Information Processing Letters 116 (2016) 409–415

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Verifying safety properties of a nonlinear control by interactive theorem proving with the Prototype Verification System

Cinzia Bernardeschi, Andrea Domenici [*]

University of Pisa, Dept. of Information Engineering, Largo Lucio Lazzarino 1, 56122 Pisa, Italy

A R T I C L E   I N F O

Article history:
Received 21 August 2015
Received in revised form 16 December 2015
Accepted 8 February 2016
Available online 9 February 2016
Communicated by L. Viganò

Keywords:
Formal Methods
Theorem Proving
Verification
Nonlinear Control
Prototype Verification System

A B S T R A C T

Interactive, or *computer-assisted*, theorem proving is the verification of statements in a formal system, where the proof is developed by a logician who chooses the appropriate inference steps, in turn executed by an automatic theorem prover. In this paper, interactive theorem proving is used to verify safety properties of a nonlinear (hybrid) control system.

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

Many technical systems fall in the class of *hybrid systems*, i.e., nonlinear systems having both analog and digital components. Such systems are typically composed of an analog plant, described by linear or nonlinear equations, and a digital control, intrinsically nonlinear. In industrial practice, hybrid systems are usually analyzed by simulation. An executable model of the system is built with graphical block-based languages such as those offered by the Simulink (TM), Scilab, or ScicosLab environments [1,2], or textual languages such as Modelica [3] or CIF [4], or a combination of the two, and the model is executed to simulate the system under various conditions.

While simulation is a mainstay of system development and is a necessary tool for validation, it cannot provide developers with the confidence afforded by formal verification. Formal verification of nonlinear systems may be difficult, but automatic or semiautomatic tools can provide valuable support to this task.

Schupp et al. [5] recently published an overview of hybrid systems verification, with short outlines of tools and techniques for reachability analysis, examples of benchmark problems, and current challenges. A survey of works on formal verification of hybrid systems was published by Alur [6], who identifies some broad areas of research, including *symbolic reachability analysis* and *deductive verification*.

In the area of symbolic reachability analysis, research is focused on algorithms to compute or approximate a system's *reach*(*ability*) *set*, i.e., the set of states reachable from any of the admissible initial states, with the goal of verifying whether the set contains unsafe states. For example, Tiwari and Khanna [7] propose techniques to approximate reach sets for different classes of hybrid automata, based on qualitative abstraction [8], which in turns relies on model checking. Model checking is also used by Cimatti et al. [9], who implement a quantifier-free encoding of hybrid automata with the NuSMV [10] model checker.

[*] Corresponding author. Tel.: +39 050 2217674; fax: +39 050 2217600.
E-mail address: andrea.domenici@ing.unipi.it (A. Domenici).

http://dx.doi.org/10.1016/j.ipl.2016.02.001
0020-0190/© 2016 Elsevier B.V. All rights reserved.

Many tools have been developed to support the analysis of hybrid systems, including UPPAAL [11] for timed automata, HybridSAL [12] based on the SAL [13] model checker, ARIADNE [14], and HSOLVER [15]. In particular, ARIADNE has been used for nonlinear hybrid system verification based on an assume-guarantee method, and HSOLVER has been applied to safety verification with constraint propagation and abstraction refinement.

In the area of deductive verification, KeYmaera [16] is an interactive theorem-proving environment based on sequent calculus and tailored to the differential dynamic logic $d\mathcal{L}$ [17]. KeYmaera has been developed specifically for hybrid systems, unlike other general-purpose theorem provers, such as Coq [18], based on the calculus of inductive constructions and intuitionistic logic, and Isabelle [19], based on higher-order logic and functional programming.

In this paper, the PVS (Prototype Verification System) theorem prover is used to prove basic properties of a typical case study, the level control of a storage tank. This simple example shows that a higher-order theorem-proving tool can support developers in expressing and verifying a natural line of reasoning rooted on domain knowledge.

