POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Formal Verification of Infrastructure as Code

Tesi di Laurea Magistrale in
Computer Science and Engineering

Author: **Michele De Pascalis**

Student ID: 904966
Advisor: Prof. Matteo Pradella
Co-advisors: Dr. Michele Chiari
Academic Year: 2020-21

Abstract

Infrastructure-as-Code (IaC) is a system administration and software management paradigm that gained relevance in the industry with the widespread adoption of cloud computing technologies. Although IaC theoretically opens up the possibility for automatic verification of infrastructural specifications, available works on the subject focus on analysing operational aspects of the infrastructure lifecycle, such as deployment, orchestration and management. Little effort has been performed towards verifying structural and qualitative aspects of infrastructural code. In this document, we present DOML-MC, a prototype model checker back-end for DOML, an IaC language that is being developed as part of the PIACERE project. Infrastructural elements, their attributes and associations between them are encoded as an SMT problem, which is solved by the Z3 SMT solver. This approach proved useful to check critical or desirable properties of the analysed IaC document, but also to seek ways in which the infrastructure could be enriched to meet such properties.

Keywords: Infrastructure-as-Code, Satisfiability Modulo Theories, Model Checking

Abstract in lingua italiana

Infrastructure-as-Code (IaC) (in italiano "infrastruttura come codice") è un paradigma di amministrazione di sistema e gestione del software che ha acquistato rilevanza nell'industria con l'adozione diffusa delle tecnologie di *cloud computing*. Sebbene l'IaC apra teoricamente alla possibilità di verificare formalmente specifiche infrastrutturali, i lavori disponibili sull'argomento si concentrano sull'analisi di aspetti operazionali del ciclo di vita dell'infrastruttura, come *deployment*, orchestrazione e gestione. Non molto è stato fatto allo scopo di verificare aspetti strutturali e qualitativi del codice infrastrutturale. In questo documento, presentiamo DOML-MC, un *back-end* di *model checker* prototipale per DOML, un linguaggio di IaC che è in corso di sviluppo come parte del progetto PIACERE. Elementi infrastrutturali, i loro attributi e le associazioni tra di essi sono codificati in un problema di SMT, che viene risolto dal *solver* SMT Z3. Questo approccio è risultato utile nel controllare proprietà critiche o desiderabili del documento IaC analizzato, ma anche per ricercare modi in cui l'infrastruttura possa essere arricchita per andare incontro a tali proprietà.

Parole chiave: Infrastructure-as-Code, Satisfiability Modulo Theories, Model Checking

Contents

Introduction

Infrastructure-as-Code (IaC) is a system administration and software management paradigm that gained relevance in the industry with the widespread adoption of cloud computing technologies. Cloud computing allowed the management of pieces of infrastructure to be automated, and enabled the development of languages that a software developer or system administrator can use to describe such automation as code. The resulting time saving, configuration reproducibility and robustness to operational error caused IaC to capture the interest of many companies that provide or employ cloud services.

Inevitably, in simplifying the management of infrastructure, IaC created room for the development of progressively larger and more complex infrastructural architectures, in which development errors became more frequent and harder to locate. This leads one to imagine applying formal verification technologies, akin to those used to verify the correctness of software programs, to infrastructural code. Nevertheless, little research has been performed in this direction, and the existing works focus on verifying properties of the process of deploying the infrastructure, rather than properties of the described infrastructure itself.

In this work, we consider multiple approaches for the representation of descriptive properties of IaC models in logical systems, to enable the automated mechanical verification of formally structured logical specifications. We produce two prototypes based on different logical back-ends, namely Prolog and Z3, through which formal specifications written in *ad hoc* DSLs can be verified against TOSCA infrastructural templates. The prototypes are shown to be effective in analysing a simple TOSCA topology template; however, Prolog is discarded as a back-end due to an inconvenience in its semantics for negation.

Finally, we arrive at the DOML Model Checker (DOML-MC), a model checker back-end encoding infrastructural descriptions as Satisfiability Modulo Theories (SMT) problems for Microsoft's Z3 solver. DOML-MC has been developed within the scope of the Programming trustworthy Infrastructure As Code in a sEcuRE framework (PIACERE) project, a research project funded as part of the European Union's Horizon 2020 programme. The aim of PIACERE is to investigate new technologies for DevOps, improving

ease of development and vendor independence of IaC. In order to achieve this, within the project, a new IaC language, the DevOps Modelling Language (DOML), is being developed, together with a set of tools to operate with it. DOML-MC is among these tools.

To evaluate the tool, we check simple structural properties of four DOML documents, properties such as the fact that software packages that depend on one another should be deployed on computing nodes that can communicate through a common network. Verification times are not instantaneous, as is often the case with model checking, but the approach is nonetheless interesting due to the expressiveness of SMT formulas used to encode requirements, as well as the possibility of performing model synthesis.

The rest of this document is organized as follows: Chapter 1 discusses Infrastructure-as-Code and provides a survey of works aimed at IaC verification; Chapter 2 illustrates our approach to the problem, and the prototypes that were developed during its investigation; Chapter 3 presents the DOML Model Checker and explains its functioning; finally, Chapter 4 evaluates the conclusions and future developments of this work. Implementation details of all software projects developed in the scope of this work are reported in the appendices.

1 | Background

1.1. Infrastructure as Code

Infrastructure-as-Code (IaC) is an approach in the management of computing infrastructure according to which computing resources should be defined and automatically provisioned, configured and managed in source code akin to that used to describe computer programs. This allows system administrators to apply to infrastructural descriptions the same tools and techniques that are used by developers for their code. Quoting K. Morris [20],

> The premise is that modern tooling can treat infrastructure as if it were software and data. This allows people to apply software development tools such as version control systems (VCS), automated testing libraries, and deployment orchestration to manage infrastructure. It also opens the door to exploit development practices such as test-driven development (TDD), continuous integration (CI), and continuous delivery (CD).

1.1.1. A brief history of Infrastructure as Code

A first known instance of using code to describe configuration before the spread of cloud technologies was Mark Burgess' tool CFEngine [8] in 1993, used to replicate configuration on heterogeneous UNIX systems. The tool is described by the author as an interpreter for a configuration language, so the intent to represent configuration as code emerges. Although in its first presentation the "configuration program" was presented as a set of actions, rather than a description of the desired configuration, Burgess proceeded to develop the concept of *convergent operators*, i.e., actions that make the system converge to a fixed point, which is its desired final state. This can be seen as a precursor to the concept of *idempotence*, widespread in modern IaC technologies.

Configuration management tools gained traction in the first decade of the 2000s: some

notable examples supporting IaC are Puppet[1] (initially released in 2005), Chef[2] (2009), Salt[3] (2011) and Ansible[4] (2012). As an example of the modern approach in configuration management IaC, Ansible modules, the functional atoms of the Ansible system, are generally built to be idempotent, so that invoking them twice on the same system has the same effect on it as one single invocation, which should be enough to reach the desired configuration on the system. This also allows structuring module invocations to have a declarative appearance, describing what the configuration should be rather than the actions to be taken to reach it.

Listing 1.1: Example invocation of the `ansible.builtin.file` Ansible module. Notice how the desired absence of a directory is described as a state, rather than for instance as `action: delete`.

```
- name: Recursively remove directory
  ansible.builtin.file:
    path: /etc/foo
    state: absent
```

With the advent of virtualization and cloud technologies, infrastructural entities such as computers and network devices were replaced by their virtual counterparts, which could be created, managed and destroyed programmatically. Cloud Service Providers (CSPs) introduced interfaces with the main configuration management tools, that could be then used to describe the state of infrastructural resources, i.e., they could be used for provisioning. Additionally, the principal CSPs introduced IaC languages of their own for provisioning on their platforms: Amazon Web Services *CloudFormation*[5] was released in 2011, Google Cloud *Deployment Manager templates*[6] and Microsoft Azure *Resource Manager templates*[7] in 2014.

During the same years, pieces of software were released that allowed to control virtualized environments through code. *Vagrant*[8] (first version in 2010) uses Ruby scripts written in

[1] `https://puppet.com/`
[2] `https://www.chef.io/`
[3] `https://saltproject.io/`
[4] `https://www.ansible.com/`
[5] `https://aws.amazon.com/cloudformation/`
[6] Google Cloud Deployment Manager documentation:
`https://cloud.google.com/deployment-manager/docs`
[7] ARM template documentation:
`https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/`
[8] `https://www.vagrantup.com/`

a declarative DSL to pilot pre-existing virtual machine managers, such as VirtualBox and KVM, allowing to manage the lifecycle of virtual machines with minimal user interaction; in Vagrant practice, configuration management tools like Chef or Puppet can also be leveraged to further automate the configuration of the managed VMs.

A short while after, in 2013, *Docker*[9] saw its first release. In contrast to virtual machines managed by Vagrant, Docker is used to run software in *containers*, virtualized environments that provide *OS-level virtualization*, a form of virtualization in which all virtualized units share the same operative system kernel, while the layers above, such as the filesystem, are virtualized. Docker provides *Dockerfile*s as a way to script the creation of container images, from which containers are then instantiated as needed. Container instantiation, lifecycle management and virtual networking can also be specified in pieces of code, interpreted by a tool called *Docker Compose*.

Using IaC to manage cloud infrastructure has historically been tied to the CSP or CSPs on top of which the infrastructure is to be deployed. The API and language used to write IaC scripts is highly specific to the CSP providing the interface; this causes service migration between CSPs to require a new development phase, during which the infrastructural code has to be rewritten to conform to the service structure of the new provider. An immediate consequence is vendor lock-in. In 2014 HashiCorp, the software company developing Vagrant, released a new tool called *Terraform*[10]. Terraform interprets declarative infrastructure specifications written in a language called *HashiCorp Configuration Language* (*HCL*), and interfaces with the Application Programming Interfaces (APIs) provided by various CSPs to provision the described architectures. The advantage of Terraform is the introduction of a syntactically uniform language to describe and materialize cloud infrastructures; however, below the syntactic layer, the code interface of the modules used to communicate with different CSPs is still provider-specific, and vendor lock-in persists as a major concern when using IaC to manage cloud infrastructure.

Being tied to the industry since its inception, IaC has mainly relied on proprietary languages and *de facto* standards. Establishing an open standard for cloud infrastructure development through IaC is a goal of the *Topology and Orchestration Specification for Cloud Applications* (TOSCA)[11], developed and sponsored by the *Organization for the Advancement of Structured Information Standards* (OASIS), that approved version 1.0 of the standard in 2014. Several projects targeting TOSCA have since been developed,

[9] `https://www.docker.com/`
[10] `https://www.terraform.io/`
[11] `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca`

such as Cloudify[12] and XLAB's xOpera[13]. The latter originated as an improvement on the DICER tool [2], produced within the scope of the DICE project, funded as part of the Horizon 2020 European research and innovation programme[14]. While targeting an open standard, Cloudify and xOpera suffer from a similar issue to that of Terraform, as the contents of the infrastructural specifications must be written in a sub-language that depends on the CSP chosen for deployment, impairing cross-provider portability.

1.1.2. Phases of infrastructure lifecycle targeted by IaC

Provisioning

Infrastructure provisioning is the act of making available infrastructural resources, such as computing nodes, networks, storage etc. In a pre-cloud conception, this step consisted in a host of manual operations, such as physically retrieving and powering computers, cabling and setting up network devices and formatting drives. Cloud providers now offer access to virtual resources like virtual machines and networks, and since these resources are created and managed by software, their provisioning can be automated.

Configuration management

Once computing resources are available, they have to be set up, to create the operating environment and install the dependencies needed for the deployed software to execute. This is known as *configuration management*. Reaching the needed configuration requires installing third-party software and editing configuration files in the operative system on the computing resources that compose the infrastructure, and in the early days such tasks were performed manually. This implied a difficulty in replicating the same configuration in more than one instance. When reconfiguring a resource after resetting it to restore an initial desired state, this leads to lack of reproducibility; when replicating the same configuration in parallel on multiple simultaneously functioning resources, this leads to a phenomenon known as *configuration drift*. Automation of configuration management leverages IaC to address these issues.

