ABSTRACT
Proper techniques and testing are two important approaches that assist in the development of high quality software while traditionally these approaches have been seen as rivals. We augment the usual wp semantics of substitution with an explicit notation of frame, which allow us to develop a simple self-contained theory of generalized substitution outside their usual context of the B method. In doing so, we gain some useful insight about the nature of substitutions, which enables us to resolve some heather to problematic issues concerning substitutions within the B method.

Keywords: Generalised Substitution Language, Software Inspection Technique.

1. INTRODUCTION
1.1 Abrial’s Generalised Substitution Language
Abrial finally succeeded in utilising the various theories that have been developed so far, into a coherent and comprehensive method for formal software development. For this purpose, Abrial created the specification languages Z and B, which both at present play major roles in the area of formal software development.

Here, the Generalised Substitution Language, abbreviated to GSL, will be presented. It provides the basic elements of the Abstract Machine Notation that is used in the specification of B machines.

1.2 Overview of GSL Constructs
An overview of GSL constructs and their semantic definition is given in the following table. The meaning of each construct is defined in terms of its weakest precondition. To conform with Abrial’s notation, the weakest precondition of a Generalised Substitution S to establish a post-condition Q is written as [S]Q.

<table>
<thead>
<tr>
<th>Construct Name</th>
<th>GSL Notation</th>
<th>Weakest Precondition [S]Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skip</td>
<td>skip</td>
<td>[skip]Q = Q</td>
</tr>
<tr>
<td>Assignment</td>
<td>x := E</td>
<td>[x := E]Q = Q&lt;\x&gt;E</td>
</tr>
<tr>
<td>Multiple Assignment</td>
<td>x, y := E, F</td>
<td>[x, y := E, F]Q = Q&lt;\x, y&gt;E, F</td>
</tr>
<tr>
<td>Pre-conditioned Substitution</td>
<td>P | S</td>
<td>[P | S]Q = P &amp; [S]Q</td>
</tr>
<tr>
<td>Guarded Substitution</td>
<td>G == S</td>
<td>[G == S]Q = G \Rightarrow [S]Q</td>
</tr>
<tr>
<td>Bounded Choice</td>
<td>S | T</td>
<td>[S | T]Q = [S]Q &amp; [T]Q</td>
</tr>
<tr>
<td>Sequential Composition</td>
<td>S, T</td>
<td>[S; T]Q = [S][T]Q</td>
</tr>
<tr>
<td>Unbounded Choice</td>
<td>@x \bullet S</td>
<td>[@x \bullet S]Q = \forall x \bullet [S]Q</td>
</tr>
</tbody>
</table>

1.3 Operational Interpretation of GSL Constructs
The Skip Substitution Abrial’s skip is essentially the same skip that Dijkstra used in his Guarded Command Language. It doesn’t change the program state and hence its weakest precondition is equivalent to the post-condition.

1.3.1 The Assignment and Multiple-Assignment Substitution
The meaning of a single assignment x := E is that after its execution x will have the value of E, where E is an arbitrary expression. Accordingly, to derive the construct’s weakest precondition to establish some post-condition Q, each x in Q has to be replaced by E. In case of a multiple-assignment, substitution is performed simultaneously. The assignment and multiple-assignment substitutions are analogously defined to Dijkstra’s assignment statements.

1.3.2 The Bounded Choice Substitution
The bounded choice substitution captures the fact that a choice between two Generalised Substitutions S and T can be made at a certain point of program execution. However, it is unknown which of them will be chosen.

Accordingly, both choices must be taken into account as a possible program behaviour, what explains the derived weakest precondition to be the conjunction between [S]Q and [T]Q.

1.3.3 The Guarded Substitution
The guarded Substitution behaves like S if the guard is true, and behaves miraculously if the guard is false. This means, under the assumptions \( P \) \& \( g \), a guarded Substitution establishes any result one may ask for.
An example is given by:

\[ [g \rightarrow S]Q = (g \varnothing [S]Q) = ([g] g \varnothing [S]Q) \]

If the guard \( g \) is false, the weakest precondition of \( g \rightarrow S \) becomes true independently of the substitution \( S \) or the post-condition \( Q \). Therefore, outside the condition \( g, g \rightarrow S \) would establish any post-conditions that we ask for - even contradictory ones.

The interpretation for this impossible language construct is to say that \( g \rightarrow S \) is not feasible outside its guard. There is no real operational interpretation of infeasibility since infeasible substitutions are not considered to be implementable; however, if a non-deterministic choice is offered two commands \( S \) and \( T \) to proceed with, we can convince ourselves that it always has to choose a feasible rather than an infeasible one, as is illustrated in the last paragraph of this section.

