A Tool for Verification of Big-Data Applications

Marcello M. Bersani, Francesco Marconi, Matteo Rossi
DEIB, Politecnico di Milano, Italy
{firstname.lastname}@polimi.it

Madalina Erascu
Institute e-Austria Timisoara & West University of Timisoara, Timisoara, Romania
merascu@info.uvt.ro

ABSTRACT

Quality-driven frameworks for developing data-intensive applications are becoming more and more popular, following the remarkable popularity of Big Data approaches. The DICE framework, designed within the DICE project (www.dice-h2020.eu), has the goal of offering a novel profile and tools for data-aware quality-driven development. One of its tools is the DICE Verification Tool (**D-VerT**), which allows designers to evaluate their design against safety properties, such as reachability of undesired configurations of the system. This paper describes the first version of **D-VerT**, available open source at github.com/dice-project/DICE-Verification.

CCS Concepts

•**Software and its engineering** → **Formal software verification;**

Keywords

Formal verification; temporal logic.

1. INTRODUCTION

Big Data is a prominent area that researches innovative solutions to support the entire life-cycle of data-intensive applications (DIAs), i.e., software and hardware infrastructures able to process huge amounts of information over frameworks which require high computational power. The area gained considerable relevance in the recent years, promoted by the pervasive spread of applications like, for instance, Facebook or Twitter.

Defining methodologies and frameworks for the development of DIAs leveraging Big Data technologies is nowadays fundamental. The DICE project [1] tackles this issue by defining techniques and tools for the data-aware quality-driven development of DIAs. In the DICE approach, designers model DIAs through UML diagrams tagged with suitable annotations capturing the features of Big Data applications, and in particular their *topology*. A topology provides an abstract representation of a DIA through directed graphs, where nodes are of two kinds: *computational nodes* implement the logic of the application by elaborating information and producing an outcome; *input nodes* bring information into the application from the environment. The semantics underlying the topology typically changes depending on the target Big Data technology.

The DICE Verification Tool (**D-VerT**) checks whether a given topology reaches an unwanted configuration; i.e., whether it allows for bad executions that do not conform to some non-functional requirements. It focuses on the analysis of the effects of the incorrect design of timing constraints which might cause the following anomalies: *(i)* latency in processing the information; and *(ii)* monotonic growth of the size of used memory.

This paper presents the architecture of the **D-VerT** tool, the verification approaches it supports, and some experiments carried out for applications based on Apache Storm (storm.apache.org).

2. VERIFICATION TOOL OVERVIEW

Verification (see Fig. 1) is performed on annotated UML models which contain all the necessary information related to a topology. To this end, **D-VerT** receives a DTSM (DICE Technology Specification Model, "DICE-profiled Model" in Fig. 1), a UML diagram which captures, through ad-hoc stereotypes, the technological aspects of a DIA platform that the designer is going to adopt for deploying the application. At the DTSM level, the designer specifies the topology and all the relevant parameters that are needed for the verification, e.g., the number of processes that can be run in a computational node or the emit rate of the source nodes. The designer also selects a property to be checked, using suitable templates in the main IDE of the DICE framework.

The DTSM annotated model and the property to be verified are converted into a formal model that is suitable for verification. **D-VerT** is designed to support different verification approaches which are tailored to express various kind of properties. Based on the property to verify and on the type of model the user specifies, **D-VerT** creates through a series of transformations a file containing the respective formal model ("Logic Model" in Fig. 1), selects the appropriate solver and runs the verification. All the parameters defining the verification approach to use and, possibly, other configuration settings related to the model-checker are selected by the user on the IDE when specifying the running configuration. This "configuration model" is provided as input to **D-VerT** along with the DICE-profiled UML Model. The outcome produced by the solver is interpreted and translated so that it can be sent back to Verifier-GUI which, finally, outputs the result. In particular, **D-VerT** shows whether the property holds or not and, if the property is violated, it presents a trace of the system that violates it.

2.1 Tool Architecture

D-VerT is composed of three modules, as shown in Figure 1.

- **DTSM2Json** converts DTSM diagrams into an intermediate

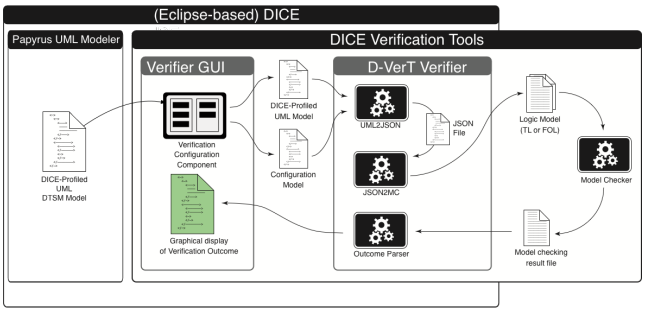

Figure 1: Verification workflow

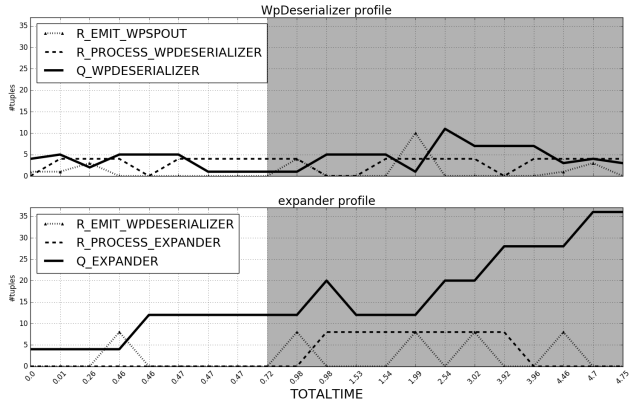

Figure 2: Example of D-VerT output trace

description of the topology specified in a JSON file.