Software deployment

Software deployment is the last step towards getting a software solution running on the provisioned and configured infrastructure. This usually means transporting the software

[12] `https://cloudify.co/`

[13] xOpera is presented in XLAB's blog:
`https://www.xlab.si/blog/xopera-get-your-orchestrator-pitch-perfect/`

[14] `https://ec.europa.eu/programmes/horizon2020/en/home`

artefact onto one or more computing resources, setting the necessary execution environment (environment variables, restart policies on failure, logging, etc.) and finally executing the software. Once again, given the number of variables involved, this task benefits from an automated workflow in terms of reproducibility and celerity.

An additional step can be taken by setting up what is called a *Continuous Integration/-Continuous Deployment* (*CI/CD*) pipeline. In CI/CD the deployment automation is taken all the way back to the software developer: an appropriate commit in a centralized software repository, usually a VCS repository, triggers a pipeline of functional and integration tests, after which the software is automatically deployed.

Orchestration

Modern IT services are often characterized by volatile demand; while over-provisioning is obviously wasteful in terms of cost and resources usage, dimensioning the capabilities of an infrastructure according to the typical load leaves the possibility of service discontinuity on the occasion of a demand spike. In the case of an infrastructure directly defined in terms of physical resources, enacting the flexibility to accommodate a change in demand means fetching and manually provisioning new machines, with the associated expenses, then configuring the new execution environment and redeploying the software.

Virtualization and programmable interfaces for resource management allowed an ulterior kind of automation, called *orchestration*, aimed at providing the desired flexibility with minimal inconvenience. In a cloud setting, a piece of software called an *orchestrator* can automatically ask the cloud service provider for a more capable machine and deploy the software service on top of it (vertical scaling) or, where parallelism is more effective, replicate the deployment on a number of machines (horizontal scaling). Once the service demand has receded to its usual levels, the additional resources can be returned to the cloud service provider, optimizing overall cost.

1.2. The PIACERE project

The work underlying this thesis was motivated within the scope of a project titled *Programming trustworthy Infrastructure As Code in a sEcuRE framework* (PIACERE). The PIACERE project has been funded as part of the Horizon 2020 European programme. As stated in the project proposal [14] (Section 1),

> The main objective of the PIACERE project is thus to provide means (tools, methods and techniques) to enable most organizations to fully embrace the

Infrastructure-as-Code approach, through the DevSecOps philosophy, by making the creation of such infrastructural code more accessible to designers, developers and operators (DevSecOps teams), increasing the quality, security, trustworthiness and evolvability of infrastructural code while ensuring its business continuity by providing self-healing mechanisms anticipation of failures and violations, allowing it to self-learn from the conditions that triggered such re-adaptations.

To achieve the IaC DevSecOps concept, PIACERE will provide an integrated **DevSecOps framework** to develop, verify, release, configure, provision, and monitor infrastructure as code. The extensible architecture and modular approach of PIACERE will support the different DevSecOps activities. Using a single **integrated environment to develop (IDE)** infrastructural code will unify the automation of the main DevSecOps activities and will shorten the learning curve for new DevSecOps teams. PIACERE will allow DevSecOps teams to model different infrastructure environments, by means of abstractions, through a novel ***DevOps Modelling Language*** **(DOML)**, thus hiding the specificities and technicalities of the current solutions and increasing the productivity of these teams. Moreover, PIACERE will also provide an extensible **Infrastructural Code Generator (ICG)**, translating DOML into source files for different existing IaC tools, to reduce the time needed for creating infrastructural code for complex applications. The provided **extensibility mechanisms (DOML-E)** shall ensure the sustainability and longevity of the PIACERE approach and tool-suite (new languages and protocols that can appear in the near future).

The most fundamental resulting artefact through which developers and other technicians will interface with the PIACERE framework will be the DevOps Modelling Language, an IaC language providing an abstraction over specific infrastructure management tools. Quoting [14] again (Section 1.1), "the DevSecOps team will not be burdened by the wide variety of underlying IaC languages and configuration protocols and will be able to reason on the infrastructural stack needed by the applications in a more effective way".

Another objective of PIACERE is to "[provide] the DevSecOps Teams with the tools to verify the correctness of the infrastructural models and the trustworthiness and security of the IaC and the associated software components". This objective unfolds in three key results (KRs), the first of which, KR5, shall be the *Verification Tool* (*VT*). The VT "will be a suite of static analysis and model checking tools aiming at verifying that models and resulting pieces of code fulfill specific properties, ranging from consistency of the

model (e.g., the characteristic of the resource selected for running a certain component should match the required characteristics in terms of speed and memory) to non-functional aspects such as safety, performance, and privacy properties related to the exchange of data among software components (e.g., data cannot be transferred to a third party without going through some filtering component)". The model checking component of this KR is the main focus of this thesis.

1.3. Formal verification of Infrastructure as Code

Verification of IaC, analogously to software verification, can be broadly divided in two scopes, static analysis and dynamic analysis. Dynamic analysis consists of running the IaC specifications in a controlled environment and observing the resulting behaviour looking for faults, and is suited for discovering problems regarding the implementation of the tools that interpret IaC, or the way the described infrastructure interacts with an environment close to the real world. On the other hand, static analysis is concerned with the problems that can be inferred from a specification in itself, and is more apt to discover faults in the conceptual modelling of the infrastructure, or in the way it is described. The two approaches are complementary to each other: while static analysis cannot fully predict how the modelled infrastructure will react to a real running environment, dynamic analysis can only test one scenario at a time, and thus in general no suite of dynamic tests can be exhaustive enough to cover all corner cases that could be presented to the infrastructure in a production setting.

Model checking [11] is a formal-verification approach where an engineered system is modelled in a logical framework in which it is feasible to check that certain desired properties are guaranteed, or whether undesirable situations may occur. Traditionally, systems are modelled with some kind of graph or transition system, which are natural representations for evolving stateful systems such as electronic devices or imperative programs. Recently, techniques involving SAT (*Boolean Satisfiability*) and *Satisfiability Modulo Theories* (SMT) solvers appeared. They leverage advancements in algorithms for SAT (such as DPLL) to provide tools that often are very fast on practical models, despite the theoretical complexity bounds.

The translation of the verification target into the modelling language can be performed manually by a software architect, or automatically. Manual translation benefits from deep knowledge of the target, but it can be laborious and error-prone. Devising an automatic translator is seldom trivial, and often comes with a cost in generality.

Due to well-known results in Computer Science, there is no automatic procedure capable

of singling out all and only the pieces of software, written in a Turing-complete programming language, whose behaviour conforms to a given non-trivial property. Consequently, automatic verification approaches must be more restrictive than it would be desirable in guaranteeing that a program satisfies a requirement. This does not necessarily apply to IaC, where the concern is rather about the capability of the logical framework to express system properties. E.g., a model focused on relationships between cloud computing nodes will not be able to predict an internal application malfunction on one of such nodes.

In this section, works in IaC model checking are described.

K. Jayaraman et al. [16] developed SecGuru, a tool that analyses firewall ACLs within infrastructures deployed in the Azure cloud. SecGuru enables inspection of the differences resulting from an ACL update in terms of network packets that are allowed or blocked by the firewall, and to check the behaviour of the firewall against a given *policy*. This is obtained by encoding the action taken by the firewall on packets and the policies into an SMT problem, then solved by the Z3 SMT tool. The resulting instances correspond to the network packets that are either blocked or allowed. The tool was evaluated on real and synthetic policies, and is currently active in the Azure cloud, reportedly having a "measurable positive impact in prohibiting policy misconfigurations".

A. Brogi et al. [7] propose a tool to verify the validity of TOSCA management plans. The user enriches each TOSCA node template with a set of compatible states. Then, they characterize management operations and states by specifying the states in which the nodes providing the required capabilities need to be for the execution of the plan to be successful. The tool checks the validity of a plan by creating a state-representation of the whole infrastructure. In this representation, a state is a valid combination of the states of individual nodes, and a transition between states is a management operation on a node such that the operation's requirements are satisfied. The validity of the plan is then equivalent to the existence of an operation-labelled path.

W. Chareonsuk and W. Vatanawood [9] present a toolchain to perform formal verification of interacting web services in a TOSCA specification. The process relies on user-supplied information describing the behaviour of the modelled services, written in the *Web Services Business Process Execution Language* (WS-BPEL, or BPEL). The user provides a BPEL description of services running on the infrastructure, and integrates it in the TOSCA service template using *ad hoc* node and relationship types. This IaC is then compiled into a PROMELA specification, against which *Linear Temporal Logic* (LTL) [11, Section 2.3] formulas can be verified using the SPIN model checker [15]. As an example, the authors propose checking *safety* properties, expressed as formulas of the form $\Box\neg P$, i.e., "it is

always true that P does not hold", where P is some undesirable condition.

R. Shambaugh et al. [23] developed Rehearsal, a tool to analyse Puppet configurations by compiling them into a formal language describing filesystem operations, and then translating them into SMT specifications. The Z3 SMT solver is used to look for models representing executions that violate the principles of determinism and idempotency. Rehearsal was tested on a small set of 13 Puppet configurations gathered from GitHub and Puppet Forge, and found bugs in 6 of them. Benchmarks on the test set run on a quad-core 3.5 GHz Intel Core i5 with 8 GB RAM measured average checking times within 3 seconds for all configurations.

J. Lepiller et al. [19] identified a class of cloud-related security vulnerabilities, called *intra-update sniping vulnerabilities*. These occur when an infrastructure update operation, despite transitioning between secure states, traverses insecure intermediate states, for instance because components are updated in the wrong order. To detect this vulnerability class in CloudFormation templates, the authors developed Häyhä, which models the described infrastructure as a *dataflow graph*. Häyhä was evaluated on a set of open-source CloudFormation templates: while no vulnerability was detected, the tool showed performances acceptable for integration in a deployment workflow, as execution time was within 1 second for all templates.

H. Yoshida et al. [24] propose a method for manual modelling of TOSCA service templates in the formal specification language CafeOBJ. This method is useful for proving that orchestration operations can reach the final state of the infrastructure while maintaining a given invariant property in intermediate states. Proving a property modelled in CafeOBJ is a form of formal verification, but it cannot be regarded as model checking since it requires user interaction.

Within the scope of the RADON project, M. Law and A. Russo [18] developed a Verification Tool (VT) for a subset of TOSCA that was devised for the project. The VT is operated through a Constraint Definition Language (CDL), in which objects (complex values with properties) can be specified together with logical properties regarding such objects. The CDL also provides the capability to import RADON models, *i.e.* RADON TOSCA topology templates, and automatically translate relevant parts of the imported models into CDL objects. The VT then uses a constraint solving technique called *Answer Set Programming* (ASP) to verify the resulting specification, and possibly search for a *correction* among a number of user-allowed modifications to the target model.

A number of authors presented techniques for model checking of cloud infrastructure in which the model is constructed manually. H. Sahli et al. [22] proposed *bigraphical reac-*

tive systems as a suitable logical framework to model cloud infrastructure lifecycles, and exemplified the use of the model checker BigMC to check relevant properties of elasticity and plasticity. K. Klai and H. Ochi [17] presented a technique for model checking the interaction of multiple cloud services accessing shared resources concurrently. The cloud services are modelled as *RCoWF* (*Resource-Constraint open WorkFlow*) nets and translated into labelled Kripke structures, against which *hybrid LTL* formulae are checked, e.g., to detect deadlocks on a concurrently accessed resource. These works are not described further here, because they do not address IaC directly.

2 | Approach and prototypes

The work plan of the PIACERE project, illustrated in [14, Section 3], identifies a number of *tasks*, which are organized in eight Work Packages (WPs).

