internal course given 2 days at Oxford Summer '99

Linking Z and CSP

C.A.R. Hoare

The objective of this course is to construct a link between Z used as a specification language and CSP used as a design language for reactive computer systems. A communicating process can be specified in Z as a schema declaring the relevant observations (traces, refusals, divergences), and describing their intended properties. A sequential process, for example in occam, is one that includes initial as well as final values for these variables. Complex specifications can be built up using the operators of the schema calculus, - conjunction, disconjunction and sequential composition.

Moving to the design phase, the process of stepwise decomposition replaces unimplementable operators (like conjunction, and negation) by the combinators of an implementable programming language, in our case CSP. Disjunction and sequential composition are retained; they need to be supplemented by new schema operators for recursion, parallel composition, external choice, etc.

At each step of the design, correctness is established by proof of a logical implication between the schemas. Eventually, the whole design is expressed solely in the notations of the programming language. The interpretation of this program in the schema calculus will be the strongest specification of all observations that can be made of its execution. Thus the schemas essentially provide denotational semantics for the language.

The major risk in stepwise design is the postulation of an interface or a component that turns out to be unimplementable. The extreme example is the dreaded miracle - an unsatisfiable schema, equivalent to the predicate false, which will always be provable correct. Exclusion of a miracle is achieved by a collection of healthiness conditions, which are proved to be satisfied by every program in the language. Examples are prefix closure of traces in CSP, or totality of relations in the refinement calculus. Healthiness conditions are simply expressed as algebraic laws, and can be added to the schema calculus to help prove the properties of programs and designs. Although they are individually simple their combination both introduces and explains the inavoidable complexity of the semantics of a programming languages like occam.

The course starts with the simple trace model of CSP, and used the concept of a buffer as an example. Non-determinism is modelled by disjunction and concurrency is oversimplified as conjunction. The only healthiness conditions is prefix closure. The model can be enriched by adding alphabets. This permits the definition of prefixing, and the implementation of simple recursine processes together with their proofs of correctness.

A more realistic model of CSP introduces refusals to specify the responsiveness of a process to external signals. This permits external choice to be distinguished from non-determinism. To define sequential composition, we need an additional variable to distinguish waiting states from termination, and to distinguish deadlock from SKIP.

The final lecture shows the link between the simple specifications of Z, and the pre-condition/post-condition pairs used by VDM and other refinement calculi. Following the examples of CSP, we introduce a variable *ok* to distinguish non-divergent from divergent states. Simple healthiness conditions ensure divergence is the worst thing that can happen and that it cannot be recovered by sequential composition.

These lectures have used the schema calculus of Z to define the essential features of the reactive programming paradigm. Perhaps further advantage can be taken of the modularity and extensibility of Z schemes to define additional features and even different paradigms in a unifying framework.