_myapp.nodes.WebApp,
             some_redis -> some_doml.nodes.Redis,
             some_db -> some_doml.nodes.SQLDatabaseService,
             some_load_balancer ->
             some_tosca.nodes.LoadBalancer,
             none -> none,
             else -> some_tosca.nodes.Compute]]
```

### 2.2.4 Licensing information

All software artifacts described in this section are available under the Apache 2.0 license (https://www.apache.org/licenses/LICENSE-2.0).

### 2.2.5 Download

The source code of the Prolog-based prototype can be found at the following link:

https://github.com/mikidep/piacere-formal-verification/tree/master

The source code of the SMT-solver-based prototype can be found at the following link:

https://github.com/mikidep/piacere-formal-verification/tree/z3

# 3 DOML Model Checker Prototype

## 3.1 Implementation

In this section, we describe the implementation of the DOML model checker prototype and its future development plans.

### 3.1.1 Functional description

This prototype goes one step further towards the integration with the PIACERE framework, by accepting input service models in one of the early proposals for the PIACERE DOML language. In particular, the DOML version we refer to is the one detailed in [3]. Throughout the rest of this section, we refer to this DOML proposal as simply "DOML". This tool's frontend will be incrementally adapted to the newest version of the DOML as defined by WP3, as soon as it is sufficiently detailed.

Thus, the frontend of our tool contains a parser for DOML blueprints. The original parser for this version of the DOML, developed as part of KR1 in the context of WP3, has been developed by using the Xtext framework, which makes it hard to reuse it to develop our Python prototype, because the intermediate representation offered by the existing parser is not general enough. Thus, we had to implement a new Python parser.

For this tool, we chose the Z3 SMT solver as the backend. This choice has been based on the experiments we conducted with the TOSCA model checker prototypes we describe in Section 2, which highlighted that an SMT solver allows for a more user-friendly domain-specific language for expressing user-supplied specifications.

Thus, this prototype will check user-supplied requirements expressed in a domain-specific language similar to the one described in Section 2.1.1.3.2. In the final version, this language will be part of DOML, and requirements will be stated in DOML blueprints directly.

Moreover, this prototype also checks several standard consistency issues, such as unresolved references to non-existing components, and circular dependencies.

Note that, however, this prototype is still at an early stage of development. Thus, only the custom DOML parser and the consistency checking mechanisms have been currently implemented. The interface with the SMT solver, which is the core of the model checking tool, will be implemented in a future release.