As can be seen in [14, Section 3.2.2], the main WPs are scheduled to have their first release simultaneously. This created difficulties due to the fact that often the development of components depends on the definitive state of other components, developed in different WPs. Most notably, the majority of the expected resulting tooling cannot be developed without a final definition of the DOML (WP3). WP4, containing the Verification Tool, is supposed to be developed alongside WP3, but developing a model checker for an IaC language for which we had no solid definition or constraints was clearly a conundrum. Many subtleties of model checking involve finding or developing a logic tool with exactly the minimal expressive power required by the class of models to represent and inspect, but in the absence of a DOML definition, which was especially late to appear during the execution of the project, we could not follow an approach of this kind from the beginning.

While waiting for the DOML to reach a stage of definition that allowed us to develop the final tool, we investigated the field with proofs of concept and prototypes. We chose an IaC language that was close enough and at least as expressive as we expected DOML to be, that is, OASIS TOSCA, and we conducted a few experiments towards performing model checking on it.

Model checking is traditionally aimed at software verification, or more generally at the verification of dynamic systems whose behaviour can be expressed in terms of states and transitions among them.[1] As presented in Section 1.3, model checking of cloud systems and IaC also mainly focused on dynamic aspects of infrastructure, such as deployment and management procedures. In our work, we wanted to inspect IaC in its declarative side, *e.g.*, analysing the relationships between infrastructural entities, rather than the order in which the entities ought to be concretized by a CSP.

For this reason, we focused on modelling IaC and representing infrastructural specification

[1] For an extensive treatment of the techniques, see [11].

in terms of *constraint satisfaction*, and using *constraint programming* technologies to implement our target solution.

2.1. The TOSCA Prolog prototype

2.1.1. Logic programming and the Prolog language

Logic programming [3] is a programming paradigm founded on formal logic, in which a program is given as a set of predicates acting on terms, together with a set of logical constraints relating the predicates. In our first prototype, we used Prolog as our of logic programming language. Prolog [12] is the most widely adopted programming language concretizing the theory of logic programming, and several implementations exist.

In a Prolog program, elementary knowledge is represented by *terms*, which can be either simple or compound. A simple term can be a number, such as `45`, an *atom*, which is a term which only represents or identifies itself, such as `greg`, or a variable, such as `X`. Atoms start with a lowercase letter, variables start with an uppercase letter. Simple terms can be combined in *compound terms*: given the simple terms above, we can create the compound term `person(greg, 45)`.

A *predicate* represents something that can be true of one or more terms, or a relationship between terms. A Prolog program consists of a set of *clauses*, logical constraints relating predicates and terms. A *fact* is an assertion that a predicate holds for certain terms independently of external knowledge. As an example, considering the terms presented above, together with the atom `ppth`, the fact:

```
works(person(greg, 45), ppth).
```

relates the predicate `works`, with arity 2, the term `person(greg, 45)` and the term `ppth`, stating that the predicate holds for these two terms. This fact could represent the knowledge that "The person named Greg who is 45 years old works at the PPTH".

It is then desirable to express knowledge relating more than one predicate. This is done via *rules*, which are clauses composed of a *head* predicate, and a body composed of one or more logically combined predicates. A rule is used to assert that the head predicate holds for certain terms *if* the body holds. For example:

```
employed(X) :- works(X, _).
```

The sign ':-', which is concrete syntax for the more formal '←', should be considered a reversed implication, and can be read as "if". Together with the fact that '_' is a placeholder that stands for any term, the above rule can be read as "X is employed if X works anywhere".

The collection of clauses that constitute a Prolog program are also called its *database*. The database can then be queried, a *query* being a predicate applied to some terms, possibly including variables. Querying the Prolog instance results in asking whether it follows from the clauses in the database that a predicate holds for certain terms. The Prolog instance can reply negatively or positively; if the terms in the query contain variables, the response to the query contains the values of the variables for which the predicate holds. If the two clauses above are in our database, querying for the name of an employed person results in the expected answer:

```
?- employed(person(X, _)).
X = greg.
```

On the other hand, specifying that the age of the person we are seeking should be 30, we are met with a negative answer:

```
?- employed(person(X, 30)).
false.
```

The mechanism through which Prolog matches terms and variables is called *unification*. For example, when determining whether a fact in the database is a satisfactory instance of a query, Prolog checks that the two predicates are the same, and then tries to unify the respective terms, *i.e.*, checks if they can be equal for some assignment of the variables that appear in them.

2.1.2. Representing IaC within Prolog

Unification being its central functioning mechanism, the Prolog language can be used for constraint programming [1, Section 9.2.1]. Our approach was to encode infrastructural elements and their metadata as follows:

- Scalar values such as integers, floating point numbers and strings were encoded as their respective counterpart in Prolog;

- Boolean values were encoded with two atoms, `true` and `false`;
- A usage of a TOSCA intrinsic function such as `get_input` was encoded with a term with a corresponding structure, e.g., `get_input(`*IName*`)`, where *IName* is the name of the input;
- A TOSCA property definition was encoded with the term `property(`*PName*`,` *PType*`,` *Required*`)`, where *PName* is an atom identifying the name of the property, *PType* is an atom for its type and *Required* is either `true` or `false` depending on whether the property is defined to be required or not;
- A TOSCA capability definition within a node type definition was encoded with the term `capability(`*CName*`,` *CType*`)`, where *CName* is an atom identifying the name of the capability and *CType* is an atom for its type;
- A TOSCA requirement definition within a node type definition was encoded with the term `requirement(`*RName*`,` *RCap*`,` *RNType*`,` *RRel* `, occurrences(`*ROccLB*`,` *ROccUB*`))`, where *RName* is an atom for the name of the requirement, *RCap* is an atom for the required capability, *RNType* is an atom for the type of nodes admitted as a target (including subtypes), *RRel* is an atom representing the relationship underlying the requirement, and *ROccLB* and *ROccUB* are respectively lower and upper bound for the number of occurrences of the requirement, *ROccUB* possibly being the atom `unbounded`;
- A TOSCA node type definition was encoded with the fact `node_type(`*NType*`,` *EType*`,` *PDefs*`,` *CDefs*`,` *Reqs*`)`, where the predicate `node_type` relates the following terms:
  - *NType*, an atom identifying the node type;
  - *EType*, an atom identifying the type extended by this node type, or `none` if no type is extended;
  - *PDefs*, a list of property definitions, encoded as defined above;
  - *CDefs*, a list of capability definitions, encoded as defined above;
  - *Reqs*, a list of requirement definitions, encoded as defined above;
- A TOSCA capability type definition was encoded with the fact `cap_type(`*CType*`,` *EType*`,` *PDefs*`)`, where the predicate `cap_type` relates the following terms:
  - *CType*, an atom identifying the capability type;

  - *EType*, an atom identifying the type extended by this capability type, or `none` if no type is extended;
  - *PDefs*, a list of property definitions, encoded as defined above;

- A TOSCA property within a node template or capability was encoded with the term `property(`*PName*`,` *PValue*`)`, where *PName* is defined as above and *PValue* is the term representing the value of the property;
- A TOSCA capability offered by a node template was encoded with the term `capability(`*CName*`,` *Props*`)`, where *CName* is the atom representing the name of the capability, and *Props* is a list of properties, encoded as defined above;
- A TOSCA requirement satisfied for a node template was encoded with the term `requirement(`*RName*`,` *RTarget*`)`, where *RName* is the atom representing the name of the requirement, and *RTarget* is the node template provided to satisfy the requirement;
- A TOSCA node template definition was encoded with the fact `node(`*NName*`,` *NType*`,` *Props*`,` *Caps*`,` *Reqs*`)`, where the predicate `node` relates the following terms:
  - *NName*, an atom identifying the node by its name;
  - *NType*, an atom identifying the node type;
  - *Props*, a list of properties, encoded as defined above;
  - *Caps*, a list of capabilities, encoded as defined above;
  - *Reqs*, a list of requirements, encoded as defined above;
- A TOSCA policy was encoded with the fact `policy(`*PName*`,` *PType* `,` *PTargets*`)`, where *PName* is an atom identifying the policy by its name, *PType* is an atom identifying the policy type, and *PTargets* is a list of atoms identifying node templates which are the targets of the encoded policy.

Information about infrastructural elements and their metadata was obtained by parsing an example TOSCA template with a suitably modified third-party TOSCA parser[2]. A Prolog instance was automatically populated with the information encoded as facts.

[2]The modified version can be found at `https://github.com/mikidep/tosca-parser`.

Listing 2.1: A generated fact encoding a node type.

```
node_type(
    'doml.nodes.Redis',
    'tosca.nodes.SoftwareComponent',
    [property(component_version, version, false), property(
       admin_credential, tosca.datatypes.Credential, false)],
    [capability(redis_endpoint, 'doml.capabilities.Endpoint.
       Redis'), capability(feature, 'tosca.capabilities.Node'
       )],
    [requirement(host, 'tosca.capabilities.Container', 'tosca
       .nodes.Compute', 'tosca.relationships.HostedOn',
       occurrences(1, unbounded)), requirement(dependency, '
       tosca.capabilities.Node', 'tosca.nodes.Root', 'tosca.
       relationships.DependsOn', occurrences(0, unbounded))]
).
```

After this, a file containing auxiliary Prolog predicates was loaded into the instance, to provide the building blocks for knowledge base querying.

Listing 2.2: An auxiliary predicate, relating node types with their ancestors.

```
extends_node_type(TypeA, TypeA).
extends_node_type(TypeA, TypeB) :-
    node_type(TypeA, ExtTypeA, _, _, _),
    ExtTypeA \= none,
    extends_node_type(ExtTypeA, TypeB).
```

A user with some Prolog experience could then query the Prolog instance to obtain derived facts about the encoded infrastructure. Nonetheless, we found that requiring a user to know or learn Prolog to use the tool would be too intimidating, and that a small specification language would be more approachable. Since the most recent versions of TOSCA use YAML as a base language, we developed a YAML sub-language to express logic specifications: we also expected the final version of the DOML to be a YAML-based language, thus a similarly structured specification language would have been immediate to embed. The language had a tree structure, in which the leafs of the trees corresponded to

facts in the database, and the intermediate nodes were constructed with logical connectors such as `and`, `or` and `not`. The full specification of the language can be found in the appendix, in Section A.2.

Listing 2.3: A specification written in the YAML specification language, matching node templates offering a certain capability, having a `password` property which is not provided using the `get_input` intrinsic TOSCA function.

```
- 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:
        - $p
        - get_input:
            args: [$_]
```

Specifications are then converted into Prolog predicates as follows:

- Scalar values are converted into the corresponding Prolog values;
- Variables (beginning with '`$`' followed by a lowercase letter or underscore) are converted into Prolog variables;
- Variables that are interpolated in the specification description, which is used to generate the output of the query, become arguments of the resulting predicate;
- `and`, `or` and `not` connectors are converted in the corresponding connectors in Prolog,

respectively ‘,’, ‘;’ and ‘\+’;

- Atomic formulas are converted into the relative predicate;
- Custom predicate invocations are also converted into the relative predicate;
- `match` atomic formulas are converted into unifications.

Listing 2.4: The Prolog predicate generated from the specification in listing 2.3.

```
hardcoded_password(X) :-
    node(X, NodeType, Props0, _, _),
    subset([property('password', P)], Props0),
    type_offers_capability(NodeType, 'tosca.capabilities.
       Endpoint.Database'),
    \+ (P = get_input(_)).
```

Further implementation details can be seen in Appendix A.

2.1.3. Evaluation and shortcomings

When profiled, the Prolog prototype showed promising results for scalability. When run on the included example topology, which included 26 infrastructural elements between top-level and imported elements, and querying the three included example checks, on an Intel i7-7700HQ machine with 16 GB of RAM, the tool showed running times below one second. Furthermore, profiling showed that the most time-consuming operations were loading third-party libraries and parsing the TOSCA topology; the time involved in querying the Prolog instance for each result were in the order of magnitude of 10^{-5} seconds.

