Software Verification and Testing

Lecture Notes: Z III

Schemas

remark: we have used Z-definitions for defining relations and functions

question: how can one structure and combine system descriptions
in a specification?

approach: schemas are used for this purpose in Z

this lecture: schemas explained via simple specification examples

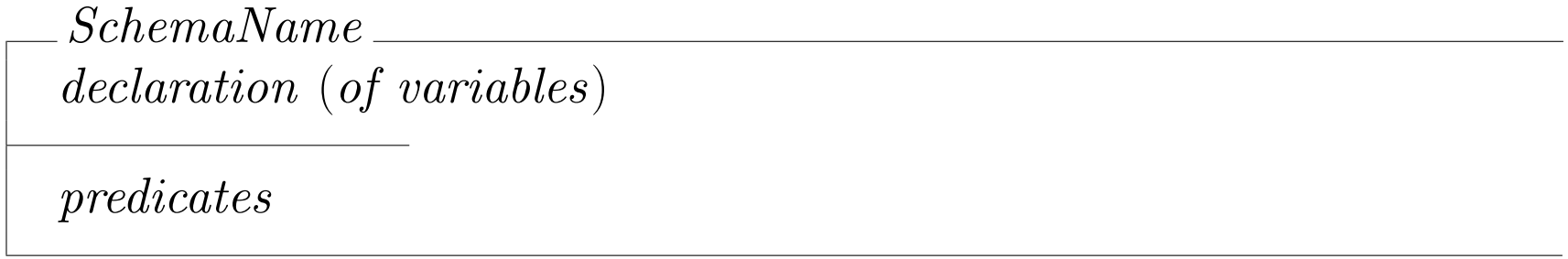
common distinction:

- state schemas define variables of a system and their relations
- operation schemas define the functions that change the state of a system

remark: useful for defining abstract data types

Schemas

general syntax:



remarks:

- variables can be listed or separated by semicolons
- predicates in different lines are implicitly conjoined

A Book Database

example: Wolfgang is a dedicated collector of books

- he has so many books that he tends to buy duplicates
- he estimates that his flat is big enough for 2000 books only
- he needs a database to manage his collection

basic type: $[BOOK]$

axiomatic definition:

$$\begin{array}{|l} max\_size : \mathbb{N} \\ \hline max\_size = 2000 \end{array}$$

A Book Database

state schema for collection

$BookCollection$

$collection : \mathbb{P}\, BOOK$

$\#\, collection \leq max\_size$

explanation:

- the schema name is $BookCollection$
- the variable $collection$ is declared as a set of elements of type $[Book]$
- the system invariant expresses that the size of the collection must not exceed max_size

A Book Database

schemas and types: $BookCollection$ (with its operations) is an abstract datatype

- $b : BookCollection$ says that b is a book-collection
- $(b.\, collection \rightsquigarrow \{PaleFire, Crash, Waverley\})$ denotes that the current collection b is bound to the values $PaleFire$, $Crash$ and $Waverley$
- we write the binding associated with schema S as θS
- we can identify a state schema with its set of bindings

schemas and declarations:

- schemas can be used in declarations, e.g.,
  $\{b : BookCollection \mid \#b.\, collection = 41\}$ or
  $\{BookCollection \mid \#\, collection = 41 \bullet collection\}$
- the possible bindings of a schema are constrained by the declaration

A Book Database

schemas and predicates: schemas can be used as predicates, .i.e. $BookCollection$ defines the subset of all possible book collections that contain no more than 2000 books

A Book Database

an operation schema:

BuyBook

$\Delta\, BookCollection;\ newbook? : BOOK$

$newbook? \notin collection$

$collection' = collection \cup \{newbook?\}$

explanation:

- inputs to operations are written $in?$; outputs to operations are written $out!$
- ΔS denotes that the state variables of schema S are imported; their values may change
- ΞS denotes that imported variables may not change their values
- unprimed and primed variables denote the state before and after the operation

A Book Database

preconditions and postconditions:

