Deliverable D4.3

Infrastructural model and code verification – v3

Editor(s):	Andrea Franchini Matteo Pradella
Responsible Partner:	Politecnico di Milano/Polimi
Status-Version:	Final v1.0
Date:	30.05.2023
Distribution level (CO, PU):	PU

Project Number:	101000162
Project Title:	PIACERE

Title of Deliverable:	D4.2 - Infrastructural model and code verification – v3
Due Date of Delivery to the EC	31.05.2023

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

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 used via its REST API or CLI, and has been integrated with the PIACERE IDE (KR2). 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	01.05.2023	First draft version	Andrea Franchini (Polimi)
v0.2	07.05.2023	KR 5 overview	Andrea Franchini (Polimi)
v0.3	13.05.2023	Add Bibliography	Andrea Franchini (Polimi)
v0.4	15.05.2023	Add Technical documentation and diagrams – *Ready for review*	Andrea Franchini (Polimi)
v0.5	24.05.2023	Internal review	Adrian Noguero (Go4IT)
v0.6	26.05.2023	Final Version, comments addressed. Ready for final quality check.	Andrea Franchini (Polimi)
v1.0	29.05.2023	Quality review check done. Ready for submission.	Juncal Alonso (TECNALIA)

Table of contents

List of tables

List of figures

Terms and abbreviations

CSP	Cloud Service Provider
DevOps	Development and Operation
DoA	Description of Action
EC	European Commission
GA	Grant Agreement to the project
IaC	Infrastructure as Code
IEP	IaC execution platform
IOP	IaC Optimization
KPI	Key Performance Indicator
SW	Software
DOML	DevOps Modelling Language
DOMLR	DOML Requirements -- the DSL in DOML to describe custom user requirements
IM	Intermediate Model
IMC	Intermediate Model Checker

Executive Summary

This is the final deliverable, superseding D4.2 [1] and D4.1 [2] . It describes the work done in T4.1 in the third year to the development of the DOML Model Checker (DMC – KR5), 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 and focuses 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 latest version presented in this document extends the DMC functionalities by adding the ability to write custom requirements in the DOML model using a Domain Specific Language.

Future updates will focus on support of new DOML versions, Quality of Life improvements, and updates to the built-in requirements to better fit the project direction.

1 Introduction

The present deliverable describes 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). 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.

This deliverable is the final one in a series of three deliverables. It provides updates to the previous deliverable D4.2 and D4.1, describing the activities concerning KR5 performed throughout the second year and third year of the project.

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.

1.1 About this deliverable

In this deliverable we describe the latest additions to the DMC that have been developed, as well as documenting the relevant changes, since the version described in D4.2. This document presents the final architecture of the DMC, as well as detailed description of its workflows.

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, and one based on the Z3 Theorem Prover**,** 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, as described in D4.2.

For the shake of completeness, this document includes an updated version of the implementation and delivery and usage of the component. These sections have suffered small challenges from the previous deliverables.

1.2 Document structure

The document as follows:

- 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 summarizes the findings emerged from the use of the DMC.
- Section 5 draws conclusions.
- Section 6 contains the bibliography.

2 KR 5-VT overview

KR 5-Verification Tools (VT) focuses on the development of a set of verification tools, which includes a model checker able to verify the structural consistency and correctness of DOML models.

Since D4.2, the model checker was already functional and able to verify the structural consistency and correctness properties of an input DOML model, using the Z3 SMT solver.

Thereafter, it has been extended to support a Domain Specific Language (DSL) to allow users to write their own requirements in a fragment of first-order logic that closely resembles the same structure used by the built-in requirements through the Z3 SMT solver API, and at the same time, keeps the same naming convention of DOML to produce easily readable expressions.

Moreover, to exploit the properties of the Z3 SMT solver, an experimental synthesis feature has been developed, which is able to produce a complete model starting from an incomplete one.

2.1 Changes in v3

The newer versions of the model checker (v2.0 and newer) add support for the following features:

- Support for DOML 2.2 (REQ95)
- DSL (called DOMLR) allowing users to write custom requirements in the DOML file (REQ104)
- Command Line Interface to allow standalone use of the model-checker with extended features
- Updated built-in requirements to accurately cover certain configurations.
- Preliminary support for a Synthesis Module, that allows to generate correct models starting from incomplete ones.

