Abstract. This paper describes the formal verification of parallel programs using a rewrite and induction based theorem prover like LP and a higher order theorem prover based on the Calculus of Inductive Construction, namely Coq. The chosen specification environment is UNITY, a subset of temporal logic for specifying and verifying concurrent programs. By means of an example, a lift-control program, we describe the embedding of UNITY and we show how to verify mechanically program properties using the two provers. Then we summarize a comparison between the theorem proving environments, based on our practical experience with both systems for the verification of UNITY programs.

Keywords: Formal Verification, UNITY, theorem prover methodology, LARCH Prover, COQ, Computer Checked Proof.

1 Introduction

The verification of parallel programs have inspired several methodologies. One of these, UNITY introduced by Chandy & Misra [5], has been used and developed in several provers [4, 18, 6, 16, 1]. UNITY is a theory to specify and verify concurrent programs. It provides us with a formalism to express the relevant properties of a program, an appropriate language to construct well-formed formulas and a proof system to construct proofs.

The aim of the paper is not to identify the best tool to mechanically prove UNITY specifications, if such a tool exists, but to evaluate the compromises of the different approaches associated with the power of the prover by comparing the implementations of UNITY made in LP [6] and in Coq [18]. Moreover, the comparison takes UNITY as a means and not as a goal in the sense that we will discuss the implementation of the objects one could find in any embedding of a programming theory, such as state, variables and assignments.

This comparison is based on our practical experience for the verification of a lift-control program taken from [2] and already completed with the HOL-UNITY system [1]. This example is interesting because it is a good archetypical problem that allows us to show how we deal with a program manipulating naturals, booleans, and abstract data types such as arrays. As the correctness proof has
been verified independently with the Coq proof assistant and with the LARCH prover, it allows us fruitful comparisons on the use of a theorem prover with regards to the mechanization of Unity proofs.

The design and development of the LARCH proof assistant LP were motivated primarily as a means to debug LSL specifications [17] but they have also been used to establish the correctness of hardware designs [23, 7] and to reason about algorithms involving concurrency [13, 21]. It is a general-purpose theorem prover for multi-sorted first-order logic, it is based on equational term rewriting [14] and it supports proofs about axiomatic specifications. All proofs are carried out by applying rewrite rules, proofs by case splitting, induction, contradiction and application of inference rules. LP does not contain any predefined theory, but automatically declares the sort bool with the corresponding logical operators.

Coq [9] is a theorem prover based on the Calculus of Inductive Construction (CIC) which is a higher order typed calculus with inductive definitions [29]. It has been used to prove the correctness of hardware designs [10, 26] and to reason about different theories on protocols [15, 3]. The system Coq is divided into three parts. The first part is the logical language in which we write our axiomatizations and specifications. The second part is the proof assistant which allows the development of mathematical proofs. It provides several tactics to built the proof. It is important to note that the tactics implement the backward reasoning, that is the construction of the proof is made in the top-down manner. The proofs, when they are achieved, are considered as terms of λ-calculus due to the Curry-Howard isomorphism. The third part is a program extractor which allows the synthesis of a functional program from the constructive part of its proof of correctness. This last feature of Coq is not taken into account in the comparison.

The paper is organized as follows. We start by a brief description of the fundamental concepts of Unity. Next we describe the mechanization of Unity logic in both theorem provers and we point out the differences between the embeddings of the lift-control program using LP and Coq. Following that, we discuss the automation in the two provers and the facilities given by the logic.

2 Unity and the Lift-Control Program

A Unity program consists of four sections: a Declare section that declares the variables used in the program, an Initially section that describes the initial values of the variables, an Always section that is a set of shorthands, and an Assign section that consists of a non-empty set of assignment statements.

The following example is taken from [2]. It describes a lift that moves between a number of floors to serve requests on these. The bottom and top floors are specified with two constant parameters min and max.

Program {Lift(min,max)}
Declare
  floor : integer {up, move, stop, open : bool}[] req : array[min..max] of bool
Initially
  floor = min[] up, move, stop, open = false, true, true, false[] ∀i. min ≤ i ≤ max ∧ req[i] = false
Always
  above = ∃i : floor < i ≤ max ∧ req[i]