Development of Sequential Programs

Event-B can be used to develop concurrent and distributed programs. As a particular case, it can be used to develop sequential programs. This is what we will do in this set of slides.

At the moment we will not focus on the actual proofs; we will rather have a look at how the general process goes by using a simple example: searching in an array.

Previous session:
• Guarded events
• Decompose sequential program into guarded events
• Phone agenda example

Today:
• Synthesize sequential program to perform array searches
• On the way: more Event-B builtins, proof obligations

Not without difficulties:
– Deciding which new events / variables to generate
– Prove that desirable properties are still retained

General Ideas
• Event model decomposition by refinement:
  – Start with very few events (sometimes only one), very few state variables
  – (Sometimes) End with many events and variables
    * New events / variables introduced during refinement
    * Each piece should be refinable independently from the others
    * Every event takes care of some aspect of the system architecture
  – At the end: assemble the events to construct a single program (or set of thereof)
Sequential Programs are usually specified by means of:
- A pre-condition
- and a post-condition

It is represented with a Hoare-triple

\[
\{ \text{Pre} \} \ P \ \{ \text{Post} \}
\]

Example 1: The search Program

- We are given (Pre-condition)
  - a natural number \( n: n \in \mathbb{N} \)
- We are given (Pre-condition)
  - a natural number $n$: $n \in \mathbb{N}$
  - $n$ is positive: $0 < n$
  - an array $f$ of $n$ elements built on a set $S$: $f \in 1..n \rightarrow S$
  - a value $v$ known to be in the array: $v \in \text{ran}(f)$

- We are looking for (Post-condition)
  - a value $v$ known to be in the array: $v \in \text{ran}(f)$

- We are given (Pre-condition)
  - a natural number $n$: $n \in \mathbb{N}$
  - $n$ is positive: $0 < n$
  - an array $f$ of $n$ elements built on a set $S$: $f \in 1..n \rightarrow S$
  - a value $v$ known to be in the array: $v \in \text{ran}(f)$

- We are looking for (Post-condition)
Example 1: The search Program

- We are given (Pre-condition)
  - a natural number \( n \): \( n \in \mathbb{N} \)
  - \( n \) is positive: \( 0 < n \)
  - an array \( f \) of \( n \) elements built on a set \( S \): \( f \in 1 \ldots n \rightarrow S \)
  - a value \( v \) known to be in the array: \( v \in \text{ran}(f) \)

- We are looking for (Post-condition)
  - an index \( r \) in the domain of the array: \( r \in \text{dom}(f) \)

- We are given (Pre-condition)
  - a natural number \( n \): \( n \in \mathbb{N} \)
  - \( n \) is positive: \( 0 < n \)
  - an array \( f \) of \( n \) elements built on a set \( S \): \( f \in 1 \ldots n \rightarrow S \)
  - a value \( v \) known to be in the array: \( v \in \text{ran}(f) \)

- We are looking for (Post-condition)
  - an index \( r \) in the domain of the array: \( r \in \text{dom}(f) \)
  - such that \( f(r) = v \)

- Input parameters are constants
- The pre-condition corresponds to axioms of these constants
- Output parameters are variables
- The post-condition is in the guard of a unique event
- [When developing several programs in the same module,
  - input parameters can also be variables of a special "init" event]
Encoding a Hoare-triple in an Event System

\[
\begin{align*}
&\{ n \in \mathbb{N} \\
&0 < n \\
&f \in 1..n \rightarrow S \\
&v \in \text{ran}(f) \} \\
&\text{search} \\
&\{ r \in \text{dom}(f) \\
&f(r) = v \}
\end{align*}
\]

Event status

Note: **status anticipated**

- **convergent**: new event in refined machine
  - Concerned with *variants* — will see in short
- **anticipated**: new event which is not convergent but which should become convergent.
- Most times, status is **normal**
  - Not related to the *variant* section
  - Not explicitly stated
Assignments

$x := E(v)$ deterministic assignment

$x :\in E(v)$ nondeterministic assignment
- $x$ is assigned an element from the set described by $E(v)$

$x :| Q(v, x')$ nondeterministic assignment
- Nondeterministically select some $x'$ such that $Q(v, x')$ holds and assign it to $x$.
- Note that $x$ may occur inside $v$

Every assignment is modeled as a before-after predicate $S(v, v')$

Development of the search Program: Refinement

\begin{align*}
\textbf{inv1}_1: & \quad r \in 1 .. n \\
\textbf{inv1}_2: & \quad v \notin f[1..r-1] \\
\textbf{variant1}: & \quad n - r
\end{align*}

\begin{align*}
\text{init} \quad & \quad r := 1 \\
\text{progress} \quad & \quad \text{convergent} \\
& \quad \text{when} \quad f(r) \neq v \\
& \quad \text{then} \quad r := r + 1 \\
& \quad \text{end}
\end{align*}

\begin{align*}
\text{final} \quad & \quad \text{when} \quad f(r) = v \\
& \quad \text{then} \quad \text{skip} \\
& \quad \text{end}
\end{align*}