2.2 Functional description and requirements coverage

Table 1. Functional requirements[1] addressed by the DMC

Req ID	Description	Status	Requirement Coverage at M30
REQ95	VT tools (model checker) must be able read DOML language (ex REQ56)	MUST HAVE	Done
REQ103	Verification Tool (model checker) must verify the structural consistency of the DOML models	MUST HAVE	Done
REQ104	Verification Tool (model checker) must verify the correctness of DOML models, with respect to some	MUST HAVE	Done

[1] Functional Requirements defined in D2.2 [12].

	correctness properties provided in DOML		
REQ105	Verification Tool (model checker) must verify the completeness of DOML models	MUST HAVE	Done

The table 1 above can be summarized as follows:

- The model checker can interface with the IDE leveraging the DOMLX format, which is an XML-based representation of the DOML model and is generated by the IDE in a step preceding the actual model checking. This format keeps all the meaningful relationships between infrastructural elements. New DOML versions require updating the DMC internal representation to account for changes in the DOML metamodel; this has been done for DOML version 2.2 (REQ95)
- The model checker can determine whether a model is:
    - *Structurally consistent*: there are no conflicts, contradictions that could prevent the ICG from generating code, or lead to ambiguous situations. (REQ103)
    - *Complete*: the model contains all the required elements needed for a successful deployment. (REQ105)
    - *Correct* with respect to built-in and user-defined requirements. User-defined requirements can be specified in the proper section of DOML using a DSL. (REQ104)

REQ95: Since D4.2, the DMC could already parse correctly DOMLX files received from the IDE representing the DOML model. During year 3 we focused on extending support to newer DOML versions, and fixing mismatches between DOML and the DMC internal representation. Since there are now multiple DOML versions that could be passed to the same instance of the DMC, we introduced automatic version detection of the DOML version in use by the model. Users can specify in the DOML the version they want to use. Alternatively, the DMC can be overridden to use a specific DOML version.

REQ103: Most of the requirement functionality was already achieved, we focused to fixing small issues and provide relevant error messages when the DOML model is. Certain requirements have been made optional because most obvious structural errors are already identified in the IDE editor, in order to save time during the model checking.

REQ104: The model checker can be extended by the users in the DOML model through the *functional requirements* field, that allows them to specify custom requirements. These requirements are written in a DSL that allows users to write complex rules through first-order logic, while prioritizing legibility.

REQ105: The built-in requirements of the model checker ensures that most critical relationships between infrastructural elements are present and valid. Requirements are updated accordingly to feedback from design sessions. An optional tool has been developed to check model compatibility with selected Cloud Service Providers; feedback from this tool is presented as a compatibility matrix.

2.2.1 Built-in requirements

The DMC verifies DOML models against a collection of requirements devised to highlight the most common mistakes made by users when specifying cloud deployments. Here we list and describe such requirements.

Table 2. Requirements considered 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 isolated.
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 in 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.
All network interfaces belong to a security group.	Makes sure all network interfaces have been configured to belong to a security group. This way, the user will be reminded to configure adequate rules for each network.
All external SaaS can be reached only through a secure connection.	Makes sure that an HTTPS rule is enforced for a Network Interface of a Software Component that interfaces with a SaaS.

2.2.2 DOMLR – DSL for custom user requirements

The new addition of DOMLR (DOML Requirements) is a Domain Specific Language (DSL) that can be used inside DOML under the *functional requirements* block as follows:

~~~
functional_requirements {
    req_group_1 ```
    # Your DOMLR here.
    ```;
    req_group_2 ```
    # Other set of requirements
    ```;
}
~~~

The 'req_group_1', 'req_group_2' keys are for self-documentation, but do not serve a purpose in the DOMLR.

A requirement has the following form (here targeting DOML 2.2):

```
+ "All Virtual Machines have a Interface and at least 512MB of RAM"
forall vm (
    vm is class abstract.VirtualMachine
    implies
    exists iface (
        vm has abstract.ComputingNode.ifaces iface
        and
        vm has abstract.ComputingNode.memory_mb >= 512
    )
)
error: "A vm lacks an associated interface or has less than 512 MB of
RAM"
```

The above requirement checks that for all virtual machine elements specified in DOML, they all have at least one Network Interface and at least 512 MB of memory. The title is highlighted in blue, and it is for self-documentation. The logical expression, written in first-order logic, is highlighted in green, and evaluates either to true or false. The error message is displayed when the requirement is not satisfied, and it is highlighted in red.

Before the title in blue, there is a '+' or '-' sign, that indicates whether the requirement will be evaluated in positive or negative form. In **positive** form, when the expression is evaluated to 'true', the requirement will be **satisfied**. In **negative** form, when the expression is evaluated to 'true', the requirement will be **not satisfied**. In other words, in positive form we check that a property we want in your model, while in negative form you check that an undesired condition exists.

Example of positive form requirement

```
+ "All VMs have at least one interface"
vm is class abstract.VirtualMachine
Implies
exists iface (
     vm has abstract.ComputingNode.ifaces iface
)
error: "A VM has no associated interface."
```

Example of negative form requirement

```
- "All VMs have at least one interface"
vm is class abstract.VirtualMachine
and
not exists iface (
     vm has abstract.ComputingNode.ifaces iface
)
error: "VM {vm} has no associated interface."
```

The two code snippets above describe the same requirement, the only difference being the positive and negative form. While harder to write, the negative form provide advantages in terms of performance and in error handling, since the '{vm}' can be replaced by the name of the actual element that causes the issue.