Considering the development process, Prolog lent itself well to fast prototyping, sharing the same immediateness as many untyped programming languages: knowledge bases such as the one we are considering are easily encoded in free terms and predicates relating them.

Prolog is a Turing-complete language: while this fact implies that any computable search can be potentially performed using it, it also implies that, without imposing structural constraints on the predicates being used, we cannot guarantee termination. The predicates resulting from the encoding of the specification language we developed are not recursive, so queries involving them are bound to terminate, but in order to inspect properties which

are recursive in their structure, such as node type inheritance relationships, the definition of recursive auxiliary predicates is required.

As a logical back-end, Prolog was ultimately discarded due to the difficulty in encoding quantifiers, and most importantly, due to its *negation-as-failure* semantics [12, sections 4.4 and 4.5]. Particularly, in our case, negation-as-failure meant that a user querying $\neg(X = 3)$ would receive a negative answer instead of, *e.g.*, a satisfactory model having $X = 4$. We found this behaviour to be counterintuitive with respect to the results an inexperienced user might expect, from a tool that allows them to express properties as logical formulas.

2.2. The TOSCA Z3 prototype

Our dissatisfaction with Prolog's negation-as-failure semantics led us to investigate logical back-ends which would allow us to express properties in a more familiar logical language.

2.2.1. Satisfiability Modulo Theories, SMT-LIB and Z3

Satisfiability Modulo Theories (SMT) [6] is an approach in deciding the satisfiability of sets of first-order formulas. First-order logic is undecidable in general[3]. However, one can consider the satisfiability of formulas belonging to a sublogic of first-order logic (*e.g.*, the quantifier-free fragment), evaluated in a fixed theory (*e.g.*, integer linear arithmetic). For some combinations of sublogic and theory, decidability is recovered.

Several SMT solvers have been developed, each one supporting different logics and theories, and implementing different algorithms, resulting in diverging solving performances on specific problems. In order to provide a unified interface with a consistent set of features, the SMT-LIB standard [4] was developed. Solvers implementing the SMT-LIB standard support a subset of a set of standard logic/theory combinations, and accept SMT problems specified in a DSL syntactically similar to a LISP language. The DSL features type-like constructs called *sorts*, which are assigned to each term, and correspond to sets in the underlying theory.

Starting with version 2.6 of the standard [5], the language supports the declaration of algebraic data types, which, used in their simplest form, allowed us to declare *enumeration types*. In the following SMT-LIB code, a sort is declared that allows only three values, and then an uninterpreted function operating on this sort is declared and constrained so that its image is always different from its argument:

[3]This was proved by A. Church [10] in 1936.

Listing 2.5: Example SMT-LIB code.

```
(declare-datatypes ((Colour 0)) (((red) (green) (blue))))
(declare-fun f (Colour) Colour)
(assert (forall ((x Colour)) (distinct (f x) x)))
```

Among the available options, Microsoft's Z3 [21] is a state-of-the-art SMT solver supporting version 2.6 of the SMT-LIB standard. Besides supporting the SMT-LIB interface, Z3 also provides APIs for several programming languages, including a Python interface, which is widely used. The following Python code describes the same SMT problem as listing 2.5:

```
import z3

colour_sort, (red, green, blue) = z3.EnumSort("Colour", ["red
    ", "green", "blue"])
f = z3.Function("f", colour_sort, colour_sort)
solver = z3.Solver()
x = z3.Const("x", colour_sort)
solver.append(z3.ForAll([x], f(x) != x))
```

The following additional lines check the satisfiability of the specified SMT problem and, if a satisfying model is found, print it to standard output:

```
is_sat = solver.check()
print(is_sat)
if is_sat == z3.sat:
    print(solver.model())
```

The output follows:

```
sat
[f = [red -> green, else -> red]]
```

`sat` means that the problem is satisfiable, while `[f = [red -> green, else -> red]]`

is the found model, containing a satisfying interpretation for the declared `f` function.

2.2.2. Representing IaC within Z3

Infrastructural elements were encoded as follows:

- Node types were encoded as the finite sort `NodeType`, also containing special value `none`;
- Node templates were encoded as the finite sort `Node`, also containing special value `none`;
- Node property names were encoded as the finite sort `NodeProp`;
- Strings found in the topology were mapped onto string symbols and encoded as the finite sort `StringSym`;
- TOSCA intrinsic functions are encoded as the finite sort `FuncSym`;
- Property values and lists of values were encoded with the mutually recursive algebraic data types `Val` and `List_Val`, with the following inductive definitions:

```
inductive Val :=
| int Int
| str StringSym
| float Real
| list List_Val
| func FuncSym List_Val
| none

inductive List_Val :=
| nil
| cons Val List_Val
```

- A function named `node_type` with signature `Node` → `NodeType` was declared, and constrained to relate nodes with their node type;
- A function named `node_prop` with signature `Node` → `NodeProp` → `Val` was declared, and constrained to assign nodes and property names to the relative property value;

As was the case with the previously presented prototype, although a user with basic knowledge of SMT techniques could operate on the generated model to gather the desired

information manually, we decided to develop a small querying language. This time, we decided to deviate from the YAML structure, and developed a more traditional free-space DSL, whose grammar is described in detail in Section B.2 of the appendix. The language featured a number of primitives mapping the constructs used to encode the target IaC code in an SMT problem.

2.2.3. Evaluation

During our work on this second prototype, a first definition of the DOML was reached. This prompted us to abandon the prototype to work on a model checker that would target the DOML instead of TOSCA, which we adopted as a temporary target while the main language was not available. For this reason, in its current state, the Z3 TOSCA prototype is incomplete in its representation of infrastructural elements, and the specification language is rather inexpressive.

To evaluate the prototype's performance, the same example topology as the one included in the Prolog prototype was used. The running time was once again below the second, with most of the time being Python startup and imports loading. The times needed to construct the infrastructural model representation, encode the specification and solve the constraints were all in the order of magnitude of 10^{-2} seconds. Due to the incomplete nature of the prototype, these results must be taken with care, as the solving time would probably increase considerably with the complexity of the model representation.

3 | DOML-MC: a model checker back-end for DOML

In this chapter, the *DOML Model Checker* (DOML-MC) is introduced. DOML-MC is a DOML model checker back-end, targeting a provisional representation of the DOML, encoding attributes of the described infrastructural elements and associations between them in the Z3 SMT solver. The solver can then be leveraged to verify properties regarding the constructed knowledge base, or to find ways in which the DOML model can be extended in order to satisfy such properties. Implementation details about DOML-MC can be found in Appendix D.

3.1. The chosen state of the DOML

At the time of this work, a final version of the DOML has not been defined. Still, in order to achieve an operable version of the model checker, some form of DOML needs to be fixed. Within the scope of WP3, the team responsible for *Infrastructural Code Generation* (ICG) proposed a JSON-based language to represent DOML models, that would be more feasible to parse outside the IDE framework (Eclipse[1]) chosen as a font-end interface to end users. For an example demonstrating a small DOML model represented in this format, refer to Appendix C.

Listing 3.1: Two application-layer elements as described in the DOML JSON format.

```
[
    {
        "typeId": "application_SoftwarePackage",
        "name": "wordpress",
        "consumedInterfaces->postgres": [
            "db_interface"
        ],
```

[1] https://www.eclipse.org/

```
        "exposedInterfaces": []
    },
    {
        "typeId": "application_DBMS",
        "name": "postgres",
        "exposedInterfaces": [
            {
                "typeId": "application_SoftwareInterface",
                "name": "db_interface",
                "endPoint": "5432"
            }
        ]
    }
]
```

At about the same time, an abstract specification of the DOML was produced by WP3 [13, Annex to D3.1 – DOML Specification]. This can be used as a metamodel describing the structure of DOML models in terms of *DOML elements*, each belonging to a *class* among those listed in the specification. The classes are arranged in a subclassing hierarchy, and for each class there is a list of *attributes* that pertain to its elements, and *associations* relating elements with other elements. Attributes and associations come with *multiplicities* similar to those found across the Unified Modeling Language (UML).

In this version of the DOML, a model is divided in three layers: the *application* layer, the *infrastructure* layer and the *concretization* layer. In the application layer, the model is given in terms of its most abstract components, such as software components and external Software-as-a-Service (SaaS), each providing and/or consuming a number of software *interfaces*, defining the operational relationships between different components.

In the infrastructure layer, the model is described in terms of infrastructural components, such as virtual machines, networks, storage, but in a format that is still independent of the final CSPs involved.

In the concretization layer, one or more *concretizations* are provided. Each concretization is an infrastructural specification describing the concrete infrastructural components as offered by the CSPs, corresponding to the elements presented in the infrastructure layer, as well as information regarding which CSP provides which elements, and CSP-specific metadata.

Amid these layers, *deployments* describe how elements map onto other elements in other layers, as is for software components needing to be deployed onto virtual machines, or in the same layer, as is for containers needing to be hosted onto virtual machines.

It must be noted that this JSON format was presented as an abstract proof-of-concept, once again without a real specification, and that no software targeting it as a source for ICG was ever actually developed. As a consequence, the provided examples had not faced the validation that would have resulted from being automatically translated by ICG software, and showed several criticalities. Among these, they did not express part of the information specified in [13] for DOML models, such as the deployments. This will be exploited further in this chapter, in Section 3.4.1, as the developed tool will be shown capable of filling missing parts of the specification.

3.2. Metamodel and Intermediate Model

3.2.1. The metamodel and its YAML representation

As previously discussed, [13] specifies a class-based metamodel, in which DOML elements belong to classes featuring attributes and associations. These features could be inherited by a class from a superclass. This metamodel can be used as a basis to represent DOML models in an *intermediate model*, serving as a starting point for the process representing the model in the context of an SMT problem.

The metamodel is used to devise a simplified representation, in which the structure of DOML models is flattened to a network of associations between elements. Additionally, the metamodel can be used to derive constraints that must be satisfied by all DOML models, and constitute both a baseline verification step to apply to complete models, and a set of guides for the SMT solver for the task of model synthesis.

In DOML-MC, the metamodel is encoded as a YAML document, a format that can be easily parsed by the software. For each class, the representation encodes class-superclass relationship, as well as class attributes with the type of data they must be assigned to, and class associations with the class of elements they must point to. Attributes can also have default values, and both attributes and associations have multiplicity bounds. Classes are referred to by their qualified name, which is obtained from their name by prepending a prefix among `commons_`, `application_`, `infrastructure_` and `concrete_`, in order to disambiguate classes with the same name in different layers (for example, both the infrastructure and the concrete layers have a `Network` class). Attribute and association names are referred to by their mangled name, which is obtained by prefixing

the qualified name of the class they belong to, followed by '::', to their name (as in `application_SoftwarePackage::exposedInterfaces`); again, this serves the purpose of disambiguating attributes and associations with the same names featured in distinct classes.

Listing 3.2: A DOML class encoded in the YAML metamodel document.

```
SoftwarePackage:
    superclass: application_ApplicationComponent
    attributes:
        isPersistent:
            type: Boolean
            multiplicity: "1"
            default: false
    associations:
        exposedInterfaces:
            class: application_SoftwareInterface
            mutiplicity: "0..*"
        consumedInterfaces:
            class: application_SoftwareInterface
            mutiplicity: "0..*"
```

Occasionally, [13] also specifies constraints for the instances of a class, but these were not encoded in this representation.

Some pairs of associations in the specification (such as `infrastructure_NetworkInterface::belongsTo` and `infrastructure_Network::connectedIfaces`) are evidently the inverse of one another. This is specified in the YAML metamodel by marking one association of the pair with the field '`inverse_of:`', assigned to the mangled name of the other association.

Network topologies are particularly relevant in specifications concerning IaC. For this reason, although they are specified to be strings in the original specification, it is convenient to represent IP addresses and address ranges as integers and integer bounds respectively. In this manner, the model checker can represent the fact that an IP address belongs to a network as the corresponding integer being within the upper and lower bounds associated with the network.