This paper is structured as follows: In Section 2, essential information on the PVS language and deduction system is provided; Section 3 introduces the case study; Section 4 describes the formalization of the case study and how the PVS is used to prove that certain constraints guarantee safe operation of the system; Section 5 discusses the case study and relates it to the general topic of hybrid system analysis; and Section 6 concludes the paper.

2. The Prototype Verification System

The PVS is an interactive theorem prover developed at Computer Science Laboratory, SRI International, by S. Owre, N. Shankar, J. Rushby, and others [20,21] and it has been applied to many fields, including formal verification of hardware and safety-critical systems [22–24]. Its formal system is based on sequent calculus [25–27], together with a typed higher-order language.

A PVS user writes a *theory* in the PVS language [28], then uses the PVS theorem proving environment [29] to prove selected formulas of the theory.

2.1. The PVS language

In a PVS theory, one can declare *types*, *constants*, *variables*, and *formulas*. The PVS type system is very flexible, providing users with standard mathematical types (e.g., naturals, integers, and reals) and allowing them to define *uninterpreted* types, to build *record* and *tuple* types similar to records in programming languages, and to define *function* types (e.g., "*the set of functions from integers to reals*"). In particular, functions returning Boolean values are called *predicates*. It is also possible to define *subtypes* by adding constraints to previously defined types. One can then declare constants and variables (including function constants and variables) and write formulas. A formula is a named logical statement composed of atomic formulas, logical connectives, and quantifiers.

Each formula is identified by a name and qualified by a keyword specifying if the formula is an *axiom* or not. The PVS prover takes axioms as proved statements, whereas it requires the other formulas to be proved. Axioms are recognized by the `AXIOM` keyword, the other formulas by such keywords as `LEMMA`, `THEOREM`, or other synonyms. Examples of PVS declarations are found in Section 4.

The PVS environment includes a large number of prepackaged fundamental theories, called the *prelude* [30]. An even larger number of theories, covering, e.g., mathematical analysis, algebra, or probability, is available in additional libraries, such as the NASA Langley PVS Library [31,32].

2.2. The PVS deduction system

As previously mentioned, PVS is based on the sequent calculus. A *sequent* is an expression with this structure:

$$A_1, A_2, \ldots, A_m \vdash B_1, B_2, \ldots, B_n$$

where the A_i's are the *antecedents* and the B_i's are the *consequents*. The '$\vdash$' symbol is called a *turnstile* and may be read as "*yields*". Each antecedent or consequent is a formula built with atomic formulas, connectives, and quantifiers.

A sequent is true if any formula occurs both as an antecedent and as a consequent, or any antecedent is false, or any consequent is true. Proving a formula (a *goal*) consists in expressing it as a sequent without antecedents and applying inference rules until one of the previous conditions for truth is met.

The PVS prover presents the user with the initial sequent corresponding to the formula to be proved. The user applies a series of inference steps, invoking a prover command at each step. A prover command may result in the application of a single inference rule of the sequent calculus, or a combination of several rules, possibly chosen and iterated according to some pre-packaged *strategy*. Some of the manipulations made available by the PVS prover include: (i) *Instantiating variables*, in particular by introducing fresh *Skolem* constants; (ii) *decomposing formulas* into simpler ones; (iii) *introducing lemmas*; and (iv) *applying substitutions*. Some commands transform the current goal into two or more *subgoals*: For example, the `split` command transforms a goal of the form $A \Rightarrow B \vdash C$ into two subgoals $B \vdash C$ and $\vdash A, C$.

Usually, a PVS user directs the proof by making informed choices about the main steps (such as introducing the appropriate lemmas) and lets the prover deal with low-level, tedious and error-prone manipulations. The prover, however, supports also high-level proof strategies, such as induction.

3. Water level control

The problem of controlling the level of a liquid (say, water) in a tank is a well-known case study in control theory. There are several versions of this problem, and in this paper the one presented in [33] is considered. The problem is stated as follows: