Guiding the Correction of Parameterized Specifications

Jean-François Couchot\textsuperscript{1,2} and Frédéric Dadeau\textsuperscript{3}

\textsuperscript{1} INRIA Futurs, ProVal, Parc Orsay Université, F-91893
\textsuperscript{2} LRI, Univ Paris-Sud, CNRS, Orsay F-91405
\textsuperscript{3} Lab. d’Informatique de Grenoble, BP. 72, Saint-Martin d’Hères F-38402

Abstract. Finding inductive invariants is a key issue in many domains such as program verification, model based testing, etc. However, few approaches help the designer in the task of writing a correct and meaningful model, where correction is used for consistency of the formal specification w.r.t. its inner invariant properties. Meaningfulness is obtained by providing many explicit views of the model, like animation, counter-example extraction, and so on. We propose to ease the task of writing a correct and meaningful formal specification by combining a panel of provers, a set-theoretical constraint solver and some model-checkers.

1 Introduction

When designing safe softwares, writing formal specifications is a hard but valuable task, since consistency between the program and its inner properties can be translated into Proof Obligations (POs) that can then be discharged into some more or less automated prover.

The B method \cite{B方法} ranges in this scope, by providing a formal development framework. It starts from often parameterized abstract specifications, called abstract machines, that are later refined until a parameter-free implementation is obtained. Intuitively, an abstract machine is composed of state variables, an initialisation operation, some operations that modify the values of state variables and an invariant that represents properties that must hold for each state of the execution.

Invariant verification techniques can be divided into two main categories: model-checking (Spin \cite{Spin}, SMV \cite{SMV}), which exhaustively explores the states of the execution graph, and deductive approaches, based on automatic provers (Simplify \cite{Simplify}, Yices \cite{Yices}, haRVey \cite{havvey}) or interactive ones (COQ \cite{COQ}, PVS \cite{PVS}, HOL \cite{HOL}). Intuitively, model-checking aims at checking whether all reachable states satisfy a given property, which is then called an invariant. On the other hand, deductive approaches try to show that a property is inductive, i.e., established by initialisation and preserved through each operation. The B method is conceived as a deductive approach, in which the generated POs
are discharged into a set-theoretical prover $^{16,12}$. Some model-checkers for $\mathbb{B}$ machines have been developed, such as ProB $^{31}$ or the BZ-Testing-Tools $^{3}$ animator, but all of them require finite and enumerated data structures to work.

Each inductive property is obviously an invariant, whereas an invariant is, seldom if ever, inductive. If the invariant is not inductive, invariance proofs fail. In an interactive mode, the modeler is asked to manually end the proof, which might be useless since the errors are located in the specification. In an automatic mode, the proof may end. When it does, it generally answers “not valid”, sometimes while extracting the subformula that originates this answer. This allows the engineer to modify either the invariant, or the code of the operation, or both. When the proof does not terminate, the validity of the considered formula is unknown. Thus, the engineer is left with unknown results concerning the inductivity of the invariants he wrote.

This paper presents an original framework relying on the efficient combination of deductive and model-checking approaches, in order to be able to help the engineer in the process of designing, automatically checking the correctness and (eventually) correcting a formal specification.

This framework is depicted in Fig. 1. We first consider the translation of the formal specification into the Why language. Why $^{24}$ is a verification tool that is dedicated to build POs for a variety of provers, and runs them to perform the verification. In case of failure of all the provers, our approach aims at providing a concrete counter-examples for the induction property. This task starts with an instantiation of the parameters, by using recent results in proof techniques applied on software verification $^{26}$. It makes it possible to use CLPS $^{3}$, a set-theoretical constraint solver, to solve the instantiated constraint, which results in a counter-example if the formula is invalid. In addition, $\mathbb{B}$ animators $^{31,11}$ are employed to automatically produce an execution trace leading to this counter-example when it exists. The modeler obtains, for free, some guidance information that helps her/him to modify the incriminated part of her/his formal specification. Furthermore, in order to be able to give a feedback on the PO validity, when the proof does not terminate, or when the prover is unable to decide the theory involved within the formula, we also propose to instantiate the formula before checking its consistency with CLPS.

This paper is organized as follows. Section 2 presents the consistency checking of $\mathbb{B}$ machines, that makes it possible to establish their correction, and presents the running example that will illustrate our approach throughout the following sections. We give in Sect. 3 the translation of $\mathbb{B}$ machines into the Why language. Then, we present in Sect. 4 the instantiation techniques that are sufficient to compute a counter-example. In Sect. 5 we show how to help the designer in the task of correcting her/his specification by computing an execution trace that leads to the counter-example. Finally, we draw a comparison with some related works in Sect. 6 before we conclude and present future works in Sect. 7.