DOML-MC reads this YAML document encoding the metamodel, to obtain a convenient representation as Python objects describing each class, together with a list of tuples specifying which pairs of associations are inverses.

Listing 3.3: The Python object representing the class `application_SoftwarePackage`.

```
DOMLClass(
    name='application_SoftwarePackage',
    superclass='application_ApplicationComponent',
    attributes={
        'isPersistent': DOMLAttribute(
            name='isPersistent',
            type='Boolean',
            multiplicity=('1', '1'),
            default=False
        )
    },
    associations={
        'exposedInterfaces': DOMLAssociation(
            name='exposedInterfaces',
            class_='application_SoftwareInterface',
            multiplicity=('0', '*')
        ),
        'consumedInterfaces': DOMLAssociation(
            name='consumedInterfaces',
            class_='application_SoftwareInterface',
            multiplicity=('0', '*')
        )
    }
)
```

3.2.2. Converting DOML into the intermediate model

Comparing the JSON-based DOML format and the described metamodel, one can imagine representing the target DOML model as an *intermediate model* based on the metamodel, consisting in a set of *elements* belonging to classes and presenting attributes and associ-

ations between them, in conformance with the specification in [13]. DOML-MC performs this representation, obtaining a set of Python objects corresponding to the elements in the model.

Listing 3.4: The Python object representing the software package `wordpress` in the example in Appendix C.

```
DOMLElement(
    name='wordpress',
    class_='application_SoftwarePackage',
    attributes={'commons_DOMLElement::name': 'wordpress'},
    associations={
        'application_SoftwarePackage::consumedInterfaces': {
            'postgres_db_interface'
        },
        'application_SoftwarePackage::exposedInterfaces': set
           ()
    }
)
```

In compiling this intermediate model, if two elements are associated through an association that is defined to have an inverse in the metamodel, DOML-MC automatically reciprocates the relationship in the inverse association.

3.3. Z3 representation

As was shortly demonstrated in Section 2.2.1, an SMT problem can be formulated providing the following components:

- *Sorts*, which can be thought of as sets or types of elements appearing in the sought model;
- *Functions*, including nullary functions or *constants*, operating between elements of the declared sorts;
- *Assertions*, stating logical facts, concerning the available functions, that must hold in the sought model.

Given a problem, the SMT solver can check its satisfiability: if the problem is satisfiable,

the solver can then produce a model consisting of an *interpretation* for each declared function. Additionally, assertions in the problem can be *tracked*, *i.e.* labelled; if the problem is found to be unsatisfiable, the solver can provide an *unsatisfiable core* of the problem, *i.e.* a subset of assertions that is sufficient to observe unsatisfiability. Of this subset, the tracked assertions are then returned. Z3 in particular does not guarantee the found unsatisfiable core to be *minimal*, hence there could be proper subsets of the found unsatisfiable core that are sufficient for unsatisfiability.

3.3.1. Sorts and functions

DOML-MC formulates an SMT problem, representing the constructed metamodel and intermediate model with SMT constructs as follows:

- The `Class` sort is a finite-valued sort representing classes in the metamodel;
- The `Attribute` sort is a finite-valued sort representing attributes in the metamodel;
- The `Association` sort is a finite-valued sort representing associations in the metamodel;
- The `Element` sort is a finite-valued sort representing elements in the intermediate model;
- String values in the intermediate model are mapped to atomic *string symbols*, represented with the finite sort `StringSym`;
- Attribute data are mapped to values of the tagged union sort `AttributeData`, having three constructors:
  - `int` : `Int` → `AttributeData`, for integer attribute values;
  - `bool` : `Bool` → `AttributeData`, for boolean attribute values;
  - `ss` : `StringSym` → `AttributeData`, for string attribute values;
- The function `elem_class` : `Element` → `Class` relates DOML elements to their classes;
- The function `attribute` : `Element` → `Attribute` → `AttributeData` → `Bool` encodes the relation between elements, attributes and attribute values: `attribute`(`elem`, `attr`, `ad`) is interpreted with `true` if and only if the DOML element represented with `elem` has the attribute represented with `attr` assigned to an attribute value represented with `ad`;

- The function `association : Element → Association → Element → Bool` encodes the relation between elements, associations and other elements: `association(elems, assoc, elemt)` is interpreted with `true` if and only if the DOML element represented with `elems` is associated, through the association represented with `assoc`, to the DOML element represented with `elemt`;

3.3.2. Metamodel assertions

From now on, the following conventions are employed:

- For every class, element, attribute, association or attribute value ξ, ρ_ξ denotes a constant evaluated to the above described representation of ξ;
- For every two classes C_1 and C_2, $C_1 <: C_2$ indicates that C_1 is a subclass of C_2. In particular, $<:$ is the reflexive, transitive closure of the inverse of the relation specified in the metamodel relating each class to its direct `superclass`. Note that this means that a class has always at least itself as a subclass, i.e. for every class C in the metamodel, $C <: C$ holds.

After the declaration of sorts and functions representing DOML entities, a series of assertions is added to the SMT problem to ensure that found models are coherent with the constraints in the metamodel.

- For every class C in the metamodel and for every attribute A defined for C, the following is asserted:

$$\forall(e_s : \texttt{Element})(d : \texttt{AttributeData}).$$
$$\texttt{attribute}(e_s, \rho_A, d) \rightarrow \left(\bigvee_{C' <: C} \texttt{elem\_class}(e_s) = \rho_{C'} \right) \wedge \mathit{TypeCond}$$

  In the above, $\mathit{TypeCond}$ is defined according to the value type τ_A specified for A in the metamodel:

  - If τ_A is `Boolean`, $\mathit{TypeCond} := \mathtt{q}_{\mathtt{bool}}(d)$;
  - If τ_A is `Integer`, $\mathit{TypeCond} := \mathtt{q}_{\mathtt{int}}(d)$;
  - If τ_A is `String`, $\mathit{TypeCond} := \mathtt{q}_{\mathtt{ss}}(d)$;
  - If τ_A is an enumeration type, $\mathit{TypeCond} := \bigvee_{v \in \tau_A} d = \rho_v$, where $v \in \tau_A$ signifies that v is a string representing one of the possible values of τ_A;

  For every datatype constructor c, $\mathtt{q}_c$ is a function, called a tester for c, constrained to

evaluate to truth if and only if its argument was constructed through c. In solvers complying with the SMT-LIB 2.6 standard, as in Z3, testers are automatically generated when datatypes are declared.

This assertion is added to the SMT problem to ensure that if an element has a value for A in the sought model, the element belongs to a subclass of C, and the attribute value is of type τ_A.

The assertion is tracked with the label '`attribute_st_types` A';

- For every class C in the metamodel and for every attribute A defined for C, if the lower bound on the multiplicity defined in the metamodel for A is 1, the following is asserted:

$$\forall(e_s : \texttt{Element}).$$
$$\left(\bigvee_{C'<:C} \texttt{elem\_class}(e_s) = \rho_{C'}\right) \rightarrow \exists(d : \texttt{AttributeData}).\texttt{attribute}(e_s, \rho_A, d)$$

This assertion is added to the SMT problem to ensure that if an element belongs to C, or a subclass of C, the element has at least an attribute value for A.

The assertion is tracked with the label '`attribute_mult_lb` A';

- For every class C in the metamodel and for every attribute A defined for C, if the upper bound on the multiplicity defined in the metamodel for A is 1, the following is asserted:

$$\forall(e_s : \texttt{Element})(d, d' : \texttt{AttributeData}).$$
$$\texttt{attribute}(e_s, \rho_A, d) \wedge \texttt{attribute}(e_s, \rho_A, d') \rightarrow d = d'$$

This assertion is added to the SMT problem to ensure that every element has at most an attribute value for A, *i.e.*, concretely, any two attribute values that are related to any single element through A are equal.

The assertion is tracked with the label '`attribute_mult_ub` A';

- For every class C in the metamodel and for every association A defined for C, the

following is asserted:

$$\forall(e_s, e_t : \texttt{Element}).$$
$$\texttt{association}(e_s, \rho_A, e_t)$$
$$\rightarrow \left(\bigvee_{C' <: C} \texttt{elem\_class}(e_s) = \rho_{C'} \right) \wedge \left(\bigvee_{C' <: C_A} \texttt{elem\_class}(e_t) = \rho_{C'} \right)$$

In the above, C_A is the class specified in the metamodel for target elements of A.

This assertion is added to the SMT problem to ensure that if an element is associated to a target element through A, the element belongs to a subclass of C, and the target element belongs to a subclass of C_A.

The assertion is tracked with the label '`association_st_classes` A';

- For every class C in the metamodel and for every association A defined for C, if the lower bound on the multiplicity defined in the metamodel for A is 1, the following is asserted:

$$\forall(e_s : \texttt{Element}).$$
$$\left(\bigvee_{C' <: C} \texttt{elem\_class}(e_s) = \rho_{C'} \right) \rightarrow \exists(e_t : \texttt{Element}).\texttt{association}(e_s, \rho_A, e_t)$$

This assertion is added to the SMT problem to ensure that if an element belongs to C, or a subclass of C, the element is associated with at least another element through A.

The assertion is tracked with the label '`association_mult_lb` A';

- For every class C in the metamodel and for every association A defined for C, if the upper bound on the multiplicity defined in the metamodel for A is 1, the following is asserted:

$$\forall(e_s, e_t, e'_t : \texttt{Element}).\texttt{association}(e_s, \rho_A, e_t) \wedge \texttt{association}(e_s, \rho_A, e'_t) \rightarrow e_t = e'_t$$

This assertion is added to the SMT problem to ensure that every element is associated to at most an element through A, *i.e.*, concretely, any two elements that are targets of A for any single source element are equal.

The assertion is tracked with the label '`association_mult_ub` A';

- For every pair of associations A_1 and A_2 that are defined to be mutual inverses in the metamodel, the following is asserted:

$$\forall(e_s, e_t : \texttt{Element}).\texttt{association}(e_s, \rho_{A_1}, e_t) \leftrightarrow \texttt{association}(e_t, \rho_{A_2}, e_s)$$

  This assertion is added to the SMT problem to ensure that A_1 and A_2 behave as inverses in the sought model, *i.e.* for all e_s and e_t, A_1 associates e_s to e_t if and only if A_2 associates e_t to e_s.

  The assertion is tracked with the label '`association_inverse` A_1 A_2'.

3.3.3. Intermediate model assertions

Another set of assertions is used to encode the intermediate model, derived from the DOML document, in the SMT problem that is being constructed:

- For every element E in the intermediate model, the following is asserted:

$$\texttt{elem\_class}(\rho_E) = \rho_{C_E}$$

  In the above, C_E is the class to which E belongs according to the intermediate model.

  This assertion is added to the SMT problem to ensure that the found interpretation for `elem_class` reflects the fact that E belongs to C_E.

  The assertion is tracked with the label '`elem_class` E C_E'.

- For every element E_s in the intermediate model, the following is asserted:

$$\begin{aligned}&\forall(a : \texttt{Attribute})(d : \texttt{AttributeData}).\\ &\quad \texttt{attribute}(\rho_{E_s}, a, d) \leftrightarrow \bigvee_{A \in Attrs(E_s)} a = \rho_A \wedge d = \rho_{Val(E_s,A)}\end{aligned} \tag{3.1}$$

  In the above, $Attrs(E_s)$ denotes the set of attributes defined for E_s in the intermediate model, while $Val(E_s, A)$ denotes the value assigned to attribute A for element E_s. It should be noted that the disjunction of zero clauses is taken to mean falsehood.

  This assertion is added to the SMT problem to ensure that the found interpretation for `attribute` reflects the attributes declared for the elements in the intermediate

model.

The assertion is tracked with the label '`attribute_values` E_s'.

