A Proof Environment for Concurrent Programs

Naïma BROWN and Dominique MERY*

CRIN-CNRS & INRIA Lorraine, BP 239
54506 Vandœuvres-lès-Nancy, France.
email: brown@loria.fr, mery@loria.fr

Abstract. Unity [CM88, Mer92, Kna90], as action systems approach [BS91], is a formal method that attempts to decouple a program from its implementation. Therefore, Unity separates logical behaviour from implementation, it provides predicates for specifications, and proof rules for deriving specifications directly from the program text. This type of proof strategy is often clearer and more succinct than argument about a program’s operational behaviour. Our research fits into Unity’s methodology. Its aims to develop a proof environment suitable for mechanical proof of concurrent programs. This proof is based on Unity [CM88], and may be used to specify and verify both safety and liveness properties. Our verification method is based on theorem proving, so that an axiomatization of the operational semantics is needed. We use Dijkstra’s wp-calculus to formalize the Unity logic, so we can always derive a sound relationship between the operational semantics of a given Unity specification and the axiomatic one from which theorems in our logic will be derived.

Automated theorem proving, concurrency, program verification, formal specifications, Unity, B

1 Introduction

In a mechanically verified proof, all proof steps are validated by a computer program called a theorem prover. Hence, whether a mechanically verified proof is correct is really a question of whether the theorem prover is sound. The theorem prover used in our research is B-Tool [BT91a, BT91b, BT91c]. B provides a platform for solving the problem specification and correct construction of software systems. It is a flexible inference engine which forms the basis of a computer-aided system for the formal construction of provably correct software. Using a mechanized theorem prover to validate a proof presents an additional burden for the user, since machine validated proofs are longer and more difficult to produce. However, if one trusts the theorem prover, one may then focus attention on the specification that was proved. This analysis may be facilitated by

* on sabbatical leave at the Department of Computing Science University of Stirling under the European Science Exchange Programme Royal Society - CNRS
consulting the mechanized proof script. The proof environment for concurrent programs presented in this paper is based on Unity, which has two important characteristics:

- Unity provides predicates for specifications, and proof rules to derive specifications directly from the program text. This type of proof strategy is often clearer and more succinct than argument about a program's operational behaviour. David Goldschlag in [Golg90] described how to mechanically verify a Unity program with the Boyer-Moore prover [BM88].
- Unity separates the concern of algorithm from that of architecture. It defines a general semantics for concurrent programs that encourage the refinement of architecture-independent programs to architecture-specific ones. In a paper presented at the European Workshops on Parallel Computing [BM92], we have already described a systematic method for mapping (implementing) a Unity specification into the programming language Occam.

2 Unity

This section is a brief discussion of the Unity notation and programming logic. The section described the logic defined by Chandy and Misra together with the modification suggested by Sanders to incorporate the notion of strongest stable predicate and weakest stable predicate.

2.1 A Programming Notation

A Unity program consists of a declaration of variables, a specification of their initial values, and a set of multiple-assignment statements. A program execution starts from a state satisfying the initial condition and loops forever; in each step of execution some assignment statement is selected non-deterministically and executed (Fig.1). Non-deterministic selection is constrained by the rule of fairness. A state of a program is called a fixed point if and only if execution of any statement of the program, in this state, leaves the state unchanged. Proofs

![Diagram](Fig.1. Execution of a Unity program)

in Unity are based on assertions of the type \( \{p\} s \{q\} \), where \( s \) represents the Unity assignment statement, \( p \) the precondition that must be validated before the execution of \( s \), and \( q \) the post-condition that results from the execution of