- $newbook? \notin collection$ is a precondition of the operation $BuyBook$;
  it must hold before the operation can be executed
- $collection' = collection \cup \{newbook?\}$ is a postcondition of the operation;
  it must hold after the operation has been executed
- a further derived precondition comes from the schema $BookCollection$:
  $\# collection < max\_size$ (otherwise the invariant is violated)

discussion: the analysis of pre- and postconditions is essential for verification

- initialisation analysis: system constraints are satisfiable
- safety analysis: operations are never applied outside their domains

A Book Database

question: what if the preconditions are violated?

answer: exception handling

- three possible situations

$$\#\, collection < max\_size \wedge newbook? \in collection$$
$$\#\, collection = max\_size \wedge newbook? \notin collection$$
$$\#\, collection = max\_size \wedge newbook? \in collection$$

- exception type ("disjoint union")

$$RESPONSE ::= bookowned \mid nospace \mid bookowned\_and\_nospace \mid success$$

A Book Database

exception schemas: book is already owned

$$
\begin{array}{|l}
\hline
AlreadyOwned \\
\hline
\Xi BookCollection \\
newbook? : BOOK \\
reply! : RESPONSE \\
\hline
\# collection < max\_size \\
newbook? \in collection \\
reply! = bookowned \\
\hline
\end{array}
$$

A Book Database

exception schemas: no more space

$NoSpace$

$$\Xi BookCollection$$
$$newbook? : BOOK$$
$$reply! : RESPONSE$$

$$\#collection = max\_size$$
$$newbook? \notin collection$$
$$reply! = nospace$$

A Book Database

exception schemas: no more space and already owned

$$
\begin{array}{|l}
\hline
OwnedNoSpace \\
\Xi BookCollection \\
newbook? : BOOK \\
reply! : RESPONSE \\
\hline
\# collection = max\_size \\
newbook? \in collection \\
reply! = bookowned\_and\_nospace \\
\hline
\end{array}
$$

A Book Database

buying schema: (incl. success message and explicit preconditions)

BuyBook

$$\Delta BookCollection$$
$$newbook? : BOOK$$
$$reply! : RESPONSE$$

$$\#collection < max\_size$$
$$newbook? \notin collection$$
$$collection' = collection \cup \{newbook?\}$$
$$reply! = success$$

A Book Database

full buying schema:

$$BuyingBook \mathrel{\widehat=} BuyBook \lor AlreadyOwned \lor NoSpace \lor OwnedNoSpace$$

- more on schema combination later. . .

initialisation: (after initialisation the collection is empty)

$BookCollectionInit$

$BookCollection'$

$collection' = \varnothing$

Hotel Booking

basic types:

$$[GUEST]$$

$$HOTEL ::= Room1 \mid Room2 \mid \cdots \mid Room15$$

$$RESPONSE ::= success \mid fullybooked \mid notaguest$$

Hotel Booking

state schema:

$$
\begin{array}{|l}
\hline
\textit{Hotel} \\
\textit{guests} : \mathbb{P}\, \textit{GUEST} \\
\textit{freerooms} : \mathbb{P}\, \textit{HOTEL} \\
\textit{bookedrooms} : \mathbb{P}\, \textit{HOTEL} \\
\_\ \textit{occupies}\ \_ : \textit{GUEST} \leftrightarrow \textit{HOTEL} \\
\hline
\textit{guests} = \operatorname{dom} \textit{occupies} \\
\textit{bookedrooms} = \operatorname{ran} \textit{occupies} \\
\textit{freerooms} = \textit{HOTEL} \setminus \textit{bookedrooms} \\
\hline
\end{array}
$$

Hotel Booking

initialisation: (after initialisation the hotel is empty)

$InitHotel$

$$Hotel'$$

$$occupies' = \varnothing$$
$$bookedrooms' = \varnothing$$
$$freerooms' = HOTEL$$

- postconditions are not irredundant. . .