Regarding the syntax, it resembles the first order logic notation. Note that the language is not indent based, although for better legibility is indented. Words between '<' '>' are placeholders.

- Universal quantifier: *forall <variables, comma ',' separated> (<expression>)*
- Existential quantifier: *exists <variables, comma ',' separated> (<expression>)*
- Connectives (using infix notation – e.g: *a and b*):
  - *and*
  - *or*
  - *implies* -- equivalent to: *(not A) or B*
  - *iff* -- "if and only if", equivalent to: *(A implies B) and (B implies A)*
- Negation: *not <expression>*
- Parenthesis '(', ')' are optional, but can improve legibility and override precedence. If you are in doubt about precedence, use parenthesis.

The precedence of these operators is the following:

`exists/forall > not > or > and > implies > iff`

These keywords are used to connect together multiple expressions. There is no need to instantiate variables. Relationship and class/element IDs are extracted automatically from the DOML metamodel names, and have the form <layer>.<element> for classes and <layer>.<element>.<relationship> for relationships. They are available for quick reference here:

https://piacere-model-checker.readthedocs.io/en/latest/reference_index.html

In the example '*vm is class abstract.VirtualMachine'* is an expression of the kind '*<var> is class <class ID>*', which specifices that the *'vm'* variable is a virtual machine.

The other expression, '*vm has abstract.ComputingNode.ifaces iface*', of the kind '*<var> has <relationship ID> <var>*' specifies a relationship '*abstract.ComputingNode.ifaces*' between the variables *'vm'* and *'iface'*. We don't need to specify that *'iface'* is an interface since it is implied in the relationship.

Another expression is '*vm has abstract.ComputingNode.memory_mb >= 512*', which is of the form '*<vm> has <relationship ID> <comparison operator> <value>',* which specify that 'vm' should have the attribute 'memory_mb' greater or equal to 512. For attributes that are of type String or Boolean, only '==' and '!=' are available, with '<=', '<', '>=', '>' raising an error.

DOMLR supports three types: Integers, Strings (enclosed by double quotes) and Booleans (':true', ':false')

To confront two element attributes, '*vm1 has abstract.ComputingNode.memory_mb >= vm2 has abstract.ComputingNode.memory_mb'* is a variation of the above syntax that compares the attribute values of two elements.

2.3 Main innovations

Infrastructure-as-Code (IaC) offers several advantages that make it a powerful approach for managing and provisioning infrastructure resources. By expressing infrastructure configurations as code, IaC eliminates manual processes and reduces human error, leading to greater reliability and faster deployment cycles. IaC enables developers to define infrastructure components such as servers, networks, and storage in code, making it easier to provision, configure, and manage infrastructure at scale. However, as the complexity of infrastructure-as-code configurations increases, the need for verification techniques becomes crucial. Formal verification provides a systematic and rigorous approach to ensure that the model accurately represents the intended infrastructure state. It helps identify potential issues and ensures compliance with desired specifications, security standards, and best practices. By applying formal verification techniques such as model checking, organizations can detect and prevent configuration errors, resource conflicts, or security vulnerabilities before deployment. This reduces the risk of infrastructure failures, enhances system resilience, and ultimately improves the overall quality and reliability of the infrastructure-as-code implementation.

The DOML model checker is a powerful tool that can identify issues before deployment and provide the user with detailed errors and possibly with a course of action to solve those issues. Using the Z3 SMT solver it builds an intermediate representation of the model and checks that the relationships specified in the requirements between elements are respected.

A DSL integrated within the DOML (called DOMLR, "DOML Requirements") allow users to write their own custom requirements to check the correctness of specific situations that might not be covered adequately by the built-in requirements. This language resembles plain English and is a fragment of first-order logic that features the same naming convention of DOML. Elements, their attributes and their relationships can be used to specify, for example, desired configurations.

The output of the model checker is a report detailing whether all requirements are satisfied. In case some requirements are not satisfied, a detailed error explains (when possible) which element is causing the problem, and which actions can be performed to fix it.

A tool to check the DOML model compatibility with selected Cloud Service Providers has been developed and is currently being tested. This tool provides compatibility matrices highlighting which configurations are supported by specific vendors. For the time being, it focuses on *architecture, OS* of Virtual Machines and valid *KeyPairs*. When ready, it will be integrated into the DMC and will be invoked by adding a special directive in the DOMLR section of the DOML (*functional requirements*), or through its own RESTful endpoint.

