Refinement

Imagine we have a temporal specification, \( \text{Spec} \), about which we can establish some properties:

\[
\vdash \text{Spec} \Rightarrow \text{Properties}
\]

Now, maybe the specification we have lacks some details, so we produce a revised specification, say \( \text{Spec}' \).

Clearly we do not want to prove \( \vdash \text{Spec}' \Rightarrow \text{Properties} \) if we can avoid it, and surely we would like all the behaviours of \( \text{Spec}' \) to also be behaviours of \( \text{Spec} \).

Thus, as in many formal specification techniques, we can refine \( \text{Spec} \) to a new specification \( \text{Spec}' \) as long as we can establish that

\[
\vdash \text{Spec}' \Rightarrow \text{Spec}
\]

Example

Let us consider a sample initial temporal specification and its refinement:

\[
\begin{align*}
\text{Spec} &= p \land \diamond q \\
\text{Spec}' &= p \land \circ q
\end{align*}
\]

Since \( \vdash (p \land \circ q) \Rightarrow (p \land \diamond q) \), then \( \text{Spec}' \) is indeed a refinement of \( \text{Spec} \). Consequently, if we know that

\[
\vdash \text{Spec} \Rightarrow (\diamond p \land \diamond q)
\]

(which is indeed true) then it must also be the case that

\[
\vdash \text{Spec}' \Rightarrow (\diamond p \land \diamond q).
\]

Aside: The meaning of ‘\( \Rightarrow \)’

What does it mean when we say \( \varphi \Rightarrow \psi \)?

Recall that, models for our temporal formulae are of the form \( \langle N, \pi \rangle \), representing

So, \( \varphi \Rightarrow \psi \) means: for all \( \pi \), if \( \langle N, \pi \rangle \models \varphi \) then \( \langle N, \pi \rangle \models \psi \).

Alternatively: the set of models that satisfy \( \varphi \) is a subset of the set of models that satisfy \( \psi \).
Refinement (again)

So, when we say $Spec_2 \Rightarrow Spec_1$ this also means that
models satisfying $Spec_2 \subseteq$ models satisfying $Spec_1$.

So, refinements

$$Spec_1$$
$$Spec_2$$
$$\ldots$$
$$\ldots$$
$$Spec_n$$

are successively reducing the number of possible models (and so the set of possible execution sequences).

Example

$$Spec = p \land \Diamond q$$ has models such as

$$p \quad q$$

$$p \quad \quad q$$

$$p \quad \quad \quad q$$

$Spec$ can be refined to $Spec' = p \land \Box q$ which only has models of the form

$$p \quad q$$

So, models of $Spec'$ are a subset of the models of $Spec$.

Linking Specifications

Consider a specification, $Spec_1$, for a program element $E_1$, together with another specification, $Spec_2$, for element $E_2$.

In programming languages, the elements can be executed together in a variety of ways (e.g. interleaved, concurrent).

In addition, the ways in which information can be passed between these elements should be identified (see later).

Note that, if no such communication occurs between elements, then they cannot have any significant effect on each other (though we may still be interested in the evolution of the combined system).

So, we must first decide upon, and describe, the form of concurrency and communication used in our systems.

Concurrency

Throughout the following, we will assume we have several independent elements, $E_1, E_2, \ldots E_n$.

These elements may be processes, objects, threads, agents, etc.

We will primarily tackle the simplest mechanism for combining elements, namely synchronous true concurrency.

Before we do this, we should recap what this means.

In the following, we will look at various alternatives, the choice of which can make a great deal of difference to the ease/difficulty of temporal specification.
Interleaving .vs. True Concurrency

- **True Concurrency** allows the elements to be executing at the same time, but independently:

  ![Diagram: True Concurrency](image1)

- **Interleaving** ensures that only one of the elements can execute at any one time.

  ![Diagram: Interleaving](image2)

Characterising interleaving:

\[ \bigwedge \neg (\text{executing}(E_1) \land \text{executing}(E_2)) \]

Synchrony .vs. Asynchrony

- **Synchronous** execution ensures that all elements work on the same notion of next moment (shared clock).

  ![Diagram: Synchronous](image3)

- **Asynchronous** execution allows the elements to work at different speeds.

  ![Diagram: Asynchronous](image4)

Again, we will mainly use **synchronous, true concurrency**, though we will explore other possibilities later.

Concurrency

We have

- \( \text{Spec}_A \), a specification of element \( A \), and
- \( \text{Spec}_B \), a specification of element \( B \).

Now, we want to specify the system comprising \( A \) and \( B \)

- executing **concurrently**, and
- executing **synchronously**, but without any **communication**.

To do this, we simply conjoin the specifications, i.e.

\[ \text{Spec}_A \land \text{Spec}_B \]

Example: Specification

\[ \begin{align*}
\text{Spec}_A &= a \land b \land (c \land (a \lor b)) \\
\text{Spec}_B &= x \land (y \land (z))
\end{align*} \]

Then the synchronous, concurrent combination of \( A \) and \( B \) is

\[ \text{Spec}_A \land \text{Spec}_B = (a \land x) \land (b \land (c \land y) \land (z \land (a \lor b))) \]

Notice how, in the above example, there is BrickRed interaction between the propositions in \( \text{Spec}_A \), namely \( \{a, b, c\} \), and the propositions in \( \text{Spec}_B \), namely \( \{x, y, z\} \).
Example: Models/Executions (1)

When we consider models/executions over the alphabet \{a, b, c, x, y, z\}, we note that a model for \( \text{Spec}_A \land \text{Spec}_B \) must be a model for both \( \text{Spec}_A \) and \( \text{Spec}_B \) separately.

Sample models of \( p \land \square q \) (over the alphabet \{p, q, r\}) are

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>~r</th>
<th>~r</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sample models of \( r \land \square \neg r \) (over the alphabet \{p, q, r\}) are

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>~r</th>
<th>~r</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The only model for \( (p \land \square q) \land (r \land \square \neg r) \) here is

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>~r</th>
<th>~r</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: Semantics (1)

Consider the possible models (or execution sequences) of each of these elements separately.

- \( E_1 = x:=0; x:=x+1; x:=5; \) end
- \( E_2 = y:=7; y:=y-1; y:=0; \) end

Sample models (over all variables, \( x \) and \( y \)) for \( E_1 \) include:

- \( x=0 \) \( y=7 \)
- \( x=1 \) \( y=7 \)
- \( x=5 \) \( y=6 \)

Notice how the values of \( x \) are constrained by the code of \( E_1 \), while \( y \) is unconstrained.

Example: Models/Executions (2)

Sample models of \( r \land \square \neg r \) (over the alphabet \{p, q, r\}) are

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>~r</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The only model for \( (p \land \square q) \land (r \land \square \neg r) \) here is

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>~r</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If two formulae have models, and if there is no interaction between the alphabets, then the conjunction has a model.

Example: Semantics (2)

Sample models (again over all variables, \( x \) and \( y \)) for \( E_2 \) include:

- \( x=0 \) \( y=7 \)
- \( x=1 \) \( y=6 \)
- \( x=2 \) \( y=0 \)

Now, amongst all these, the only common model/execution for the synchronous, concurrent combination of \( E_1 \) and \( E_2 \) is:

- \( x=0 \) \( y=7 \)
- \( x=1 \) \( y=6 \)
- \( x=5 \) \( y=0 \)