- **Json2MC** instantiates the semantics of the DIA – specified in the DTSM diagram and encoded into a JSON file – in a file containing the formal model expressed in either temporal logic (TL) or first order logic (FOL).
- **Outcome Parser** processes the result provided by the solver and shows it in a graphical form on the IDE.

Each topology, specified through a DTSM diagram, is encoded into a JSON object by the **DTSM2Json** component. **DTSM2Json** loads the annotated UML file and extracts the relevant model features based on the DICE profile. These features, plus the configuration parameters provided by the user, are serialized in a JSON file.

Json2MC has been designed to be extensible and configurable. It is composed of a core component, *Model Configurator*, and a set of *model templates*, which embed the syntax and semantics of the different models that have to be produced to run the formal verification. *Model Configurator* reads the topology description encoded in JSON format and instantiates the formal model by rendering the selected template, according to the input configuration.

The result of the verification task can be a "counter-example" trace describing a computation that violates the desired property, or the message no such trace exists. When the tool returns a counter-example trace, the designer needs to inspect it to better understand the system behaviour, hence what led to the violation. To make the output of the tool more user-friendly, the **Outcome Parser** module assists the user by providing a graphical interpretation of the result. Figure 2 provides an example of output produced by the tool. Each of the two plots refers to a specific computation node.

2.2 Verification Approaches

D-VerT supports two verification approaches based on two distinct logical formalisms. Extending the tool to other formalisms is however easy, thanks to the **D-VerT** architecture that allows parametric model-to-model transformations to be configured by means of templates. We here summarize the approaches in **D-VerT**.

Bounded satisfiability checking. Given two temporal logic formulae, one modelling the temporal behaviour of a topology, and one capturing the property to be analysed, **D-VerT** checks whether it is possible to satisfy the conjunction of the former with the *negation* of the latter (i.e., whether there is an execution that violates the property); if it is not possible, the tool returns "unsatisfiable" (UNSAT), otherwise it returns an execution trace, i.e., a counterexample. In the former case (UNSAT result), we can conclude that the property holds in the model. For this approach, **D-VerT** relies on the **Zot** (github.com/fm-polimi/zot) tool as verification engine.

Reachability checking. In this approach, a topology is defined through an array-based system that undergoes verification of a safety problem. A model comprises: (i) a set of system transitions and an initial configuration; (ii) a formula that defines the set of unsafe (i.e., "undesired") states. The result is either SAFE ("undesired" configurations can not be reached in the model) or UNSAFE together with an unsafe trace ("undesired" configurations can be reached in the model and the output trace shows how). For this verification approach, **D-VerT** uses **MCMT** (users.mat.unimi.it/users/ghilardi/mcmt/) and **Cubicle** (cubicle.lri.fr).

2.3 Experimental Results

We used **D-VerT**, and in particular the bounded satisfiability checking approach, to verify different topologies ranging from a simple DIA to a more complex topology (named "focused-crawler") provided by an industrial partner within the DICE consortium. In both cases, we verify the property "all queues associated with the computational nodes have a bounded occupation level". If the property holds, then we claim that all bolts are able to process the incoming information in a timely manner. A counterexample that violates (i.e., disproves) the property corresponds to an execution of the topology where at least one queue grows with an unbounded trend. This behavior can be easily expressed as a bounded satisfiability problem with a formula constraining the size of the queues.

Figure 2 shows two of the graphical output traces provided by **D-VerT** (referring to the computational nodes *expander* and *wpDeserializer* from the given topology). It can be noticed, by looking at the quantity of information stored in queues over time (black solid lines), how they both represent a periodic model in which a suffix (in gray) of a finite sequence of events is repeated infinitely many times after a prefix. After ensuring that the trace is not a spurious model, we concluded that the expander queue, having an increasing trend in the suffix, is unbounded.

We carried out experiments on different topologies with various configurations to evaluate the performance of the tool. The results are reported at dice-project.github.io/ DICE-Verification.

Acknowledgements

Work supported by Horizon 2020 project no. 644869 (DICE).

3. REFERENCES

[1] G. Casale, D. Ardagna, M. Artac, F. Barbier, E. D. Nitto, A. Henry, G. Iuhasz, C. Joubert, J. Merseguer, V. I. Munteanu, J. Pérez, D. Petcu, M. Rossi, C. Sheridan, I. Spais, and D. Vladušič. DICE: Quality-driven development of data-intensive cloud applications. In *Proc. of MiSE*, pages 78–83, 2015.