An experimental synthesis module has been developed to investigate whether the ability of the Z3 SMT solver to produce new correct models can be applied to DOML. The results are promising but at the moment the lack of a way to integrate the results back in the IDE and frequent DOML updates led to a pause of its development.

A CLI has been developed to allow future integration of the tool outside the IDE, as well as to let user access features that are not currently included in the IDE, such as the synthesis module.

3 Overview of preliminary experiments

In most cases the model checker proved useful in catching misconfigurations through its build-in requirements, that cover most common scenarios. During design sessions, certain requirements have been extended to accommodate unforeseen scenarios.

The DSL for custom requirements (DOMLR), despite having a simple syntax, presents an inherent difficulty in using it efficiently when the user does not know first-order logic. It is feasible for a user to modify an existing requirement to cover similar situations, with the help of the manual for DOMLR.

The CLI interface is straightforward and intuitive, with detailed documentation

The experimental synthesis module performs well in specific unit tests but requires further testing and optimization to be used with real models. As a future integration, starting from the output, it is possible to generate a new DOMLX, and utilize the DOMLX to DOML service implemented as part of T3.1 to produce a synthesized DOML model. Alternatively, once DOML stabilizes its syntax, or should there be issues within the PyEcore library, it might be possible to directly generate DOML snippets that, for example, the user can copy-and-paste into the IDE editor.

4 Lessons learnt and outlook to the future

Model checking proved useful in identifying structural problems before further proceeding with other design steps. The verification times are short, usually a few seconds, but not immediate, but the tool is intended to be run as a side task. DOMLR did not receive much attention but remains a key part in extending the DMC to specific user needs.

In the next phase of the project, we are looking into adding new requirements according to the needs of the Use Case providers, and possible new versions of DOML.

Additional planned features include:

- Directives in DOMLR to exclude or include built-in requirements, or additional checks from the tool, for example the CSP compatibility tool.
- Better styled error messages, in particular the option to provide an HTML output. Either in the IDE or through the web server that already serves the DMC API.
- Integrate synthesis feature, when DOML stabilizes its syntax and further optimization is performed.

5 Conclusions

In this deliverable, we presented the work carried out by PoliMi on the KR5 within WP4. In the D4.1 we only presented proof-of-concept prototypes that had the only purpose of exploring possible solutions by experimenting with rapid prototyping, in D4.2 we presented a functioning version of the DMC, capable of reading and verifying DOML models in the intermediate format offered by the EMF framework, in D4.3 we further refined the DMC to support user custom requirements and explored synthesis of models.

The DMC is open-source and can be run both directly from source or through a Docker image, which facilitates its deployment. The main way of using the DMC is through the PIACERE IDE which uses the REST API. However, a CLI has been developed to let future users integrate the DMC in alternative workflows.

This version of the DMC fulfils the requirements for KR5. There is still room for improvements: as DOML evolves, the DMC will be updated to correctly validated the models as thoroughly as possible. Moreover, we're exploring the feasibility to add model synthesis features, to provide users with suggested fixes for errors in their DOML models.

6 References

[1] PIACERE Consrotum, “D4.2 - Infrastrcutural Model Verification v2,” 2022.

[2] PIACERE Consortium, “D4.1 - Infrastrcutural Model Verification v2,” 2021.

[3] Microsoft Research, “Z3 SMT Solver,” [Online]. Available: https://www.microsoft.com/en-us/research/project/z3-3/. [Accessed 13 5 2023].

[4] “Lark Parser,” [Online]. Available: https://pypi.org/project/lark/.

[5] “OpenAPI,” [Online]. Available: https://www.openapis.org/. [Accessed 2023 5 13].

[6] «Connexion,» [En línea]. Available: https://pypi.org/project/connexion/.

[7] “PyEcore,” [Online]. Available: https://github.com/pyecore/pyecore.

[8] “SwaggerUI,” [Online]. Available: https://swagger.io/tools/swagger-ui/.

[9] “Sphinx,” [Online]. Available: https://www.sphinx-doc.org/en/master/index.html.

[10] “Docker,” [Online]. Available: https://www.docker.com/.

[11] “Uvicorn,” [Online]. Available: https://www.uvicorn.org/.

[12] PIACERE Consortium, “D2.2 PIACERE DevSecOps Framework Requirements specification, architecture and integration strategy – v2 1.1,” 2023.

APPENDIX: Implementation, delivery and usage

1 Implementation

1.1 Fitting into 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.

1.2 Technical description

In this section we will present the DMC, its behaviour and components in detail.

The DMC is overall isolated from other PIACERE tools, except for its dependency on DOMLX and, of course, the IDE to invoke it.

