Understanding Concurrent Programs using Rely-Guarantee Thinking*

Ian J. Hayes, Larissa Meinicke, Robert Colvin, Kim Solin, Kirsten Winter

School of ITEE, University of Queensland, Australia

The rely-guarantee approach (developed in the 80s by Cliff Jones) is one of the first to provide compositional reasoning about concurrent programs. As a result one can reason about single threads individually when reasoning about a parallel program. Since rely-guarantee reasoning can handle fine-grained and non-blocking synchronisation a resurgence of interest in the approach can be observed.

Our focus is on the design of concurrent algorithms using a stepwise refinement from an abstract specification down to code. The refinement steps are driven by a set of rules collected in a refinement calculus. Our work builds largely on Carroll Morgan's refinement calculus for sequential programs which we extended (and adapted) by rules that allow the designer to introduce parallel behaviour in a refinement step. To do this we make use of the notions of rely and guarantee conditions in order to constrain the (allowable) interference between threads.

Our refinement calculus of rely-guarantee refinement for concurrent programs is presented algebraically, i.e., by algebraic rules in which rely and guarantee conditions are generalised to processes. This leads to surprisingly simple and elegant proofs. The rules of our calculus are mechanically proven to be correct using the Isabelle/HOL theorem prover.

The calculus makes use of a *wide-spectrum* language which is general enough to capture not only safety properties but also progress properties of the overall behaviour of a concurrent program (e.g., necessary when developing lock-free algorithms). Another possible (future) application is the development of concurrent algorithms in the light of a specific weak (software or hardware) memory model. That is, a normal specification of the algorithm could be enhanced by replacing the normal sequential execution of each thread's code by a non-deterministic choice over the possible (i.e., enabled) next steps to capture optimising code re-orderings (undertaken by the compiler) like e.g., done in the Java Memory Model (JMM). Weak hardware memory models on the other hand, can be captured by adding buffers, fences and flushes into the specification to determine where interference renders fences and flushes necessary in the concurrent algorithm.

* This work is supported by Australian Research Council (ARC) Discovery Project DP130102901.