This prototype, which is part of WP4, is intended to be integrated into the general PIACERE framework, and its IDE in particular (WP3), in order to support a workflow where IaC artifacts can be verified after being defined through the other parts of WP3.

#### *3.1.1.1 Overview of the approach*

In Figure 3, we show an UML sequence diagram of a typical workflow intended for the model checking tool.

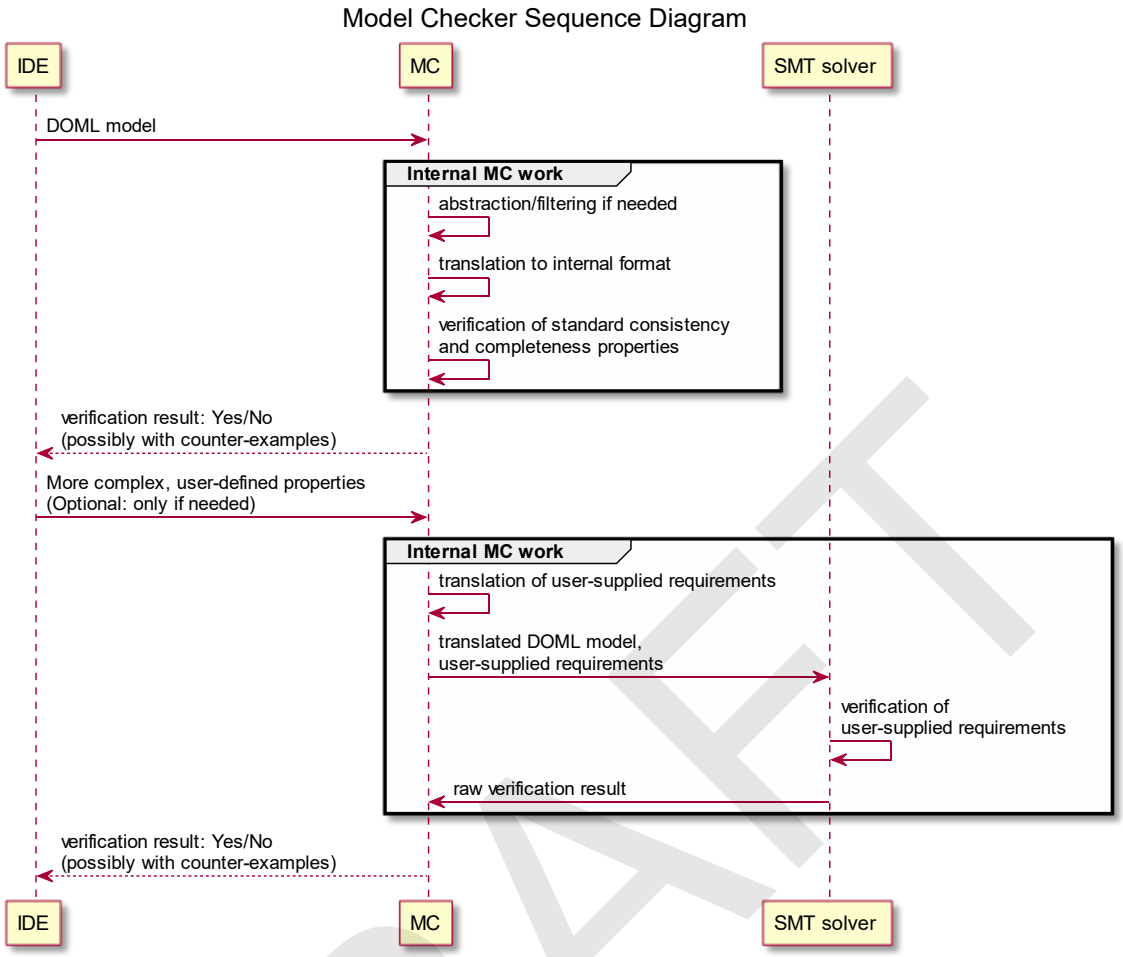


*Figure 3. Sequence diagram of the DOML model checker.*

The user invokes the model checker directly from the IDE. The IDE gives the TOSCA service template in input to the model checking tool. The tool loads it in an internal in-memory representation, and then further translates it to an input representation for the backend.

During this process, some standard consistency requirements are checked on-the-fly, and results of this check are returned to the user.

The user may then specify more complex properties through a domain-specific language. Such requirements are parsed by the tool and translated to a representation readable by the backend. The backend solver is then invoked, and its results are reported to the user.

The only part of this tool that has been currently implemented is the DOML parser, and the checkers for standard consistency and completeness properties. The interface with the model checking backend and the integration with the PIACERE IDE have not been implemented yet and are left as future work.

Overall, the parsing and standard requirements checking currently supported by the tool is performed in four phases, shown in Figure 4:

1) The user-supplied DOML blueprint is parsed and loaded, and all RMDF files are gathered and loaded, but references between components are left unresolved.
2) The tool searches the model for circular dependencies, and type covariance issues are detected.

3) References among different components are resolved, highlighting undefined symbols and references.
4) Correctness with respect to node and data types is checked.

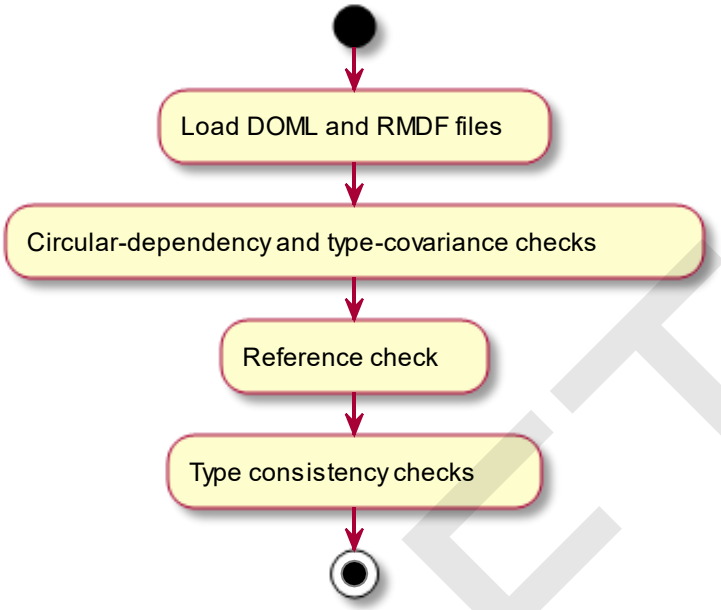


*Figure 4. DOML parsing and standard correctness-checking activity diagram.*

### *3.1.1.2 Checking of Standard Properties of DOML*

#### 3.1.1.2.1 Translation of the DOML model into an Internal Format

The tool parses and loads the user-supplied DOML model and all related RMDF (Resource Model Definition) files. RMDF files are auxiliary files required by DOML blueprints according to [3]. They describe resources, such as infrastructure nodes and node types, required by the deployment. The actual DOML blueprints describe deployments by referencing the nodes defined in RMDF files.