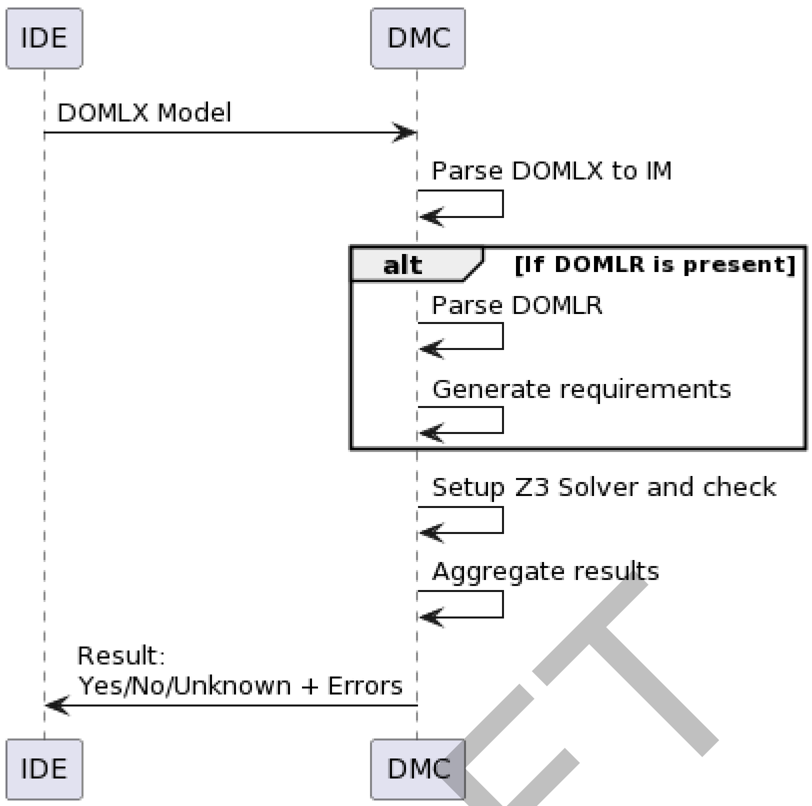

Figure 1. DMC external sequence diagram

In the above sequence diagram, the IDE pass a DOMLX representation of the DOML model to the DMC, which parses it to an internal model (IM) representation. If custom user requirements from the user written in DOMLR are present, they are parsed and their respective requirements are generated. The Z3 solver is instantiated, all requirements are evaluated singularly to produce a validation result:

- Yes: The model satisfies the requirement.
- No: The model does not satisfy the requirement, and an error message is produced.
- Unknown: The solver timed out so we cannot know for certain

All results are then aggregated and a response is sent to the IDE. If there's at least one requirement that is not satisfied, the response is a "No" with a list of all the unsatisfied requirements.

If the DMC runs into some issue (e.g. malformed DOML model) it raises an error, and communicates it to the IDE.

The IDE and the DMC communicate through a REST API, which implement a single endpoint:

- [POST] /modelcheck
  - Receives the DOMLX model
    - Optional query parameter 'version' of type 'string' to enforce a DOML version (e.g.: 'v2.2')

1.2.1.1 DOMLR

DOMLR is extracted from the DOML model as a string, or it can be passed as a separate file when using the CLI. It is then parsed with Lark [4], a Python parsing toolkit. The grammar used to parse the DSL is rather succinct and is reported here in the EBNF-like notation that Lark uses:

```
requirements    : requirement (requirement)*
requirement     : FLIP_EXPR req_name expression "error:" error_desc
req_name        : ESCAPED_STRING
error_desc      : ESCAPED_STRING

?expression     : iff_expr
?iff_expr       : (implies_expr "iff")? implies_expr
?implies_expr   : (and_expr "implies")? and_expr
?and_expr       : (or_expr "and")* or_expr
?or_expr        : (not_expr "or")* not_expr
?not_expr       : "not" not_expr -> negation
                | quantification
?quantification : "exists" bound_consts "(" expression ")" -> exists
                | "forall" bound_consts "(" expression ")" -> forall
                | "(" expression ")"
                | property
?property       : CONST "has" RELATIONSHIP CONST -> rel_assoc_expr

                | CONST "has" RELATIONSHIP COMPARISON_OP value -> rel_attr_value_expr
                | CONST "has" RELATIONSHIP COMPARISON_OP CONST RELATIONSHIP ->
rel_attr_elem_expr
                | const_or_class "is" const_or_class -> equality
                | const_or_class "is not" const_or_class -> inequality
bound_consts    : [CONST ("," CONST)*]
const_or_class  : "class" CLASS
                | CONST
value           : ESCAPED_STRING
                | NUMBER
                | BOOL

FLIP_EXPR       : "+"
                | "-"
COMPARISON_OP   : ">" | ">="
                | "<" | "<="
                | "==" | "!="
BOOL            : ":true"
                | ":false"
RELATIONSHIP: /[^\W\d_]+\.[^\W\d_]+\.([^\W\d]|_)+/
CLASS: /[^\W\d_]+\.[^\W\d_]+/
// Const must start with lowercase letter
CONST: (LCASE_LETTER) ("_"|LETTER|DIGIT)*
// Comment: python/sh style
COMMENT: /#[^\n]*/
```

Up-to-date documentation of DOMLR is available at https://piacere-model-checker.readthedocs.io/en/latest/writing-requirements.html.

