Formal Verification of Infrastructure as Code [*]

Michele De Pascalis[1][2]

[1] Tallinn University of Technology, Tallinn, Estonia
michele.de@taltech.ee
[2] Politecnico di Milano, Milan, Italy
michele.depascalis@mail.polimi.it

Abstract

Infrastructure-as-Code 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

1 Introduction

Infrastructure-as-Code (IaC) is an infrastructure management and software deployment methodology that has become prevalent in the industry, shortly after the spread of cloud computing. Describing computing infrastructure in a formal language with determined semantics, this approach renders an array of techniques available for infrastructural code, techniques that were originally developed targeting source code of software programs, in the scope of software development.

Among such techniques are those inspecting the code to perform automatic formal verification, including *model checking*. Works such as Jayaraman et al. [5], Brogi et al. [1], Chareonsuk and Vatanawood [2], Shambaugh et al. [8], Lepiller et al. [7], Yoshida et al. [9], adopted model checking technologies to target IaC, focusing on operational and security properties such as idempotency, provisioning schedule validity and exposure to vulnerabilities during operation. Law and Russo [6] provided an example of checking logical properties concerning the described infrastructure, through a purposefully-developed *Constraint Definition Language* (CDL).

In this work, we explore the methodologies for performing model checking of declarative and structural properties of infrastructure described through IaC. We target the DevOps Modelling Language (DOML) [3], an IaC language developed within the scope of the PIACERE project. *Programming trustworthy Infrastructure As Code in a sEcuRE framework* (PIACERE) [4] is a research project in software engineering, funded as part of the European Union Horizon 2020

[*]This extended abstract is based on a Master's Thesis supervised by Matteo Pradella and Michele Chiari (Politecnico di Milano).

programme, aiming to develop new methodologies for IaC. This involves the development of an IaC language (DOML), as well as tools supporting its usage, such as, among others, an IDE, a tool to translate it into existing executable IaC languages, and a set of tools to verify the validity and safety of the described infrastructure.

After evaluating the fitness of logical back-ends such as Prolog, a logic programming language, and Z3, an SMT solver, we develop DOML-MC, a model checker back-end parsing IaC written in a JSON internal format of the DOML, and encoding it into an SMT problem.

2 DOML-MC: a model checker back-end for DOML

DOML-MC is a tool encoding DOML documents in an SMT problem: by adding a set of assertions, this makes it possible to use an SMT solver such as Z3 to verify properties of the modelled infrastructure, or to complete the model to obtain a model that satisfies such properties.

The format of the DOML used as a target for the tool is a provisional JSON format, that was proposed by the PIACERE team responsible for infrastructural code generation. This format was later abandoned, but the results accomplished in the development of the tool apply to all versions of the DOML that are designed following the specification in [3] more or less closely.

2.1 SMT representation

The specification in [3] was used to derive a *metamodel* for the DOML, which describes an infrastructure in terms of *elements* belonging to *classes*. Classes are related through class-subclass relationships. Elements have *attributes* and are related among themselves through *associations*. An element is allowed to have a certain attribute, or to be the source for a certain association, if these appear in the definition of its class, or a superclass of its class. For attributes and associations, *multiplicity* bounds can be specified, *e.g.*, if an association has an upper bound of 1 on its multiplicity, each element can have at most an element associated to it through such association. The metamodel was encoded in a machine-readable YAML format.

Tracing this metamodel, in order to represent infrastructural information in the SMT problem, finite sorts are created for elements, classes, attributes and associations. An additional sort encodes the string values found in the DOML document as *string symbols*; this sort is embedded, together with the sorts for integers and booleans, in a tagged union sort to represent attribute values. A function is declared to relate elements to their classes, one to relate elements and attributes, and one to relate elements to elements through associations. Then, a set of assertions, ensuring that the interpretations for these functions are coherent with the metamodel, is added to the SMT problem.

The target DOML document is parsed and translated to an *intermediate model* based on the metamodel. This is used to provide the values for the sort of elements, and to derive a set of assertions constraining the values of the declared functions to match the described infrastructure.

2.2 Usage

Z3 can be used to solve the generated SMT problem as-is to ensure its coherency with the metamodel. The added assertions are tracked with unique labels, so that, in case of a negative answer, Z3 can provide a set of reasons that is sufficient to observe incoherency. In order to verify additional properties, special assertions can be added to the SMT problem after the base construction above.

Moreover, by inserting additional values in the sort of elements, which are not constrained by the assertions generated from the intermediate model, Z3 can find interpretations for the declared functions that are compatible with the metamodel assertions, or with any additional assertion. This capability can be exploited to perform model synthesis.

2.2.1 An example property

Let A_{CN} stand for the association `infrastructure_ComputingNode::ifaces`, and let A_S stand for the association `infrastructure_Storage::ifaces`. Then the following assertion ensures that a network interface cannot be shared between two infrastructural elements:

$$\begin{aligned}\forall(e_1, e_2, i : \texttt{Element}).(&\texttt{association}(e_1, \rho_{A_{CN}}, i) \lor \texttt{association}(e_1, \rho_{A_S}, i))\\ &\land (\texttt{association}(e_2, \rho_{A_{CN}}, i) \lor \texttt{association}(e_2, \rho_{A_S}, i))\\ &\to e_1 = e_2\end{aligned} \tag{1}$$

The property can be encoded as an SMT assertion. As an example, consider a topology with two virtual machines `wpvm` and `dbvm`, sharing a network interface `wpvm_niface`: DOML-MC can be used to generate an SMT problem encoding the metamodel, the topology, and assertion (1). When Z3 is run on the problem, it returns `unsat`, meaning that the constraints given by the metamodel, the description of the topology and the added assertion are not satisfiable together. When asked for an *unsatisfiable core*, Z3 provides the following list of labels:

```
[associations wpvm wpvm_niface,
associations dbvm wpvm_niface,
iface_uniq]
```

2.3 Performance evaluation

DOML-MC was evaluated with four DOML documents, testing its capability to verify the basic coherency with the metamodel, and its ability to enrich a model in order to satisfy additional properties. This was performed both in two distinct solving procedure executions, and as a cumulative execution, to test the hypothesis that the incremental solving of the enriched problem performs better than solving the cumulative problem *ex novo*.

The largest DOML document that was used presented 49 elements, 66 attributes and 54 associations. Over 20 iterations, the solving procedure took 14 seconds to verify metamodel coherency, 42.43 seconds to perform model synthesis with additional assertions incrementally, and 50.85 seconds to perform it non-incrementally.

3 Conclusions and future developments

The chosen approach to model checking of IaC 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 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 [6] the developed verification tool is reported to take "seconds" for most of the models in the example repository.

The structure of the metamodel is flexible enough to allow for the metamodels of different infrastructural representations to be adapted to be compatible with DOML-MC. It could thus be worthwhile to attempt to reuse its intermediate model to encode and analyse different IaC languages.

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 above.

Lastly, the metamodel extracted from [3] 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 [3] itself, but these will likely not be sufficient.

References

[1] 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.

[2] 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.

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

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

[5] 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/.

[6] 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.

[7] 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.

[8] 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.

[9] 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.