### 1.3.4 The Preconditioned Substitution

The Preconditioned Substitution \( P \mid S \) behaves like \( S \) inside its precondition \( P \), but aborts when invoked outside \( P \). Under the assumption of \( P \), it can lead to any behaviour, and even refuse to terminate. The idea of preconditions goes back to Dijkstra’s work. However, he used them implicitly to insist on abortion if all of the guards are false in his if-fi construct.

### 1.3.5 The Unbounded Choice Substitution

Unbounded choice represents the choice between a (possibly) infinite numbers of substitutions \( S \) which is obtained by replacing a free variable \( z \) within \( S \) by any value. A non-deterministic assignment of any number \( x \) could, for example, be written as:

\[ \@z \cdot x = z \]

In fact, Abrial didn’t include either condition or iteration constructs in his GSL. The reason is that the both can be derived from what has already been provided. The GSL is an elementary set of atomic constructs that forms the basis for more advanced constructs like conditional statements or loops.

Before Abrial, the general opinion was to consider conditional statements as atomic constructs in themselves. However, Abrial showed that they could be decomposed into even more elementary constructs: guarded substitutions and non-deterministic choice. For example:

\[ \text{if } b \text{ then } S \text{ end can be expressed in GSL as } (b \rightarrow S) \]

\[ (b \rightarrow \text{skip}) \]

In the same way, iteration could be treated by solving fixed point equations on functions that evaluate to GSL expressions. No further constructs are really needed.

### Tic-Tac-Toe Example

The reason that this example has been chosen is that it yields to a certain complexity, but yet is simple enough to be fully proven (with some help of additional inference
rules) by the Autoprover of the B-Toolkit. Hence, Noughts & Crosses is an example how a B developments could give the most possible confidence into the implementation since a mathematical proof of all specified requirements makes a testing phase, in principle, superfluous.

Even if by its popularity, most of us should be familiar with the rules of Tic-Tac-Toe (Noughts & Crosses), they will briefly be summarised in the following paragraph.

Tic-Tac-Toe is a simple board game, which is played on an array of 3x3 fields. Two players participate in the game; one will be further designated as black, and the other one as white. At the beginning of the game, all fields are empty. Then, both players in turns occupy one field of the board, which hasn’t been occupied by one of the players yet. Played on a piece of paper, black indicates his move by a nought “o” and white indicates his move by a cross “×” - therefore the name Noughts & Crosses. To be precise, we define that black is always permitted to perform the first move after the game has started. The game finally ends when one of the players has managed to arrange three of his symbols in a row, column or diagonal of three fields. If no one of the players can perform any further moves since all fields have been already occupied, and also no winner can be identified, the game ends with a draw. The following example shows a possible course of moves where black “o” will be the winner:

The implementation of a software component to play Noughts & Crosses should allow both players to perform their moves by calling an appropriate operation. It further has to offer functions to determine if one of the players has already managed to win. Last, the application should provide an operation to restart the game. The following list gives an overview of the four operations that are necessary for interaction with the Noughts & Crosses application, along with their signature:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Input parameters</th>
<th>Returned values</th>
<th>Precondition</th>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>restart</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>Re-initialisation of the application to begin a new game.</td>
</tr>
<tr>
<td>move(ff)</td>
<td>ff (field number between 1 and 9)</td>
<td>none</td>
<td>none</td>
<td>Records the move of the player who is next to doing his turn. The field on the 3x3 array is specified by the number ff.</td>
</tr>
</tbody>
</table>

In the informal specification, we didn’t include an operation to determine if a draw has occurred. We will leave this out for simplicity reasons. However, this information can be deduced from the number of operation calls to the move(ff) operation. Since there are only 9 fields to be occupied, we’ll have a draw if after 9 calls to move(ff) no player has yet managed to win the game.

2. CONCLUSION

The methods presented in this paper manifest in more repeatable, effective and practical methods of inspection and proper verification.

A development example of the game Noughts & Crosses was performed. The development has been carried out from the informal specification to the abstract B specification and finally the implementation of the component. The derivation of the machine’s invariant from the underlying model has been discussed. In connection with the implementation the use, of library machines was illustrated on the example of an array.

So finally we reach to result that Current methods of verification and defect detection are, generally, not
repeatable, and derivation of semantic information, including invariants, from program code can support and improve the repeatability of verification and inspection tasks.

REFERENCES


