A CASE tool for demonstrating Z specifications

ANDREWS, Simon <http://orcid.org/0000-0003-2094-7456> and NORCLIFFE, Allan

Published version

ANDREWS, Simon and NORCLIFFE, Allan (1990). A CASE tool for demonstrating Z specifications. In: IEE Colloquium on Application of CASE Tools. IET, 5/1-5/4.

A CASE tool for demonstrating Z specifications

S Andrews and A Norcliffe

The CASE tool we describe is designed to enable software engineers to produce a faithful animation of specifications written in Z. Desirable properties which we feel animations of this kind should possess, and which have guided us in developing the tool, are the following.

1. The executable code (ie the animation) must be easy to produce.

2. The structure of the code should not be too far removed from the Z.

3. The animation should be sufficiently user friendly to enable a client to understand and interact with it, thus facilitating the process of validating a specification against user requirements.

The CASE tool is based around the program development tool known as CRYSTAL. CRYSTAL is reasonably well-known in AI circles and is sold as an expert system shell by Intelligent Environments Ltd in Richmond. It is essentially a rule-based programming language offering excellent input, output, and menu facilities, as well as all the standard features expected of any expert system shell. The specific advantages we see, that this environment offers as a means of transforming Z to executable code, are as follows.

1. The rule-based nature of CRYSTAL means that lines of Z, in the predicate of a schema, transform almost one-for-one into rules in CRYSTAL.

2. The expandable way in which rules are built up in CRYSTAL mirrors very closely the use of the schema calculus in Z. The developer, using the tool, can faithfully transform a Z specification starting at the schema level and finishing at the line-by-line predicate level.

3. The excellent user interface that comes with CRYSTAL enables the developer to concentrate his efforts on transforming Z instead of worrying about how to create a friendly user interface. This is an added bonus given the fact that implementation issues are positively avoided in formal specifications.

4. The animation that results can be viewed by the client at different levels. This is possible because of the folded nature of the rule-based programming in CRYSTAL. At the highest level a system might be viewed as a menu having several options such as

    quit
    initialise state
    save state
    load state
    print state
    test data invariants
    operation 1
    operation 2

    ⋮

    operation n

S Andrews and A Norcliffe are both members of the School of Engineering Information Technology at Sheffield City Polytechnic

Any operation chosen by the client can be systematically unfolded to discover the rules that make it work, thus promoting the vital interaction between client, developer and system that is necessary for requirements validation. In CRYSTAL this is feasible because at the highest level the rules are written in English. Only at the lowest level does English give way to code. What the client sees, therefore, is a faithful English translation of the developer's Z.

To illustrate these points a short example is now considered. The following is part of the Z specification of a very simple security system that might be in operation in a building to monitor the whereabouts of staff users. The system state consists of three subsets, in, out and users, of type P(STAFF_ID), and is represented by the following state schema

```
┌─State──────────────────────
│ in, out, users : P(STAFF_ID)
│
├────────────────────
│ in ∩ out = { }
│ in ∪ out = users
└────────────────────────────
```

Amongst other things the system checks people in and out of the building and the CheckInOK operation may be specified as follows

```
┌─CheckInOK──────────────────
│ Δ State
│ person_id? : STAFF_ID
│
├────────────────────
│ person_id? ∈ out
│ out'    = out \ {person_id?}
│ in'     = in ∪ {person_id?}
│ users' = users
└────────────────────────────
```

When the precondition is violated the CheckInOK operation will fail. A robust CheckIn operation can therefore be defined as follows

CheckIn ≙ CheckInOK ∨ CheckInError

CheckInError ≙ CheckInError1 ∨ CheckInError2

where the two error schemas are as follows

```
┌─CheckInError1──────────────────────
│ State
│ person_id? : STAFF_ID
│ message!   : REPORT
│
├────────────────────
│ person_id? ∈ in
│ message! = "Person already in building"
└────────────────────────────────────
```

```
┌─CheckInError2──────────────────────
│ State
│ person_id? : STAFF_ID
│ message!   : REPORT
├──────────────────────
│ person_id? ∉ users
│ message! = "Person is not a valid user"
└──────────────────────────────────
```

At the highest level the CRYSTAL coding for this Z could be the following

```
      CheckIn works
IF    CheckInOK works
OR    CheckInError works

      CheckInError works
IF    CheckInError1 applies
OR    CheckInError2 applies

      CheckInOK works
IF    person_id is entered into the system
AND   the person_id currently belongs to the set out
AND   the person_id is then removed from the set out
AND   the person_id is then added to the set in
AND   the set users is unchanged
AND   completion of the operation has been signalled

      CheckInError1 applies
IF    person_id is entered into the system
AND   the person_id currently belongs to the set in
AND   an appropriate message is output

      CheckInError2 applies
IF    person_id is entered into the system
AND   the person_id does not currently belong to the set users
AND   an appropriate message is printed
```

Obviously the developer has to expand each of these individual rules further until they are capable of being executed. But in principle this is a fairly straightforward task given the available CRYSTAL operations, and the fact that sets, functions, relations, sequences, power sets, bags etc can all be represented conveniently as arrays in CRYSTAL.

What we observe, then, is that CRYSTAL rules are not too far removed from Z and give a very faithful transformation of the Z. The animation at this level is also capable of being understood by a client even though he may know little or no Z. The client can thus interact with the specification through the CRYSTAL animation and can thus contribute meaningfully to the process of requirements validation.

With regard to the CASE tool, however, the following points are relevant. A major disadvantage of CRYSTAL is that although at a high level it faithfully represents the Z notation, at the lowest level the CRYSTAL code can be somewhat lengthy. For example, the CRYSTAL transformation of a function override operation could require 50 or more lines of code. This problem is further compounded by the fact that there is no parameter passing in CRYSTAL ie it is not possible to write a single routine for $\oplus$ and pass the appropriate parameters to it. The code must be repeated each time it is required. However, this problem of low-level coding can be avoided by writing a "Z-function" interface to CRYSTAL in C and work is currently in progress to create a library of Z-function routines ($\oplus$, $\cup$, $\cap$, # etc). These will eventually be amalgamated with the standard CRYSTAL function library supplied with the shell and used in the same way in the CRYSTAL code. The result should be a CASE tool that software engineers can use, with relative ease, to animate specifications written in Z.