First, the tool searches the directory tree where the user-supplied model is stored for available RMDF files. Then, such files and the model are parsed.

Since the DOML version we target is based on a YAML syntax, the input files are first loaded by means of a YAML parsing library. An initial consistency check makes sure that all DOML elements only have fields and properties that are allowed for them.

An in-memory representation of the model is then generated from this library's output, based on a class hierarchy that reflects the semantics of DOML blueprints. In this phase, references among different elements (such as nodes, types, etc.) in the model are left unresolved.

#### 3.1.1.2.2 Validation and Verification of Completeness and Consistency

In this phase, some important internal-validity and consistency properties of the model are checked.

First, graph-theoretic algorithms are used to explore the topology generated by the model, and all circular dependencies are detected. By circular dependency we mean, for example, a situation where a node or data type A is a sub-type of a type B, and B is a sub-type of A. Note

that such cycles may involve more than two types, leading to the need for graph theoretic algorithms.

Type covariance issues are also detected in this phase. By type covariance we mean that all types should only override fields defined in their super-types. I.e., if a type A overrides a field F from a type B, but A is not a sub-type of B, then the model is wrong with respect to type covariance.

Then, references between all elements of the model are resolved. Whenever a reference to a non-existing object is found, an exception is raised.

Finally, more type-consistency checks are performed. The tool checks whether nodes have all properties required by their types, and that such properties have the right type and cardinality.

#### 3.1.1.2.3 Verification Result and Counterexamples

Whenever one of the anomalies described in Section 3.1.1.2.2 is found, the tool raises an exception. The exception object contains all the information required for the user to identify and debug the issue. Thus, the main module of the tool can catch such exceptions, and present them to the user, together with the information about the offending part of the model.

### *3.1.1.3 Checking of User-defined Properties*

This part of the tool's feature set has not been implemented yet. Here we only give an overview of the planned features.

#### 3.1.1.3.1 Expressing User-defined Properties in DOML

The tool will include support for a domain-specific language similar to the one presented in Section 2.1.1.3.2, and specifically targeted at the input language of the Z3 SMT solver. In particular, the language will allow for specifying assertions covering all elements of the DOML model and their properties and fields. The language will also allow for specifying variables that can be bound to elements of the model that violate the user-supplied properties to be checked.

#### 3.1.1.3.2 Verification of User-defined Properties

The user-supplied properties will be translated in a format accepted by the SMT solver backend, together with the DOML model. The solver will be then invoked, and the constraint-solution process will either output parts of the model that fail to satisfy the requirements or prove that the model satisfies all such requirements.

### *3.1.1.4 Fitting into overall PIACERE Architecture*

Figure 2 shows the overall PIACERE architecture as a diagram.

The DOML mode checker is part of KR4 in the context of WP4. The purpose of this WP is to ensure the safety, security, and correctness of DOML models. The model checking tool, in particular, allows for checking the model against arbitrary user-defined requirements.

This tool is supposed to interact with the PIACERE IDE (KR2, WP3) in particular, in order to provide users with an easy-to-use interface for its verification activities. Moreover, the domain-specific language for specifying user-supplied requirements will be part of the DOML extensions (KR4 from WP3).

## 3.1.2 Technical description

In this section, we describe the technical details concerning the implementation of the DOML model checker prototype.

#### 3.1.2.1 Prototype architecture

The prototype contains the following components:

1) A library the parses YAML-based DOML and RMDF files.
2) A module that translates the output of the parser into an internal representation and performs various consistency checks.
3) A library that performs graph-theoretic checks on the model's topology.
4) A module that parses the input language for user-supplied requirements.
5) A module that translates requirements and service templates into a representation readable by the SMT solver backend and invokes it.

Note that only components from 1) to 3) have been currently implemented.