Ideas for a Refinement

Result variable $r$ is set to 1 initially

\begin{align*}
\text{Initially} \\
\text{Current situation}
\end{align*}

\begin{align*}
\text{variant}: & \quad \text{either} \\
& \quad \text{A natural number expression which must be decreased by each convergent event and not be increased by each anticipated event, or} \\
& \quad \text{a finite set expression which must be made strictly smaller by each convergent event or not made greater by each anticipated events.} \\
& \quad \text{Necessary to prove that no event diverges, i.e., moves the machine arbitrarily away from a final state.}
\end{align*}
To be Proved (as usual)

- Events refine their abstractions
- Events maintain invariants
- The exhibited variant is a natural number
- Event progress decreases the variant
- The system is deadlock free

Proof obligation: invariant preservation

- Machine with invariant $I$
- For every event with guard $G$ and action $S$ we must prove:

  $$I(v), G(v), S(v, v') \vdash I(v')$$

- Similar for initialization, but no assumption that invariant / guard holds before
- Necessary in every step (also in initial, non-refined machine)
- Refinement proofs: we will see them later
- Note: for simplicity we are not distinguishing at the moment machine state and event parameters.

Initialisation
begin
  $r := 1$
end

Event progress $\triangleq$
when
  $f(r) \neq v$
then
  $r := r + 1$
end

inv1_1: $r \in 1..n$
inv1_2: $v \notin f[1..r-1]$

axiom0_4: “a value $v$ known to be in the array: $v \in dom(f)$”
(or, $\exists i \in 1..n \cdot f(i) = v$)

• Tell me how to prove inv1_1
• Tell me how to prove inv1_2
Proof obligation: deadlockfreeness

- Event execution does not stall
- For events with guards $G_1, \ldots, G_n$:

$$G_1 \lor G_2 \lor \ldots \lor G_{n-1} \lor G_n$$

always holds
- Again: to be proven initially and after refinements
  - Need **gluing invariants** $J(v, w)$ which relate the states of the abstract machine ($v$) and that of the concrete machine ($w$)

Constructing the Final Program

We are using some **Merging Rules** to build the final program

```
init
  r := 1
progress
  when
    f(r) \ne v
  then
    r := r + 1
  end
final
  when
    f(r) = v
  then
    skip
  end
```

Merging Rule (1)

```
when
  P
  Q
then
  S
end

when
  P
  \neg Q
then
  T
end
```

- Side Conditions:
  - $P$ must be invariant under $S$
  - The first event must have been introduced at one refinement step below the second one.

- Special Case: If $P$ is missing the resulting "event" has no guard

Merging Rule (2)

```
when
  P
  Q
then
  S
end

when
  P
  \neg Q
then
  T
end

when
  P
then
  if
  Q
  then
    S
  else
    T
  end
```

- Side Conditions:
  - The disjunctive negation of the previous side conditions

- Special Case: If $P$ is missing the resulting "event" has no guard
Applying Rule M\_WHILE (special case)

\[
\begin{align*}
\text{progress} & \quad \text{when } f(r) \neq v \\
& \quad \text{then } r := r + 1 \\
& \quad \text{end}
\end{align*}
\]

final

\[
\begin{align*}
\text{final} & \quad \text{when } f(r) = v \\
& \quad \text{then } \text{skip} \\
& \quad \text{end}
\end{align*}
\]

progress\_final

\[
\begin{align*}
\text{progress\_final} & \quad \text{while } f(r) \neq v \text{ do} \\
& \quad r := r + 1 \\
& \quad \text{end}
\end{align*}
\]

Final Rule M\_INIT

- Once we have obtained an “event” without guard
- We add to it the event \textit{init} by sequential composition
- We then obtain the final “program”

Example 2: The Very Classical Binary Search

- Almost the same specification as in Example 1
- It will show the usage of more merging rules

\[
\begin{align*}
\text{search\_program} & \quad r := 1; \\
& \quad \text{while } f(r) \neq v \text{ do} \\
& \quad r := r + 1 \\
& \quad \text{end}
\end{align*}
\]

\[
\begin{align*}
\{ r \in \text{dom}(f) \} & \quad \{ f(r) = v \}
\end{align*}
\]
- We are given (Pre-condition)
  - a natural number $n$: $n \in \mathbb{N}$
  - $n$ is positive: $0 < n$

- We are given (Pre-condition)
  - a natural number $n$: $n \in \mathbb{N}$
  - $n$ is positive: $0 < n$
  - a sorted array $f$ of $n$ elements built on a set $\mathbb{N}$: $f \in 1..n \rightarrow \mathbb{N}$
- We are given (Pre-condition)
  - a natural number \( n \): \( n \in \mathbb{N} \)
  - \( n \) is positive: \( 0 < n \)
  - a sorted array \( f \) of \( n \) elements built on a set \( \mathbb{N} \): \( f \in 1..n \rightarrow \mathbb{N} \)
  - a value \( v \) known to be in the array: \( v \in \text{ran}(f) \)

- We are looking for (Post-condition)
  - an index \( r \) in the domain of the array: \( r \in \text{dom}(f) \)
  - such that \( f(r) = v \)
Specify when an array is sorted!

