POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

Formal Verification of Infrastructure as Code

Laurea Magistrale in Computer Science and Engineering

Author: Michele De Pascalis

Advisor: Prof. Matteo Pradella

Co-advisor: Dr. Michele Chiari

Academic year: 2020-2021

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 K. Jayaraman et al. [8], A. Brogi et al. [3], W. Chareonsuk and W. Vatanawood [4], R. Shambaugh et al. [12], J. Lepiller et al. [10], H. Yoshida et al. [13], 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. M. Law and A. Russo [9] 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) [6], an IaC language developed within the scope of the PIACERE project. *Programming trustworthy Infrastructure As Code in a sEcuRE framework* (PIACERE) [7] is a research project in software engineering, funded as part of the European Union Horizon 2020 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 tool to verify the validity and safety of the described infrastructure.

Firstly, we develop two prototypes targeting TOSCA (*Topology and Orchestration Specification for Cloud Applications*, an open standard IaC language), evaluating the fitness of logical back-ends such as Prolog, a logic programming language, and Z3, an SMT solver. We then 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. Approach

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 (see [5]). Considering that our work targets declarative and structural properties rather than operational ones, we did not attempt to model IaC as a system with states and transitions among them. Instead, we concentrated on a set of techniques known as *constraint programming*.

2.1. The TOSCA Prolog prototype

Prolog is a logic programming language. In logic programming [1], a program is specified by providing a set of implications between predicates applied to terms, thus composing a knowledge base. The knowledge base is then queried to verify that a fact holds, or to find satisfying assignments for a set of variables in a fact.
The Prolog-based prototype parses a given TOSCA topology template and generates facts derived from the various TOSCA entities it involves, *i.e.* node types, capability types, node templates, etc. Each fact encodes details about the involved TOSCA entities, such as property and requirement definitions for node types, properties and satisfied requirements for node templates.
Some auxiliary predicates are then added to the knowledge base, entered through a Prolog file. These are useful to specify complex relationships between entities, such as the type-supertype relationship between node types, which is specified as a recursive predicate.
The knowledge base constructed by the tool can then be queried through a YAML-based language, featuring variables, base terms corresponding to the existence of TOSCA entities, a special base term for unification, and compound terms in which base terms are composed by a logical connective, such as '`and`', '`or`' and '`not`'.
The tool was found to be efficient on a small TOSCA topology template. However, the Prolog approach was discarded due to the fact that negation in Prolog is regulated by a semantic rule named "*negation as failure*". This behaviour is counterintuitive with respect to the semantics of negation in Boolean and classical first-order logic, with which an end user without experience in logic programming is expected to be more familiar.

2.2. The TOSCA Z3 prototype

With the intention to provide a specification interface that could be readily understood by a user familiar with classical logic, Satisfiability Modulo Theories (SMT) [2] was selected as a candidate logical back-end representation. SMT problems are a generalization of propositional satisfiability (SAT) problems, extending them to the language of first-order logic, or sub-logics of it, and searching model with respect to (modulo) first-order theories for which model-finding procedures exist. For some combinations of sub-logic and theory, such as the quantifier-free fragment of the logic with the theory of real arithmetic, there are decision procedures that are guaranteed to terminate: this is not true for first-order logic in general.
Z3 [11] is a state-of-the-art SMT solver developed by Microsoft. It supports the most recent capabilities in the scope of SMT solving, and provides a convenient Python API that allows developers to construct the target SMT problem procedurally.
In the TOSCA Z3 prototype, the TOSCA topology template to be analysed is parsed, and its TOSCA entities (node types, node templates, etc.) are encoded as finite-valued *sorts*. A sort is an SMT construct that can be interpreted to be a set in ordinary first-order logic. The relationships between various entities, such as the node type for a given node template, or the value of a given property for a given node template, are encoded with functions between sorts, another construct that is made available by Z3.
Again, a small Domain-Specific Language (DSL) was developed as an interface for the user to query the model, completing the constructed SMT problem. Z3 is then used to find a satisfying instance by providing compatible interpretations for functions and unbound constants.
The prototype was only tested on the same small topology template, showing core solving times in the order of magnitude of $\times 10^{-2}$ seconds. However, these results must be taken with care, since the prototype cannot be considered complete: its development was interrupted when the PI-ACERE team working on infrastructural code generation proposed an operable format for the

DOML, prompting us to begin working on a tool that would target it directly.

3. 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 [6] more or less closely.

3.1. SMT representation

The specification in [6] 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.

3.2. 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.

3.3. 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.

4. 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 [9] 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 [6] 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 [6] itself, but these will likely not be sufficient.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

[8] 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/.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.