- For all elements E_s and E_t in the intermediate model, the following is asserted:

$$\forall(a : \texttt{Association}).\texttt{association}(\rho_{E_s}, a, \rho_{E_t}) \leftrightarrow \bigvee_{A \in Assoc(E_s, E_t)} a = \rho_A \tag{3.2}$$

In the above, $Assoc(E_s, E_t)$ denotes the set of associations through which E_s is associated with E_t in the intermediate model. Again, it should be noted that the disjunction of zero clauses is taken to mean falsehood.

This assertion is added to the SMT problem to ensure that the found interpretation for `association` reflects the associations declared between the elements in the intermediate model.

The assertion is tracked with the label '`associations` E_s E_t'.

3.4. Usage

The so-constructed SMT problem can be solved as-is to check that the provided DOML document is compatible with the constraints derived from the metamodel. If the answer of Z3 is not `sat`, the unsatisfiable core is useful to understand what issues in the analysed DOML model lead to unsatisfiability.

As an example, in the intermediate model derived from the example in Appendix C, one could try to delete the attribute `application_SoftwareInterface::endPoint` set for the element `postgres_db_interface`. Checking the satisfiability of the generated problem with Z3 results in `unsat`, with an unsatisfiable core comprising the following labels:

```
elem_class postgres_db_interface
   application_SoftwareInterface
attribute_mult_lb application_SoftwareInterface::endPoint
attribute_values postgres_db_interface
```

Referring back to Section 3.3, the labelled assertions can be interpreted as "the fact that element `postgres_db_interface` belongs to class `application_SoftwareInter-`

face, the lower bound on the multiplicity of attribute application_SoftwareInterface::endPoint and the attributes provided for postgres_db_interface are incompatible".

3.4.1. Additional constraints and model synthesis

The assertions derived from the metamodel constrain the model on an abstract, coarse and structural level, but they are still compatible with situations that do not correspond to realistic infrastructural configurations. For example, a satisfiable model could have a network interface shared between two infrastructural elements.

The above instance can be fixed to adding to the SMT problem an additional assertion, such as the following:

$$\begin{aligned}\forall(e_1, e_2, i : \texttt{Element}).(&\texttt{association}(e_1, \rho_{A_{CN}}, i) \vee \texttt{association}(e_1, \rho_{A_S}, i)) \\ &\wedge (\texttt{association}(e_2, \rho_{A_{CN}}, i) \vee \texttt{association}(e_2, \rho_{A_S}, i)) \\ &\rightarrow e_1 = e_2\end{aligned} \tag{3.3}$$

In the above, A_{CN} is the association infrastructure_ComputingNode::ifaces, while A_S is the association infrastructure_Storage::ifaces.

Another assertion that will be referenced further is the following:

$$\begin{aligned}\forall(i, n : \texttt{Element}).&\texttt{association}(i, \rho_{A_{Nb}}, n) \\ &\rightarrow \exists(a : \mathbb{Z}).\texttt{attribute}(i, \rho_{A_{Ne}}, \texttt{int}(a)) \\ &\wedge (\forall(b : \mathbb{Z}).\texttt{attribute}(n, \rho_{A_{Nl}}, \texttt{int}(b)) \rightarrow b \leq a) \\ &\wedge (\forall(b : \mathbb{Z}).\texttt{attribute}(n, \rho_{A_{Nb}}, \texttt{int}(b)) \rightarrow a \leq b)\end{aligned} \tag{3.4}$$

In the above, A_{Nb} is the association infrastructure_NetworkInterface::belongsTo, A_{Ne} is the attribute infrastructure_NetworkInterface::endPoint, A_{Nl} is the attribute infrastructure_Network::address_lb, and A_{Nu} is the attribute infrastructure_Network::address_ub.

Assertion (3.4) states that if a network interface belongs to a network, its address is within the bounds delimiting the address range of the network.

Adding additional assertions, constricting the found model to represent realistic infrastructure, is particularly useful when the SMT problem construction performed by DOML-MC

Listing 3.5 The Python code, written using the Z3 Python API, adding assertion (3.3) to the SMT problem, and tracking it with label `iface_uniq`. `assoc` is a Python dictionary relating A to ρ_A for every association A.

```
e1, e2, ni = Consts("e1 e2 i", elem_sort)
assn = ForAll([e1, e2, ni],
    Implies(
        And(
            Or(
                assoc_rel(
                    e1,
                    assoc["infrastructure_ComputingNode::
                       ifaces"],
                    ni
                ),
                assoc_rel(
                    e1,
                    assoc["infrastructure_Storage::ifaces"],
                    ni
                ),
            ),
            Or(
                assoc_rel(
                    e2,
                    assoc["infrastructure_ComputingNode::
                       ifaces"],
                    ni
                ),
                assoc_rel(
                    e2,
                    assoc["infrastructure_Storage::ifaces"],
                    ni
                ),
            )
        ),
        e1 == e2
    )
)
solver.assert_and_track(assn, "iface_uniq")
```

is leveraged to attempt *model synthesis*. This is achieved by allowing more elements to the sorts described in Section 3.3.1 than the assertions in Section 3.3.3 constrain.

In particular, notice that assertions (3.1) and (3.2) constrain attributes and associations only for E_s and E_t over which the relative assertion construction procedures iterate. Hence, a number of *unbound* values can be added to the construction of the `Element` sort, which are not iterated over in the procedures asserting (3.1) and (3.2) for elements specified in the intermediate model. The interpretations of functions `attribute` and `association` are not specified when these unbound values appear as arguments.

As an example, the following assertion can be added:

$$\begin{aligned}
&\forall(s_p, s_c, i_s : \texttt{Element}).\texttt{association}(s_p, A_{Se}, i) \wedge \texttt{association}(s_c, A_{Sc}, i) \\
&\quad \rightarrow \exists(n : \texttt{Element}). \\
&\quad (\exists(c, d, i_n : \texttt{Element}).\texttt{association}(d, A_{Ds}, s_p) \wedge \texttt{association}(d, A_{Dt}, c) \\
&\qquad\qquad \wedge \texttt{association}(c, A_{Ci}, i_n) \wedge \texttt{association}(i_n, A_{Nb}, n)) \\
&\quad \wedge (\exists(c, d, i_n : \texttt{Element}).\texttt{association}(d, A_{Ds}, s_p) \wedge \texttt{association}(d, A_{Dt}, c) \\
&\qquad\qquad \wedge \texttt{association}(c, A_{Ci}, i_n) \wedge \texttt{association}(i_n, A_{Nb}, n))
\end{aligned} \tag{3.5}$$

In the above:

- A_{Se} is the association `application_SoftwarePackage::exposedInterfaces`;
- A_{Sc} is the association `application_SoftwarePackage::consumedInterfaces`;
- A_{Ds} is the association `commons_Deployment::source`;
- A_{Dt} is the association `commons_Deployment::target`;
- A_{Ci} is the association `infrastructure_ComputingNode::ifaces`;
- A_{Nb} is the association `infrastructure_NetworkInterface::belongsTo`.

Assertion (3.5) can be interpreted as "any two elements, that are connected through a software interface, must be deployed onto computing nodes that are connected through network interfaces to the same network". Note that for elements instantiating the bound variables in the assertion, the specified associations together with the assertions in Section 3.3.2 are sufficient to constrain the class they belong to. For instance, n must be a target for association `infrastructure_NetworkInterface::belongsTo`, so it must belong to a subclass of class `infrastructure_Network`.

In Section 3.1, it was noted that the JSON format of the DOML, that was chosen as a temporary target for DOML-MC, does not allow specifying how elements are deployed onto other elements. In the metamodel, deployments are specified as elements belonging to the class `Deployment` in the common namespace. Each deployment is associated through association `source` to the source element, and through association `target` to the target element.

To perform a synthesis experiment, after loading the DOML document in Appendix C, elements `unbound0` and `unbound1` were added to the `Element` sort, and (3.5) was asserted. In the example document, element `wordpress` uses a software interface provided by element `postgres`, so one expects to have the two elements deployed on computing nodes that see each other through a common network. Z3 was able to find a satisfying model, and when checking the associations involving the added unbound elements, the following tuples are found:

```
unbound0 commons_Deployment::source postgres
unbound0 commons_Deployment::target wpvm
unbound1 commons_Deployment::source wordpress
unbound1 commons_Deployment::target wpvm
```

Each line is in the format "<*source element*> <*association*> <*target element*>". The solver "deployed" both software elements on the same virtual machine, which is a possible solution to the given constraints.

3.5. Performance evaluation

3.5.1. Metamodel assertions

The DOML metamodel redacted for the following evaluations contains 35 classes, and the subclass relation among classes is composed of 92 tuples. In total, the classes in the metamodel feature 24 attributes and 46 associations. For every attribute, DOML-MC generates 3 assertions, so a total of 72 attribute related assertions are added to and tracked by the solver. In turn, for every association, DOML-MC generates 3 assertions, and moreover an assertion is generated for each pair of mutually inverse associations, so a total of 142 association related assertions are added to and tracked by the solver. Hence, the metamodel contributes 214 assertions to the solver state before the target DOML model is analysed.

3.5.2. Benchmark on DOML models

Table 3.1 portrays the results of a benchmark performed on DOML-MC on an Intel i7-7700HQ machine with 16 GB of RAM executing macOS 10.15. The benchmark is performed on four models, labelled M1 to M4. The SMT solving results are averages over 20 iterations for each configuration.

The model labelled **M1** is available in the project source as `example_json_models/wordpress_json_example.doml`. It is also the model portrayed in Appendix C.

The model labelled **M2** is available in the project source as `example_json_models/-wordpress_json_no_iface.doml`. It is obtained from M1 by removing network interfaces `wpvm_niface` and `dbvm_niface`, in order to test the solver's ability to generate a network interface in addition to the deployments, in order to satisfy assertion (3.5).

The model labelled **M3** is available in the project source as `example_json_models/nginx-openstack_v2.doml`. It is a minimal model that was obtained by translating a DOML model, produced in the scope of an integration between the various WPs, into the JSON provisional format. No synthesis was performed on the model, instead in the two last solving tests only the additional assertions were added.

The model labelled **M4** is available in the project source as `example_json_models/POSIDONIA.doml`. It is a larger model provided as a first example for the provisional DOML JSON format.

The **Construction statistics** section of the table contains numerical measures regarding the complexity of the intermediate model parsed from the JSON DOML model. The metrics are:

- **No. of elements** is the number of constructed DOML elements;
- **No. of attributes** is the total number of attributes declared for the DOML elements;
- **No. of associations** is the total number of associations declared between the DOML elements;
- **No. of string symbols** is the total number of strings extracted from both the DOML model and the metamodel;
- **No. of IM assertions** is the total number of assertions constructed from the intermediate model, which are then added to and tracked by the SMT solver.

The following table sections report the results for five different solving configurations.

Table 3.1: DOML-MC benchmark results.

	M1	M2	M3	M4
	Construction statistics			
No. of elements	16	14	11	49
No. of attributes	22	18	14	66
No. of associations	24	18	17	54
No. of string symbols	19	17	13	55
No. of IM assertions	288	224	143	2499
	SMT solving, metamodel constraints			
Time (s)	0.76	0.47	0.25	14
Quantifier inst.	1196	551	529	9616
Conflicts	185	102	74	1662
Memory (MB)	26	28	28	72
	SMT solving, assertions			
Time (s)	1.82	1.14	0.83	46.77
Quantifier inst.	3700	1645	1329	68986
Conflicts	208	146	92	1900
Memory (MB)	32	31	31	108
	SMT solving, synthesis, incremental			
Time (s)	1.88	1.56	0.23	42.43
Quantifier inst.	24043	19789	930	154338
Conflicts	1102	1143	117	6143
Memory (MB)	32	32	28	89
	SMT solving, synthesis, non-incremental			
Time (s)	2.41	1.62	0.27	50.85
Quantifier inst.	27779	17864	565	151978
Conflicts	1136	1126	85	6116
Memory (MB)	32	32	28	87
	SMT solving, synthesis, all assertions			
Time (s)	4.45	3.39	0.86	111.11
Quantifier inst.	46421	31476	1384	559239
Conflicts	1100	1250	98	6198
Memory (MB)	38	37	31	128

In **SMT solving, metamodel constraints**, the SMT problem only contains the assertions derived from the metamodel and the intermediate model.