1.2.1.2 Synthesis (Experimental)

Synthesis at the moment is an experimental feature that can produce complete (and correct) DOML models, starting from incomplete ones.

Synthesis works by adding unbound variables to the model, and letting the logic solver try to assign them a value. If the requirements are not satisfied, the solver adds more unbound variables and checks again, up to a set amount of tries. If a solution is found, it returns the new model. For example, synthesis can map components between the abstract and the concrete infrastructure layers, and can add the Network Interfaces that the user forgot to add.

It requires further optimization, as producing complex models can take a relatively significant amount of time (several minutes).

Because of the fast development of DOML, and the intricacies related to working with DOMLX parsing library (PyEcore), we decided to postpone the generation of an actual DOML/DOMLX model until DOML is finalized, as it might be actually more efficient to directly produce DOML code instead of DOMLX.

On the left, an example of a Virtual Machine for which a NetworkInterface and a Location have been synthesized. On the right, a more complex scenario in which a VM is connected through a NetworkInterface to a network along with a Storage Element connected to the same Network.

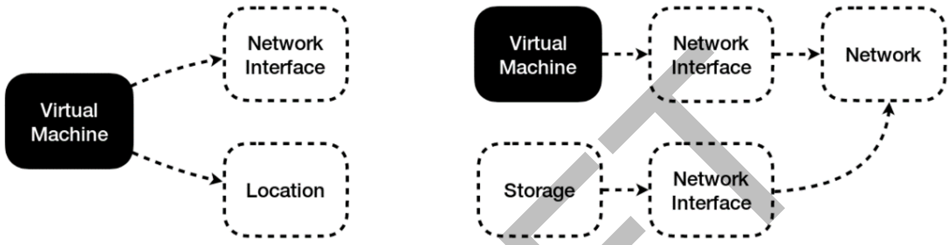

Figure 2. Examples used in the synthesis experimentation.

1.2.2 Prototype architecture

A high level description of the DMC internal architecture is shown in the following picture.

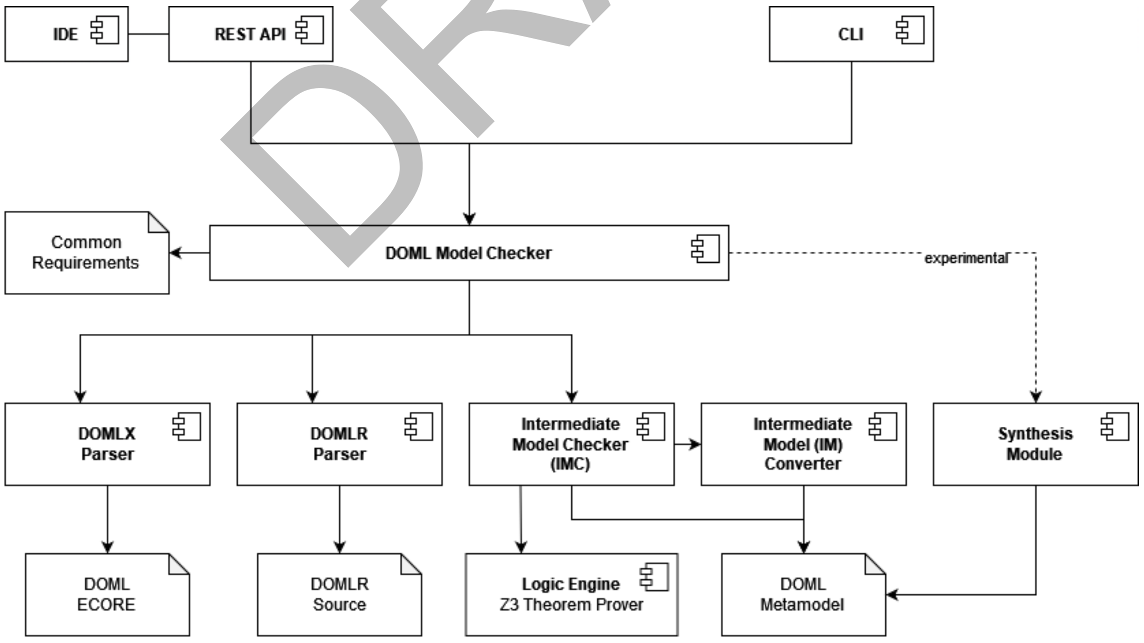

Figure 3. Component diagram of DMC

1.2.3 Components description

The DMC consist of a single module that can either be accessed through the IDE (leveraging the tool REST API) or independently via Command Line Interface (CLI). The features are split into separate submodules, each with its own purpose.