Hotel Booking

operation schemas: booking a room

Book

$\Delta\, Hotel$

$guest? : GUEST$

$room! : HOTEL$

$reply! : RESPONSE$

$room! \in freerooms$

$occupies' = occupies \cup \{guest? \mapsto room!\}$

$reply! = success$

Hotel Booking

operation schemas: booking error

$BookError$

$\Xi Hotel$

$reply! : RESPONSE$

$freerooms = \varnothing$

$reply! = fullybooked$

Hotel Booking

operation schemas: checking out the hotel

CheckOut

$$\Delta Hotel$$
$$guest? : GUEST$$
$$reply! : RESPONSE$$

$$guest? \in guests$$
$$occupies' = \{guest?\} \lhd occupies$$
$$reply! = success$$

Hotel Booking

operation schemas: check out error

CheckOutError

$$\Xi Hotel$$
$$guest? : GUEST$$
$$reply! : RESPONSE$$

$$guest? \notin guests$$
$$reply! = notaguest$$

Hotel Booking with Tab

extension: a tab collects room service, drinks, etc for a room

basic types:

$$[GUEST]$$

$$HOTEL ::= Room1 \mid Room2 \mid \cdots \mid Room15$$

$$RESPONSE ::= success \mid fullybooked \mid notaguest \mid wrongnumber \mid isaddedtotab$$

Hotel Booking with Tab

state schema:

$$
\begin{array}{|l}
\hline
\textit{Hotel} \\
\hline
guests : \mathbb{P}\, GUEST \\
freerooms : \mathbb{P}\, HOTEL \\
bookedrooms : \mathbb{P}\, HOTEL \\
\_\ occupies\ \_ : GUEST \leftrightarrow HOTEL \\
\_\ tab\ \_ : HOTEL \rightarrow \mathbb{N} \\
\hline
guests = \mathrm{dom}\ occupies \\
bookedrooms = \mathrm{ran}\ occupies \\
freerooms = HOTEL \setminus bookedrooms \\
\hline
\end{array}
$$

Hotel Booking with Tab

initialisation:

$$
\begin{array}{|l}
\hline
\textit{InitHotel} \\
Hotel' \\
\hline
occupies' = \varnothing \\
bookedrooms' = \varnothing \\
freerooms' = HOTEL \\
tab' = \varnothing \\
\hline
\end{array}
$$

Hotel Booking with Tab

operation schemas: booking a room

$$
\begin{array}{|l}
\hline
Book \\
\Delta\, Hotel \\
guest? : GUEST \\
room! : HOTEL \\
reply! : RESPONSE \\
\hline
room! \in freerooms \\
occupies' = occupies \cup \{guest? \mapsto room!\} \\
tab' = tab \oplus \{room! \mapsto 0\} \\
reply! = success \\
\hline
\end{array}
$$

Hotel Booking with Tab

operation schemas: booking error

BookError

$$\Xi Hotel$$
$$reply! : RESPONSE$$

$$freerooms = \varnothing$$
$$reply! = fullybooked$$

Hotel Booking with Tab

operation schemas: checking out

CheckOut

$$\Delta\, Hotel$$
$$guest? : GUEST$$
$$bill! : \mathbb{N}$$
$$reply! : RESPONSE$$

$$\exists\, b : \mathbb{N} \bullet$$
$$guest? \in guests$$
$$guest?\,(occupies \fatsemi tab)\, b$$
$$bill! = b$$
$$occupies' = \{guest?\} \lhd occupies$$
$$tab' = tab$$
$$reply! = success$$

Hotel Booking with Tab

operation schemas: check out error

$CheckOutError$

$\Xi Hotel$

$guest? : GUEST$

$reply! : RESPONSE$

$guest? \notin guests$

$reply! = notaguest$

Hotel Booking with Tab

operation schemas: adding to tab

$AddToTab$

$\Delta Hotel$

$room? : HOTEL$

$charge? : \mathbb{N}$

$reply! : RESPONSE$

$room? \in bookedrooms$

$tab' = tab \oplus \{room? \mapsto (tab(room?) + charge?)\}$

$occupies' = occupies$

$reply! = isaddedtotab$

Hotel Booking with Tab

operation schemas: error for adding to tab

$$
\begin{array}{|l}
\hline
AddToTabError \\
\Xi Hotel \\
room? : HOTEL \\
reply! : RESPONSE \\
\hline
room? \notin bookedrooms \\
reply! = notaguest \\
\hline
\end{array}
$$

Hotel Booking with Tab

remarks:

- various other design possibilities
- narrow types can lead to simpler specifications
- should pre- postconditions be explicit or implicit?

question: what about a hotel with n rooms?

answer: schemas must be parametrised. . .