#### 3.1.2.2 Technical specifications

The prototype has been coded in Python 3.9, for flexibility and coding speed of this language, as well as for the rich availability of libraries addressing many of the required features, including the easy interoperability with other components, such as the backend solver.

We use the PyYAML library (https://pyyaml.org/) for parsing RMDF and DOML files, as they have a YAML-based syntax.

Then, the Cerberus Python library (https://docs.python-cerberus.org/) is employed to validate the output of the YAML parsing against the syntax of DOML.

The NetworkX library (https://networkx.org/) is used for its graph-theoretic cycle-detection algorithms.

In future versions of the tool, the Z3 (https://github.com/Z3Prover/z3) SMT solver will be used.

## 3.2 Delivery and usage

### 3.2.1 Package information

We list and briefly describe files contained in the prototype package below.

#### 3.2.1.1 DOML Parser

- doml_parser: main module directory
  - model: submodule containing classes representing DOML entities
    - _unresolved: private submodule containing classes representing DOML entities whose internal references have not been resolved yet
      - resolver.py: utility class to perform reference resolution of the whole model;
      - unres_checker.py: utility class to check properties of the unresolved model, such as absence of circular dependencies and type covariance in overridden entities;
      - Other files: classes representing unresolved DOML entities;
    - Other files: classes representing DOML entities;
  - __init__.py: main module file;
  - __main__.py: module execution entrypoint;
  - _yaml_structure_schemas.py: source code containing structural schemas that RMDF and DOML documents must respect in order to be parsed;
  - errors.py: exceptions thrown by the parser in case that the parsed models are structurally invalid or incorrect;

- tests: pytest test suite for the parser
  - examples: example DOML and RMDF models, taken from [3] and adapted to work with the parser;
  - demo: models used to demonstrate tool functionality in detecting incorrect models;
- .gitignore: Git file to ignore certain files for version control;
- load.py: convenience script to load and inspect all DOML example models during development;
- poetry.lock: file required by the packaging system;
- pyproject.toml: settings file for the packaging system;
- README.md: brief user guide for the tool.

### 3.2.2 Installation instructions

First, clone the repository containing the tool (cf. Section 0) and check out the *main* branch.

The prototype requires the Python 3.9 runtime to be already installed on the system (https://www.python.org/downloads/ ), and uses the Poetry package manager, which must also be installed on the system (https://python-poetry.org/docs/#installation).

Once the above software programs have been installed, run the following command to automatically install all required Python packages:

```
$ poetry install
```

### 3.2.3 User Manual

#### *3.2.3.1 DOML Parser*

To check a DOML file, run

```
poetry run python -m doml_parser check <doml-path>
```

To explore the generated DOML model, use the `load_doml_from_path` in the `doml_parser` module, as exemplified in `doml_parser/__main__.py`. The method parses and checks the DOML model and the RMDF's in the current directory, and returns a usable `doml_parser.model.doml_model.DOMLModel` object.

### 3.2.4 Licensing information

All software artifacts described in this section are available under the Apache 2.0 license (https://www.apache.org/licenses/LICENSE-2.0).

### 3.2.5 Download

The source code of the tool is publicly available on GitHub:

https://github.com/mikidep/doml_parser

# 4 Conclusions

In this deliverable, we describe the state of the implementation of KR5 from W4. The products obtained so far consist of two prototypes for the PIACERE model checking and verification tool.

The first one targets the TOSCA IaC modelling language and has been developed with the purpose of exploring suitable techniques for model checking IaC. In particular, a model checking tool has been implemented using a Prolog implementation as its backend, and another one using the Z3 SMT solver.

The second prototype, which is still at an early stage of development, targets the DOML language as defined in [3]. It features a DOML parser, and automatically checks several important consistency and correctness properties of DOML models. Its extension to checking user-supplied complex properties by means of the Z3 SMT solver and its integration into the PIACERE IDE (KR2 of WP3) are planned. Moreover, it will be adapted to target the new versions of the DOML language (cf. Deliverable 3.1).

All developed prototypes are open source and publicly available.

# 5 References

[1] J. Crinnion, Evolutionary Systems Development, a practical guide to the use of prototyping within a structured systems methodology, New York: Plenum Press, 1991.

[2] T. Grimm, «The Human Condition: A Justification for Rapid Prototyping,» *Time-Compression Technologies,* vol. 3, nº 3, 1998 May 1998.

[3] 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.