First Refinement: the State

constants: \(n, f, v\)
variables: \(r\)

inv0.1: \(r \in \mathbb{N}\)

- Current situation

\[
\begin{array}{ccccccc}
1 & p-1 & r & q+1 & n \\
\hline
& & & & v : f[p..q] & \\
& & & & p & q \\
\end{array}
\]

Note on invariants:

- We postulate that \(v \in f[p..q]\)
  - We had before \(v \notin f[1..r-1]\)
- Assuming this, we prove that:
  - Refined events keep this property
  - Refined events progress towards termination

Note on refinements:

- In the previous example we refined one event into another event
- Now, we will refined one event into two events: inc and dec

Binary Search: the State

constants: \(n, f, v\)
variables: \(r\)

axm0.1: \(n \in \mathbb{N}\)
axm0.2: \(0 < n\)
axm0.3: \(f \in 1..n \to \mathbb{N}\)
axm0.4: \(\forall i, j \cdot \left( \begin{array}{c} i \in 1..n \\
\quad j \in 1..n \\
\quad i \leq j \\
\Rightarrow f(i) \leq f(j) \end{array} \right)\)
axm0.5: \(v \in \text{ran}(f)\)
Why are we sure we are not discarding potential solutions?

In other words, how do we know we are respecting the invariant

\[ v \in f[p..q] \]

- From axioms: \( \forall i, j \in 1..n \cdot i \leq j \Rightarrow f(i) \leq f(j) \)
- We had: \( v \in f[p..q] \land r \in p..q \)
- We know: \( f(r) < v \)
- We postulate: \( v \in f[r + 1..q] \)
- At the previous stage, inc and dec were non-deterministic
- r was chosen arbitrarily within the interval p .. q
- We now remove the non-determinacy in inc and dec
- r is chosen to be the middle of the interval p .. q

Reducing Non-determinacy

(abstract) inc
\[ \text{when } f(r) < v \text{ then } \]
\[ p := r + 1 \]
\[ r := (r + 1 .. q) \]
end

(concrete) inc
\[ \text{when } f(r) < v \text{ then } \]
\[ p := r + 1 \]
\[ r := (r + 1 + q)/2 \]
end

(abstract) dec
\[ \text{when } f(r) < v \text{ then } \]
\[ q := r - 1 \]
\[ r := p .. r - 1 \]
end

(concrete) dec
\[ \text{when } f(r) < v \text{ then } \]
\[ q := r - 1 \]
\[ r := (p + r - 1)/2 \]
end

Merging Rule M_IF

\[ \text{when } P \text{ then } \]
\[ \text{if } Q \text{ then } S \text{ else } T \]
end
end

\[ \text{when } P \text{ then } \]
\[ \text{if } Q \text{ then } S \text{ else } T \]
end
end
end

M_IF
Merging Events inc and dec by means of Rule M_IF

```plaintext
inc
when
  \( f(r) \neq v \)
  \( f(r) < v \)
then
  \( p := r + 1 \)
  \( r := (r + 1 + q)/2 \)
end

dec
when
  \( f(r) \neq v \)
  \( v \leq f(r) \)
then
  \( q := r - 1 \)
  \( r := (p + r - 1)/2 \)
end
```

Merging Events inc_dec and bin_search with Rule M_WHILE

```plaintext
inc_dec
when
  \( f(r) \neq v \)
then
  if \( f(r) < v \) then
    \( p, r := r + 1, (r + 1 + q)/2 \)
  else
    \( q, r := r - 1, (p + r - 1)/2 \)
end

final
when
  \( f(r) = v \)
then
  skip
end
```

Merging Rule M_WHILE

```plaintext
when
  \( P \)
  \( Q \)
then
  \( S \)
end
when
  \( P \)
  \( \neg Q \)
then
  \( T \)
end
```

- Side Conditions:
  - \( P \) must be invariant under \( S \)
  - The first event must have been introduced at one refinement step below the second one.

- Special Case: If \( P \) is missing the resulting "event" has no guard

Merging Events inc_dec_bin_search and init with Rule M_INIT

```plaintext
inc_dec_final
while
  \( f(r) \neq v \)
do
  if \( f(r) < v \) then
    \( p, r := r + 1, (r + 1 + q)/2 \)
  else
    \( q, r := r - 1, (p + r - 1)/2 \)
end

init
when
  \( f(r) = v \)
then
  skip
end
```

bin_search_program

```plaintext
p, q, r := 1, n, (1 + n)/2;
while
  \( f(r) \neq v \)
do
  if \( f(r) < v \) then
    \( p, r := r + 1, (r + 1 + q)/2 \)
  else
    \( q, r := r - 1, (p + r - 1)/2 \)
end
```

```plaintext
init
p, q := 1, n
r := (1 + n)/2
```
• Had we kept

\[ r \in r + 1..q \]
\[ r \in p..r - 1 \]

would have it worked at all?