- The **REST API** exposes a single endpoint /modelcheck that receives a POST request with a required body content of type 'application/xml'. It uses the OpenAPI standard, and SwaggerUI to produce the documentation.
- A **CLI** alternatively can be used to run the DMC independently from the IDE. The user must be able to convert the DOML model to DOMLX, as the tool does not include the converter itself. The CLI interface also exposes additional features and parameters, some of which are configurable through environment variables (such as the number of CPU cores to use, or forcing a specific DOML version)
- The input DOMLX model is parsed by the **DOMLX Parser**, using the PyEcore library and the DOML Ecore specification of classes, attributes, and references. It produces a python object representation of the DOML model, which is rather complex in its structure. The parser also tries to establish which DOML version to use automatically, starting from the most recent version, unless a specific version is enforced.
- If the DOMLX model includes a *functional requirement* field, its content, the DOMLR source, is parsed by the **DOMLR parser** to produce custom requirements.
- For simplicity's sake, the result of the DOMLX parser is converted to an **Intermediate Model (IM)** that is based on Python dictionaries. The conversion is carried out according to the DOML Metamodels, that detail how to correctly map and convert each element.
- User requirements and built-in requirements (common requirements) are merged into a uniform data-structure.
- The **Intermediate Model Checker (IMC)** receives the DOML model in IM form and translates into a set of assertions for the Z3 SMT solver (the logic engine), along with the requirements that are translated too into Z3 expressions. Z3 expressions consist of First-Order Logic formulas containing theories provided by the SMT solver.
- The **synthesis module** is present but is not accessible by the IDE, as it is still a work in progress. It can be invoked from the CLI.

The following sequence diagram explains the order of execution of the above components.

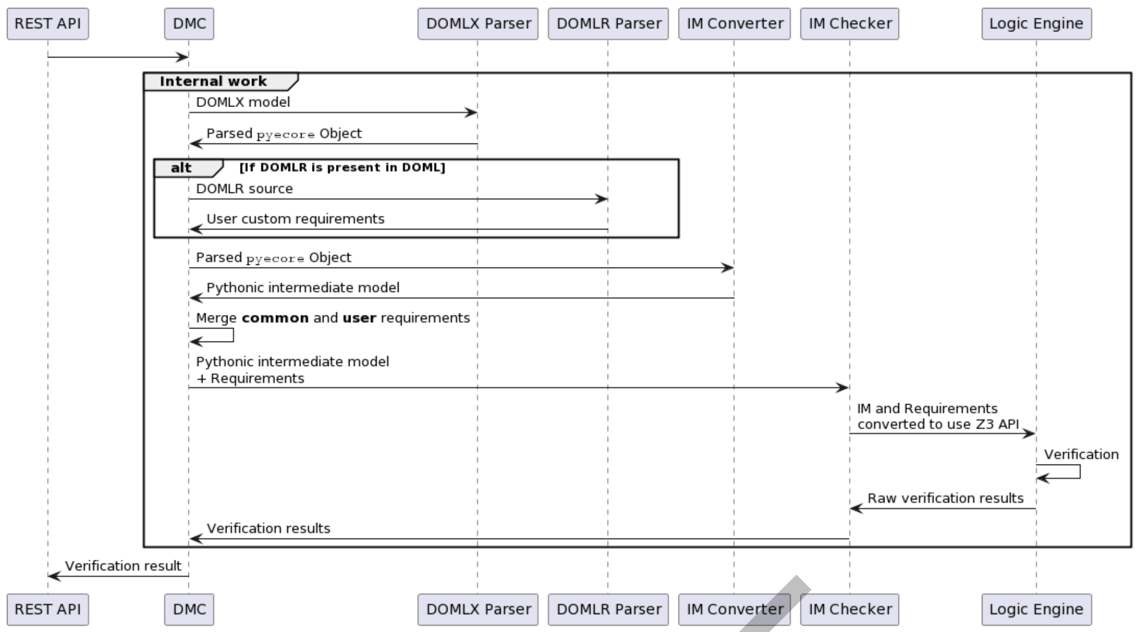

Figure 4. DMC internal sequence diagram.

The DMC is also called by the Self-Healing workflow of PIACERE to validate a DOML model, through the same REST API, as shown in the figure below.

