Real-Time Refinement

Colin Fidge

Software Verification Research Centre,
The University of Queensland, QLD 4072, Australia

Abstract. The refinement calculus is extended for real-time system development. Heuristic rules for deriving high-level designs from Z specifications are also given. A detailed example of real-time refinement is presented.

1 Introduction

Formal methods have long been touted as a solution to industry's software development problems. Unfortunately many practical issues faced by 'real-world' software developers remain beyond the reach of formal techniques. Foremost among these are the 'hard real-time' requirements of applications with precise and inviolate timing constraints. Although numerous real-time specification and programming notations have been proposed, practical design and development methodologies for real-time systems are lacking.

This paper extends the refinement calculus to make it suitable for derivation of real-time systems. This is done by adding time-based side-conditions to the refinement rules which must be provably satisfied for the refinement step to be valid, thus guaranteeing that the resultant system adheres to its timing constraints. Furthermore, a set of heuristic rules is given for directly converting from Z specifications into a high-level system design.

2 Background

Our real-time refinement methodology builds on a number of established techniques.

2.1 Z

Z is a specification language for defining, and reasoning about, a wide range of systems [9]. The Z notation consists of two parts: the schema language (a specification structuring technique) and a mathematical language.

When used to define a system consisting of a number of distinct operations there are typically two distinct uses of schemata, state and operation:

<table>
<thead>
<tr>
<th>State declarations</th>
<th>Declare a number of variables and express invariants on them in the predicate.</th>
</tr>
</thead>
<tbody>
<tr>
<td>predicate</td>
<td></td>
</tr>
</tbody>
</table>
Operation declarations

Change the declared variables as per pre and post-conditions defined by the predicate.

By convention, an operation schema that includes a state schema name in its declaration part as $\Delta State$ may change the value of the variables declared in State. An undecorated variable name $v$ appearing in an operation schema refers to the value of the variable before the operation occurs and $v'$ refers to its value after the operation has been performed.

Schemata may be composed using a number of special operators. For instance, schema conjunction,

$$Op \equiv Op_1 \land Op_2,$$

is used to merge the declarations of operations $Op_1$ and $Op_2$ and take the logical 'and' of their predicates.

The mathematical notation used within schemata is based on set theory and includes numerous operators.

2.2 The Refinement Calculus

The refinement calculus is a rigorous method for deriving executable programs from abstract specifications [5]. The calculus operates on Dijkstra's guarded command language (GCL) augmented with specification statements. In this way partly refined algorithms can be expressed in a 'mixed' notation of concrete GCL code and abstract behavioural specifications.

A specification statement,

$$w: [pre, post],$$

says that, executed in a state satisfying its pre-condition, it will terminate in a state satisfying the post-condition, while modifying only those variables listed in the 'frame' $w$. In the post-condition an undecorated variable name $v$ refers to the value of the variable in the post-state and $v_0$ refers to its value in the pre-state.

System development involves repeatedly applying refinement laws, expressed using a refinement relation, denoted $\sqsubseteq$. For instance, a law for decomposing a specification statement into two sequentially composed statements is

$$w: [pre, post] \sqsubseteq w: [pre, mid] ; w: [mid, post]$$

(assuming $v_0$ variables are not used). These laws guarantee that the right-hand side is a valid 'implementation' of the left.

When considering the practicalities of real-time programming we need to extend the guarded command language slightly. Assume the existence of output$(c, E)$ and input$(c, v)$ statements for sending the value of an expression $E$ to, or receiving a value into variable $v$ from, some external channel $c$, and an idle$(t)$ statement for effecting an idle delay of exactly $t$ time units.