Generic Schemas

example: stack of arbitrary values implemented on sequence

$$
\begin{array}{|l}
\hline
Buffer\,[X] \\
buffer : \operatorname{seq} X \\
size : \mathbb{N} \\
max\_size : \mathbb{N} \\
\hline
size = \#buffer \\
size \leq max\_size \\
\hline
\end{array}
$$

Generic Schemas

initialisation:

$$
\begin{array}{|l}
\hline
BufferInit\ [X] \\
Buffer'[X] \\
\hline
buffer' = \langle\,\rangle \\
\hline
\end{array}
$$

Generic Schemas

invariant: capacity remains unchanged

$$
\begin{array}{|l}
\hline
UpdateBuffer\ [X] \\
Buffer[X] \\
Buffer'[X] \\
\hline
max\_size' = max\_size \\
\hline
\end{array}
$$

Generic Schemas

insertion: add new value at top of stack

$$
\begin{array}{|l}
\hline
BufferIn\,[X] \\
\hline
UpdateBuffer[X] \\
x? : X \\
\hline
size < max\_size \\
buffer' = x? : buffer \\
\hline
\end{array}
$$

Generic Schemas

deletion: take value at top of stack

$$
\begin{array}{|l}
\hline
BufferOut\ [X] \\
\hline
UpdateBuffer[X] \\
x! : X \\
\hline
buffer \neq \langle\,\rangle \\
buffer' = tail\ buffer \\
x! = head\ buffer \\
\hline
\end{array}
$$

Schema Operators

idea: define operations on schemas for further structuring

- inclusion
- disjunction
- conjunction
- negation
- decoration
- change of state
- renaming
- hiding

Schema Operators

two schemas:

S

$$a : \mathbb{N}$$

$$a \leq 4$$

T

$$b : \mathbb{N}$$

$$b = 5$$

Schema Operators

inclusion: (types must match!)

$SinclT$

S

$b : \mathbb{N}$

$b = 5$

$SinclT$

$a, b : \mathbb{N}$

$(a \leq 4) \land (b = 5)$

Schema Operators

disjunction: (types must match!)

$S \lor T$

$a, b : \mathbb{N}$

$(a \leq 4) \lor (b = 5)$

Schema Operators

conjunction: (same effect as inclusion, types must match!)

$S \wedge T$

$a, b : \mathbb{N}$

$(a \leq 4) \wedge (b = 5)$

Schema Operators

negation:

$\neg S$

$a : \mathbb{N}$

$\neg(a \leq 4)$

Schema Operators

decoration: (introduction of "after" states)

S'

$a' : \mathbb{N}$

$a' \leq 4$

Schema Operators

delta and xi:

- $\Delta S \mathrel{\widehat{=}} S \land S'$
- ΞS requires that the state of S remains unchanged

$$\frac{\Xi S \qquad \Delta S}{\theta S = \theta S'}$$

Schema Operators

renaming:

$$S[x/a]$$
$$x : \mathbb{N}$$
$$\hline$$
$$x \leq 4$$

Schema Operators

hiding:

$S \setminus (a)$

$$\exists\, a : \mathbb{N} \bullet a \leq 4$$

Conclusion

what we have learned:

- Z is a powerful and versatile specification language
- Z specifications can be seen as "non-executable programs"
- properties of specifications can be analysed within first-order logic, set theory and relational calculus
- induction is a powerful tool for reasoning about abstract data types
- working with Z requires some mathematical skills

Conclusion

what we have not learned:

- formal refinement from Z specifications to executable code
- formal pre- and postcondition analysis in Z
- extension of Z to concurrent and reactive systems
- extension of Z to OO-analysis

UML vs Z: what would you use for developing

- a small, but complex safety-critical system?
- a huge data-management system?