In **SMT solving, assertions**, the problem contains the base assertions from the first configuration, as well as assertions (3.3) and (3.4).

In **SMT solving, synthesis, incremental**, after solving the problem from the first configuration, assertions (3.3) and (3.5) are added to the problem, and the solver is asked to find a satisfying configuration again. In models **M1**, **M2** and **M4**, additional values are added to the `Element` sort, in order to synthesize elements that were not specified in the original model such that the added constraints are satisfied.

In **SMT solving, synthesis, non-incremental**, the assertions used in the configuration above are added all at the same time, without performing any intermediate solving. In this configuration, solving is expected to require more time than the previous configuration, since no preliminary restriction of the search space is involved.

In **SMT solving, synthesis, all assertions**, the problem contains the base assertions used in the first configuration, as well as assertions (3.3), (3.4) and (3.5). Again, additional values are added to the `Element` sort in order to perform synthesis.

For all five configurations, the same metrics are extracted:

- **Time (s)** is the time required by Z3 to perform the solving, in seconds;
- **Quantifier inst.** is the number of times a quantified formula was instantiated during the solving;
- **Conflicts** is the number of times Z3 encountered a logical conflict while exploring the model space, resulting in a pruning of the search space;
- **Memory (MB)** is the amount of memory employed by the solving procedure, in megabytes.

It can be observed that solving time, quantifier instantiations and conflicts all increase with the number and the complexity of the assertions composing the problem. The assertions described in Section 3.4.1 are complex enough to cause solving time to increase dramatically. It must also be noted that the number of assertions derived from the intermediate model grows quadratically with the number of elements: this is due to the fact that they are structured in such a way that they do not constrain associations involving additional elements, that are not originally present in the intermediate model.

3.5.3. Considerations

A procedure verifying the same properties as the second solving configuration was implemented naively in Python, and took less than a second on model **M4**. It is evident that SMT solving is an overly-complex tool for verifying such simple properties.

On the other hand, SMT formulas like the assertions used in the benchmark are an approachable target for the translation of an IaC property specification language. Moreover, the possibility of performing model synthesis remains an interesting outcome of modelling Infrastructure-as-Code as a set of constraints in an SMT problem.

Since the performances also depend vastly on the size of the model, a possible approach for the mitigation of verification times might involve identifying parts of the model, that are independent enough to allow separate checking of most of the properties.

4 | Conclusions and future developments

The chosen approach to model checking of Infrastructure-as-Code proved to be useful for the verification of structural properties of the targeted infrastructural descriptions. Moreover, due to the model-finding capabilities of SMT solving, encoding a metamodel describing the acceptable IaC models, and the IaC model itself as an SMT problem has a dual advantage. By fully specifying the target model, one can check its coherency with the metamodel assumptions, or with any additionally specified property; by underspecifying the target model, the SMT solver can be used to complete the unspecified parts of the model, and this result can be used to resynthesize an IaC description that satisfies the provided assumptions, or to derive instructions for the user to produce the desired IaC document.

The execution times resulting from the benchmark in Section 3.5.2 show that model checking of medium-large models is not instantaneous, but model checking is traditionally known to present long execution times. For a comparison with a tool undertaking similar tasks, in [18] the Verification Tool is reported to take "seconds" for most of the models in the example repository.

Since the beginning of this work, the JSON format of the DOML originally targeted by DOML-MC has been put aside in the development of PIACERE. However, since DOML-MC works on an intermediate representation based on a metamodel that encodes an abstract specification for the DOML that should be close to final, adapting the tool to the version of the DOML that will be ultimately adopted should only involve rewriting the modules responsible for parsing the model. Similarly, by also modifying the metamodel, this work could also be adapted to target IaC languages other than DOML.

As it stands, DOML-MC is only a back-end. In order to render it operable by the end-user, some sort of user interface needs to be developed. This could be in the form of an IDE integration, being that PIACERE also focuses on the development of an IDE, and of a specification language, as was done in the prototypes described in Chapter 2. The IDE

integration should have the duty of interpreting the result of the SMT solving procedure, be it a satisfying model or an unsatisfiable core, and relate it to textual elements in the infrastructural code, possibly generating human-readable suggestions to address a negative result.

Lastly, the metamodel extracted from [13] is too abstract to ensure that synthesized models correspond to realistic infrastructure. Additional assertions ought to be added to the generated SMT problem to address this problem. An initial source for assertions can be found in the constraints specified in [13] itself, but these will likely not be sufficient. An expert user could be able to progressively refine the synthesized model by adding assertions as needed, but this possibility should be considered a last resort, for the sake of user experience.

Bibliography

[1] K. R. Apt. *Principles of constraint programming*. Cambridge University Press, 2003. ISBN: 978-0-521-82583-2.

[2] M. Artac et al. "Infrastructure-as-Code for Data-Intensive Architectures: A Model-Driven Development Approach". In: *IEEE International Conference on Software Architecture, ICSA 2018, Seattle, WA, USA, April 30 - May 4, 2018*. IEEE Computer Society, 2018, pp. 156–165. DOI: 10.1109/ICSA.2018.00025.

[3] C. Baral and M. Gelfond. "Logic Programming and Knowledge Representation". In: *J. Log. Program.* 19/20 (1994), pp. 73–148. DOI: 10.1016/0743-1066(94)90025-6.

[4] C. Barrett, P. Fontaine, and C. Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. https://smtlib.cs.uiowa.edu/. 2016.

[5] C. Barrett, P. Fontaine, and C. Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Available at https://smtlib.cs.uiowa.edu/. Department of Computer Science, The University of Iowa, 2017.

[6] C. W. Barrett et al. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. Ed. by A. Biere et al. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, pp. 825–885. DOI: 10.3233/978-1-58603-929-5-825.

[7] A. Brogi, A. Canciani, and J. Soldani. "Modelling and Analysing Cloud Application Management". In: *Proc. 4th Eur. Conf. Service Oriented Cloud Comput. ESOCC'15*. Vol. 9306. LNCS. Springer, 2015, pp. 19–33. DOI: 10.1007/978-3-319-24072-5_2.

[8] M. Burgess. *Cfengine V2.0 : A network configuration tool*. https://web.archive.org/web/20130723160143/http://www.iu.hio.no/~mark/papers/cfengine_history.pdf. Archived from http://www.iu.hio.no/~mark/papers/cfengine_history.pdf on 2016-03-04. 1993.

[9] W. Chareonsuk and W. Vatanawood. "Formal verification of cloud orchestration design with TOSCA and BPEL". In: *2016 13th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, ECTI-CON 2016* (Sept. 2016). ISBN: 9781467397490 Publisher: Institute of Electrical and Electronics Engineers Inc. DOI: 10.1109/ECTICON.2016.7561358.

[10] A. Church. "A Note on the Entscheidungsproblem". In: *J. Symb. Log.* 1.1 (1936), pp. 40–41. DOI: 10.2307/2269326.

[11] E. M. Clarke et al., eds. *Handbook of Model Checking*. Springer, 2018. ISBN: 978-3-319-10574-1. DOI: 10.1007/978-3-319-10575-8.

[12] W. F. Clocksin. *Clause and effect - Prolog programming for the working programmer*. Springer, 1997. ISBN: 978-3-540-62971-9.

[13] P. Consortium. *Deliverable D3.1: PIACERE Abstractions, DOML and DOML-E - v1*. https://www.piacere-project.eu/public-deliverables. 2021.

[14] P. Consortium. *PIACERE. Programming trustworthy Infrastructure As Code in a sEcuRE framework*. Horizon 2020 project proposal, ID: 101000162. 2020.

[15] G. J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004. ISBN: 978-0-321-22862-8.

[16] K. Jayaraman et al. *Automated Analysis and Debugging of Network Connectivity Policies*. Tech. rep. MSR-TR-2014-102. Microsoft, 2014. URL: https://www.microsoft.com/en-us/research/publication/automated-analysis-and-debugging-of-network-connectivity-policies/.

[17] K. Klai and H. Ochi. "Model Checking of Composite Cloud Services". In: *Proc. IEEE Int. Conf. Web Services, ICWS'16*. IEEE Comp. Soc., 2016, pp. 356–363. DOI: 10.1109/ICWS.2016.53.

[18] M. Law and A. Russo. *Deliverable D4.1: Constraint Definition Language*. https://radon-h2020.eu/2020/03/06/radon-constraint-definition-language-and-its-associated-vt/. 2019.

[19] J. Lepiller et al. "Analyzing Infrastructure as Code to Prevent Intra-update Sniping Vulnerabilities". In: *Proc. 27th Int. Conf. Tools Alg. for the Constr. and Anal. of Syst., TACAS'21, Part II*. Vol. 12652. LNCS. Springer, 2021, pp. 105–123. DOI: 10.1007/978-3-030-72013-1_6.

[20] K. Morris. *Infrastructure as code: managing servers in the cloud*. " O'Reilly Media, Inc.", 2016.

[21] L. M. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and J. Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[22] H. Sahli, F. Belala, and C. Bouanaka. "Model-Checking Cloud Systems Using BigMC". In: *Proc. 8th Int. Workshop Verification Eval. Comput. Commun. Syst., VECoS'14*.

Vol. 1256. CEUR Workshop Proc. CEUR-WS.org, 2014, pp. 25–33. URL: http://ceur-ws.org/Vol-1256/paper2.pdf.

[23] R. Shambaugh, A. Weiss, and A. Guha. "Rehearsal: a configuration verification tool for puppet". In: *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Des. Impl., PLDI'16*. ACM, 2016, pp. 416–430. DOI: 10.1145/2908080.2908083.

[24] H. Yoshida, K. Ogata, and K. Futatsugi. "Formalization and Verification of Declarative Cloud Orchestration". In: *Proc. 17th Int. Conf. Formal Methods Softw. Eng., ICFEM'15*. Vol. 9407. LNCS. Springer, 2015, pp. 33–49. DOI: 10.1007/978-3-319-25423-4_3.

A | Appendix: the TOSCA Prolog prototype

A.1. Implementation

A.1.1. Source code

The source code of the TOSCA Prolog prototype can be found at `https://github.com/mikidep/piacere-formal-verification/tree/master/doml_tosca_poc`.

A.1.2. External dependencies

The prototype is implemented using Python 3.9 as the programming language.

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 had to solve for our prototype to work. The fixed version of the library can be found at `https://github.com/mikidep/tosca-parser`.

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 on it 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.

Dependency management in the Python development process is achieved using Poetry (`https://python-poetry.org/`).

A.1.3. Project structure

The source code of the project is structured as follows:

- `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 auxiliary Prolog predicates;
- `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.

A.1.4. Installation and execution

As a prerequisite for the installation and execution of the prototype, Python, Poetry and SWI-Prolog must be installed on the target system. On a UNIX system, the following command installs the Python dependencies for the project:

```
$ poetry install
```

The following command executes the tool on the included example TOSCA topology:

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

The command generates the following output:

```
The required parameter db_user is not provided
The required parameter db_password is not provided
The required parameter ssh_pubkey is not provided
```

```
node webapp has unsatisfied requirement(database_endpoint,
   tosca.capabilities.Endpoint.Database, tosca.nodes.Root,
   tosca.relationships.Root, occurrences(1, unbounded))
node webapp has a requirement with num_cpus=1 and mem_size=4
   GB.
node redis has a requirement with num_cpus=1 and mem_size=16
   GB.
```

The Prolog instance can be queried further by executing the Python interpreter in its interactive mode:

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

In the so-opened Python prompt, the `prolog` object exposes a PySwip interface that can be queried:

```
>>> 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'}
>>>
```

A.2. The specification language

The Prolog PoC provides a language to query the knowledge base associated with the TOSCA topology. The language is a subset of YAML and is translated into Prolog predicates. An EBNF-like specification of the language follows: terminals are in red, non-terminals are bold, character classes are in purple and follow common regular expression syntax.

