LECTURE 12: Z SPECIFICATIONS & THE SCHEMA CALCULUS

Software Engineering

Mike Wooldridge

1 The Truth About Schema Inclusion

- We saw last week how, a schema could be included by just listing its name in the declarations part of a schema. We now look at what this actually *means*.
- Suppose we had the following definition:

S_1

$v_1 : T_1$
$v_2 : T_2$

P_1
P_2

and later on

S_2

S_1 (* schema inclusion *)
$v_3 : T_3$

P_3

- Then this would have been equivalent to:

$$
\begin{array}{|l}
\hline
\textbf{Schema } S_2 \\
v_1 : T_1 \\
v_2 : T_2 \\
v_3 : T_3 \\
\hline
P_1 \\
P_2 \\
P_3 \\
\hline
\end{array}
$$

- We now need to introduce *schema decoration*.
- Suppose we had the following declaration:

$$
\begin{array}{|l}
\multicolumn{1}{l}{S_3} \\ \hline
S_1' \\ \hline
P_4 \\ \hline
\end{array}
$$

then this declaration would have been equivalent to

$$
\begin{array}{|l}
\multicolumn{1}{l}{S_3} \\ \hline
v_1' : T_1 \\
v_2' : T_2 \\ \hline
\left.\begin{array}{r} P_1 \\ P_2 \end{array}\right\} \begin{array}{l} \text{with all references to } v_1, v_2 \\ \text{changed to } v_1', v_2'. \end{array} \\
P_4 \\ \hline
\end{array}
$$

- *Remember that the decorated form of a variable means "the variable* after *the operation has been performed"; the undecorated version means "the variable* before *the operation has been performed".*

- Let's now consider the Δ notation.
- Suppose we had:

$$
\begin{array}{|l}
\hline
S_4 \\
\Delta S_1 \\
\hline
P_5 \\
\hline
\end{array}
$$

- This would have been equivalent to

$$
\begin{array}{|ll}
\hline
S_4 & \\
S_1 & \text{(* include } S_1 \text{ *)} \\
S_1' & \text{(* include } S_1' \text{ *)} \\
\hline
P_5 & \\
\hline
\end{array}
$$

- The Ξ notation means something similar. Suppose we had the schema:

$$\begin{array}{|l}
\multicolumn{1}{l}{S_5} \\
\Xi S_1 \\
\hline
P_5 \\
\hline
\end{array}$$

then this would expand to

$$\begin{array}{|l}
\multicolumn{1}{l}{S_5} \\
S_1 \\
S_1' \\
\hline
P_5 \\
v_1' = v_1 \\
v_2' = v_2 \\
\hline
\end{array}$$

- So when we use the Ξ notation before a schema, it means "include the decorated and undecorated version of this schema, with the postcondition that all the variables remain unchanged."

2 The Schema Calculus

- One of the nice things about Z is that it allows us some sort of modular construction; we can build things in little pieces and put them together to make big pieces.
- The way we do this is by using the *schema calculus*.
- First we need to introduce *horizontal form schemas* (as opposed to the vertical form schemas we have been looking at so far).

- **Definition:** The following vertical-form schema

$$\begin{array}{|l}\multicolumn{1}{l}{\_\, S \,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_}\\ \mathit{Declarations} \\ \hline P_1 \\ P_2 \\ \ldots \\ P_n \\ \hline \end{array}$$

may be defined in the following horizontal form

$$S \mathrel{\widehat=} [\mathit{Declarations} \mid P_1;\ P_2;\ \cdots P_n]$$

- The symbol $\widehat=$ is for schema definition; it may be read 'is defined to be'.
- Using $\widehat=$, we can make one schema an alias for another:

$$\mathit{NewPhoneBook} \mathrel{\widehat=} \mathit{PhoneBooks}$$

- On the RHS of the $\widehat=$ symbol can be any valid *schema calculus expression*.