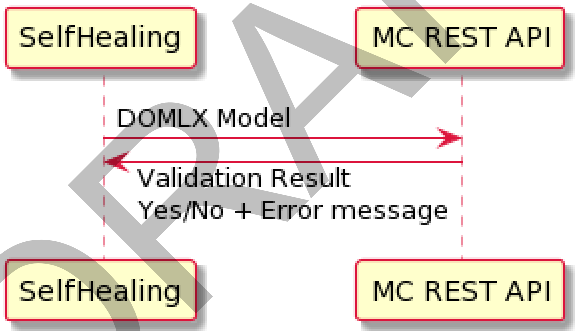

Figure 5. DMC called by the Self Healing workflow

1.2.4 Technical specifications

The DMC has been written in the Python programming language, and its library dependencies are managed through pip and a virtual environment. The REST APIs, specified in *OpenAPI* [5], have been implemented with *Connexion* [6], a Python library that implements REST APIs directly from their OpenAPI specification. The DOMLX parser is based on the *pyecore* [7] library. The RestAPIs are documented through *SwaggerUI* [8], which generates online documentation directly from OpenAPI specifications. The DSL for custom requirement has a parser written with *Lark* [4], a python parser generator library.

The general DMC documentation is written in the *reStructured Text* format and rendered with Sphinx [9]. The pytest library is used for managing regression tests.

The DMC can be both run locally or through a *Docker* [10] container, whose Dockerfile is provided for easier setup. The Docker image is based on the official Python Debian image and uses the *Uvicorn* [11] web server to deploy the REST APIs.

2 Delivery and usage

2.1 Package information

Since D4.2, few changes were made to the overall project structure, and the overall structure remains the same.

The project folder is structured as follows:

- docs – A folder containing the documentation of the project
- mc_openapi – A folder containing the source code of the project
    - handlers.py – REST endpoint action configuration
    - openapi – Folder containing the OpenAPI YAML configuration
    - notebooks – Folder containing Python notebooks used during development
    - assets -- Folder containing data sources for DOML metamodels (ECORE files, YAML target representation) for each supported DOML version
    - doml_mc – The model checker program
    - csp_compatibility – A tool to verify the model deployment compatibility with selected Cloud Service Providers (CSP).
    - domlr_parser – The parser for the DSL for custom user requirements
    - intermediate_model – Contains the definition and utilities to interface with the Intermediate Representation of the DOML model
    - xmi_parser – Contains the translator from DOMLX to Plain Python Objects
    - z3encoding – Contains the translator from IM to Z3 Logic Formulas through its Python API
    - common_reqs.py – Built-in requirements written in Z3 Python API
    - consistency_reqs.py – Consistency requirements now superseded by the IDE checks.
    - Error_desc_helper.py – Utility to retrieve values for detailed error messages.
    - imc.py – Functions to configure and run the model checker.
    - mc.py – Entrypoint of the model checker.
    - mc_results.py – Handles the results of model checking
- tests – A folder containing the suite of tests for the MC

For a simpler CI/CD integration, we deprecated the use of 'Poetry' as a package manager and moved back to 'pip' and 'venv' (which is an integrated solution with Python installations).

The synthesis feature has been developed as a standalone module which is a dependency for the model checker, available at https://github.com/andreafra/piacere-synthesis.

2.2 Installation Instructions

The DMC can be run by downloading its source code from the following Git repository:

```
$ git clone https://github.com/andreafra/piacere-model-checker
```

It can be run directly from the source code, or from a Docker container. It contains the following release branches:

- **main** – development branch. Possibly unstable.
- **y2** – DMC version developed in the second year of the project. Stable.
- **y3** – DMC version developed in the final year of the project. Stable.

2.2.1 Running from source

After cloning the repository, create a Python virtual environment with:

```
python -m venv .venv
```

and then activate it:

```
source .venv/bin/activate
```

Install the dependencies with:

```
pip install -r requirements.txt
```

Start the REST API locally with:

```
python -m mc_openapi
```

You will have to point the IDE to *http://localhost:8080* from the settings.

Alternatively, you can use the CLI. Use the following command to view the help guide:

```
python -m mc_openapi -h
```

Please refer to https://piacere-model-checker.readthedocs.io/en/latest/installation.html for the most up-to-date instructions.

2.2.2 Running as a container

If you intend to run it as a REST API, it's better to use containers.

You can build and run a container with the following commands:

```
docker build -t wp4/dmc .
docker run -d wp4/dmc
```

The container will use the Uvicorn to run the server. Remember to bind the container ports for local use using *-p 127.0.0.1:8080:80/tcp*.

2.2.3 User Manual

A complete and up-to-date documentation of the project is available at the following address:

https://piacere-model-checker.readthedocs.io

You can also build it from source running the following commands:

```
cd docs
make html
```

For the REST API specification, visit the http://[server ip]:[port]/ui/, if running locally it will be http://localhost:8080/ui/ to view the SwaggerUI documentation.

2.3 Licensing Information

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