SpecDocument ::=

(- **Query**)*

A specification document 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 interpolate a number of variables and a formula to check.

ID ::= [a-z_][A-Za_z0-9_]*

Variable ::= $**ID**

Atom ::= [^\'\n]+

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** | **PolicyDef** | **CompoundTerm**

PolicyDef ::=
(- **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. 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 and contents of topology elements. 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 and contents of a node template. Type, properties, capabilities and requirements 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 similarly.

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 predicate defined outside the specification language, in the Prolog file `predicates.pl`.

NotFormula ::=
`not:` **Formula**

AndFormula ::=
`and:`
(`-` **Formula**)+

OrFormula ::=
`or:`
(`-` **Formula**)+

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.

B | Appendix: the TOSCA Z3 prototype

B.1. Implementation

B.1.1. Source code

The source code of the TOSCA Z3 prototype can be found at `https://github.com/mikidep/piacere-formal-verification/tree/z3/doml_tosca_z3`.

B.1.2. External dependencies

Similarly to the Prolog-based prototype, this prototype is implemented using Python 3.9 as a programming language, using Poetry for dependency management and the same modified version of tosca-parser to parse TOSCA topologies.

The tool uses Z3 (`https://github.com/Z3Prover/z3`) as its logical back-end, through its publicly available Python API. The solver is automatically installed together with its Python interface by Poetry.

In order to parse the specification language, the textX parser library (`http://textx.github.io/textX/`) is used.

B.1.3. Project structure

The source code of the project is structured as follows:

- `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.

B.1.4. Installation and execution

As a prerequisite for the installation and execution of the prototype, Python and Poetry must be installed on the target system. On a UNIX system, the following command installs the Python dependencies for the project:

```
$ poetry install
```

The current version of the tool offers a Python-based interactive interface. To run and inspect the results, the following lines can be input in the interpreter:

```
$ 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]]
```

B.2. The specification language

The Z3 PoC provides a language to query the knowledge base associated with the TOSCA topology. The language is a free-space Domain-Specific Language (DSL) and is translated into Z3 sort declarations and assertions. An EBNF specification of the language follows.

⟨*Spec*⟩ ::= ⟨*Statement*⟩*

⟨*Statement*⟩ ::= (⟨*NodeDecl*⟩ | ⟨*Assertion*⟩) ';'

⟨*Assertion*⟩ ::= 'assert(' ⟨*BoolExpr*⟩ ')'

⟨*Var*⟩ ::= '$' ⟨*ID*⟩

⟨*QualId*⟩ ::= ⟨*ID*⟩ ('.' ⟨*ID*⟩)*

⟨*NodeDecl*⟩ ::= '=' ⟨*Var*⟩

⟨*Ref*⟩ ::= ⟨*TypeOfRef*⟩ | ⟨*AccessRef*⟩ | ⟨*integer*⟩ | ⟨*string*⟩

⟨*TypeOfRef*⟩ ::= 'type(' ⟨*Ref*⟩ ')'

⟨*AccessRef*⟩ ::= (⟨*QualId*⟩ | ⟨*Var*⟩) ('->' ⟨*ID*⟩)*

⟨*BoolExpr*⟩ ::= ⟨*EqBoolExpr*⟩ | ⟨*NeBoolExpr*⟩

⟨*EqBoolExpr*⟩ ::= ⟨*Ref*⟩ '==' ⟨*Ref*⟩

⟨*NeBoolExpr*⟩ ::= ⟨*Ref*⟩ '!=' ⟨*Ref*⟩

⟨*ID*⟩ ::= ('A'|...|'Z'|'a'|...|'z'|'_')('A'|...|'Z'|'a'|...|'z'|'0'|...|'9'|'_')*

In the above grammar, ⟨*integer*⟩ represents an integer number, ⟨*string*⟩ matches a quoted

string.

C | Appendix: an example DOML JSON model

An example DOML model in the JSON representation used for this project follows:

```
{
    "typeId": "commons_DOMLModel",
    "name": "doml_json_example",
    "modelname": "DOML␣JSON␣Example",
    "version": "0.1",
    "id": "doml_json_example",
    "application": {
        "name": "app_layer",
        "typeId": "application_ApplicationLayer",
        "children": [
            {
                "typeId": "application_SoftwarePackage",
                "name": "wordpress",
                "consumedInterfaces->postgres": [
                    "db_interface"
                ],
                "exposedInterfaces": []
            },
            {
                "typeId": "application_DBMS",
                "name": "postgres",
                "exposedInterfaces": [
                    {
                        "typeId": "
                            application_SoftwareInterface",
```

```
                        "name": "db_interface",
                        "endPoint": "5432"
                    }
                ]
            }
        ]
    },
    "infrastructure": {
        "typeId": "infrastructure_InfrastructureLayer",
        "nodes": [
            {
                "typeId": "infrastructure_VirtualMachine",
                "name": "wpvm",
                "interfaces": [
                    {
                        "name": "wpvm_niface",
                        "belongsTo": "net1",
                        "endPoint": "10.100.1.1"
                    }
                ]
            },
            {
                "typeId": "infrastructure_VirtualMachine",
                "name": "dbvm",
                "interfaces": [
                    {
                        "name": "dbvm_niface",
                        "belongsTo": "net1",
                        "endPoint": "10.100.1.2"
                    }
                ]
            },
            {
                "typeId": "infrastructure_Storage",
                "name": "stor1",
                "interfaces": [
                    {
```

```
                        "name": "stor1_niface",
                        "belongsTo": "net1",
                        "endPoint": "10.100.1.3"
                    }
                ]
            }
        ],
        "networks": [
            {
                "typeId": "infrastructure_Network",
                "name": "net1",
                "protocol": "TCP/IP",
                "addressRange": "10.100.1.0/24"
            }
        ]
    },
    "concretizations": [
        {
            "name": "conc_infra_1",
            "typeId": "concrete_ConcreteInfrastructure",
            "groups": [
                {
                    "typeId": "concrete_ComputingGroup",
                    "name": "VPC"
                }
            ],
            "vms": [
                {
                    "typeId": "concrete_VirtualMachine",
                    "name": "conc_wpvm",
                    "description": "t3a.large",
                    "maps": "wpvm"
                },
                {
                    "typeId": "concrete_VirtualMachine",
                    "name": "conc_dbvm",
                    "description": "m5a.large",
```

```
                "maps": "dbvm"
            }
        ],
        "storages": [
            {
                "typeId": "concrete_Storage",
                "name": "conc_stor1",
                "maps": "stor1"
            }
        ],
        "networks": [
            {
                "typeId": "concrete_Network",
                "name": "conc_net1",
                "maps": "net1"
            }
        ],
        "providers": [
            {
                "typeId": "concrete_RuntimeProvider",
                "name": "AWS",
                "description": "The␣Selected␣Runtime␣
                  Provider",
                "supportedGroups": [
                    "VPC"
                ],
                "providedVMs": [
                    "conc_wpvm",
                    "conc_dbvm"
                ],
                "storages": [
                    "conc_stor1"
                ],
                "providedNetworks": [
                    "conc_net1"
                ]
            }
```

```
                ]
            }
        ]
}
```

D | Appendix: DOML-MC implementation details

D.1. Implementation

D.1.1. Source code

The source code of DOML-MC can be found at `https://github.com/mikidep/doml-mc`.

D.1.2. External dependencies

DOML-MC is implemented using Python 3.9 as a programming language and Poetry for dependency management.

The tool uses Z3 (`https://github.com/Z3Prover/z3`) as its logical back-end, through its publicly available Python API. The solver is automatically installed together with its Python interface by Poetry.

The library NetworkX (`https://networkx.org/`), a Python library for graph and network analysis, is used to manipulate graphs encoding the subclass hierarchy in the DOML metamodel.

The tool also uses the PyYAML library (`https://pyyaml.org/`) for parsing the YAML document encoding the DOML metamodel.

D.1.3. Project structure

The source code of the project is structured as follows:

- `assets`: directory containing auxiliary files:
  - `doml_meta.yaml`: YAML document encoding the DOML metamodel;
- `doml_mc`: main module directory:

- `intermediate_model`: submodule containing class definitions for intermediate model and metamodel, as well as code converting the parsed DOML model into the intermediate model:
  - `application2im.py`: code converting the application layer into the intermediate model;
  - `concrete2im.py`: code converting the concretization layer into the intermediate model;
  - `doml_element.py`: intermediate model class definitions;
  - `doml_model2im.py`: entrypoint code converting the parsed DOML model into the intermediate model;
  - `infrastructure2im.py`: code converting the infrastructure layer into the intermediate model;
  - `metamodel.py`: metamodel YAML document parsing, class definitions and related functions;
  - `types.py`: type aliases;
- `model`: submodule parsing the DOML JSON provisional format into a Python object structure:
  - `application.py`: application layer class definitions and parsing;
  - `concretization.py`: concretization layer class definitions and parsing;
  - `doml_model.py`: DOML model class definitions and parsing entrypoint;
  - `infrastructure.py`: infrastructure layer class definitions and parsing;
  - `optimization.py`: optimization layer class definitions and parsing;
- `z3`: submodule encoding metamodel and intermediate model in Z3:
  - `im_encoding.py`: procedures and functions encoding the intermediate model in Z3;
  - `metamodel_encoding.py`: procedures and functions encoding the metamodel in Z3;
  - `types.py`: type aliases;
  - `utils.py`: utility functions and procedures to work with Z3;

  - `_utils.py`: utility functions;

- `example_json_models`: directory containing example models in the DOML JSON provisional format;
- `.flake8`: configuration file for the Flake8 Python linter;
- `.gitignore`: configuration file for the Git version control system;
- `example.ipynb`: Jupyter Notebook file exemplifying the usage of the main module;
- `poetry.lock`: file required by the packaging system;
- `pyproject.toml`: settings file for the packaging system;

D.1.4. Installation and execution

As a prerequisite for the installation and execution of the prototype, Python and Poetry must be installed on the target system. On a UNIX system, the following command installs the Python dependencies for the project:

```
$ poetry install
```

An example usage of the DOML-MC back-end can be found in the Jupyter Notebook named `example.ipynb`. To load a different DOML document, replace the path assigned to the variable `doml_document_path`.

List of Acronyms

API Application Programming Interface

ASP Answer Set Programming

CDL Constraint Definition Language

CSP Cloud Service Provider

DOML DevOps Modelling Language

DOML-MC DOML Model Checker

DSL Domain-Specific Language

IaC Infrastructure-as-Code

ICG Infrastructural Code Generation

OASIS Organization for the Advancement of Structured Information Standards

PIACERE Programming trustworthy Infrastructure As Code in a sEcuRE framework

SaaS Software-as-a-Service

SMT Satisfiability Modulo Theories

TOSCA Topology and Orchestration Specification for Cloud Applications

UML Unified Modeling Language

VT Verification Tool

WP Work Package

Listings

List of Tables

List of Notations

Notation	Description
ρ_ξ	SMT constant representing infrastructural entity ξ
$C_1 <: C_2$	class C_1 is a subclass of class C_2

Acknowledgements

I am deeply grateful for all the people who worked to develop the knowledge and tools that make Computer Science what it is today, and for those who are currently researching and working to push it further.

In particular, I would like to thank my advisor Matteo Pradella, and my co-advisor Michele Chiari, who is working with us on PIACERE, for the guidance and insights they offered me in the development of this work.

I must also thank my family, who has been encouraging and supporting me in pursuing knowledge ever since I have been able to hold a book.

Lastly, I am very grateful for my friends, for those who have known me since I began my studies, and for those who I have met along the way, who provided confrontation for my ideas and kept my morale up during these last peculiar years.