- Such an expression may be a schema definition (as above); but we can also make new schemas using the propositional connectives $\wedge$, $\vee$, $\neg$, $\Rightarrow$, ... Although these symbols are the same as in propositional logic, they have a different (but related) meaning.
- **Definition:** Two schemas are said to be type compatible if every variable common to both has the same type in both.
- We can use the connectives to make new schemas out of old ones only if they are type compatible. Let α be an arbitrary unary connective, β be an arbitrary binary connective, and S and T be the two schemas

  $$S \mathrel{\widehat{=}} [D_1; \ \cdots; \ D_m \mid P_1; \ \cdots; \ P_n]$$
  $$T \mathrel{\widehat{=}} [D_{m+1}; \ \cdots; \ D_{m+p} \mid P_{n+1}; \ \cdots; \ P_{n+q}]$$

  α S is the following schema

  $$[D_1; \ \cdots; \ D_m \mid \alpha(P_1 \wedge \cdots \wedge P_n)]$$

  If S and T are type compatible, then S β T is the following schema

  $$[D_1; \ \cdots; \ D_{m+p} \mid (P_1 \wedge \cdots \wedge P_n)\beta(P_{n+1} \wedge \cdots \wedge P_{n+q})]$$

- EXAMPLE: Specification of a robust 'Find' operation (i.e. one whose behaviour is defined even when the input name is not known).
- First define a schema which assigns the string 'okay' to a variable. This schema will be used to signify that an operation has been successful.

Success

$$rep! : REPORT$$

$$rep! = \text{'}okay\text{'}$$

- Then define a schema to capture the situation where a phone number is not in the database. Note that the schema causes an error message to be assigned to the report variable *rep*!.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{NotKnown}} \\
\Xi \mathit{PhoneBook} \\
\mathit{name}? : \mathit{NAME} \\
\mathit{rep}! : \mathit{REPORT} \\
\hline
\mathit{name}? \notin \mathit{known} \\
\mathit{rep}! = \text{`}\mathit{name\ not\ known}' \\
\hline
\end{array}
$$

- The robust 'Find' operation is

$$DoFindOp \mathrel{\widehat=} (Find \land Success) \lor NotKnown$$

the full expansion of which is:

$$
\begin{array}{l}
\textit{DoFindOp} \\
\hline
known : \mathbb{P}\,NAME \\
known' : \mathbb{P}\,NAME \\
tel : NAME \pfun PHONE \\
tel' : NAME \pfun PHONE \\
name? : PHONE \\
phone! : PHONE \\
rep! : REPORT \\
\hline
((\mathrm{dom}\ tel = known \land \mathrm{dom}\ tel' = known \\
\land known' = known \land tel' = tel \\
\land name? \in known \\
\land phone! = tel(name?)) \\
\land rep! = \text{`}okay\text{'}) \\
\lor \\
(\ \mathrm{dom}\ tel = known \land \mathrm{dom}\ tel' = known \\
\land known' = known \land tel' = tel \\
\land name? \notin known \\
\land rep! = \text{`}name\ not\ known\text{'}) \\
\hline
\end{array}
$$

- After logical simplification, the expanded schema becomes:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{DoFindOp}} \\
\hline
known : \mathbb{P}\, NAME \\
known' : \mathbb{P}\, NAME \\
tel : NAME \pfun PHONE \\
tel' : NAME \pfun PHONE \\
name? : PHONE \\
phone! : PHONE \\
rep! : REPORT \\
\hline
\mathrm{dom}\ tel = known \\
\land\ known' = known \land tel' = tel \\
\land\ ((name? \in known \\
\quad \land\ phone! = tel(name?) \\
\quad \land\ rep! = \text{`}okay\text{'}) \\
\quad \lor \\
\quad (\ name? \notin known \\
\quad \land\ rep! = \text{`}name\ not\ known\text{'})) \\
\hline
\end{array}
$$

Things to Note

- The use of abstraction: The derived version of *DoFindOp* is easier to read and understand than the expanded version!
- The behaviour of the system is now rigorously specified. For instance, we could prove that, when the precondition of the find operation is satisfied, then a phone number is found.
- Notice that the value of the variable *phone*! is undefined when the operation fails.