Temporal Logic

[Introducing Formal Methods]

Michael Fisher

Department of Computer Science, University of Liverpool, UK

[MFisher@liverpool.ac.uk]

An Introduction to Practical Formal Methods Using Temporal Logic

What are Formal Methods?

Overall we are concerned with *Formal Methods*.

Definition:

> "Mathematically based techniques for the specification, development and verification of software and hardware systems."

In particular, we will examine methods based upon formal logics. But why?

Formal logics are *mathematical notations* with well understood, and clear, semantics.

In addition, there are a wide range of automated *tools* that can handle formal logics.

Critical Systems

Getting a hardware/software system ‘wrong’ during development can often be expensive

$\rightarrow$ leads to patches, or even recall/replacement.

In some cases this can even be *dangerous*, for example in *critical systems* such as

- aircraft/spacecraft control,
- industrial process control,
- power station control,
- telecommunications, etc.

So, where the software/hardware is ‘important’, increased assurance is often required.

Common Problems

But this definition of *importance* is even wider.

If *business critical systems* fail, the viability of a business may be compromised, e.g. financial, security, privacy.

Even when systems are not critical, companies are increasingly employing formal methods to instill confidence, find bugs early, improve efficiency, etc.

There are many methodologies for developing large software/hardware systems, but often

- their notation is full of ambiguity, and
- it is almost impossible to check that systems actually implement their requirements.

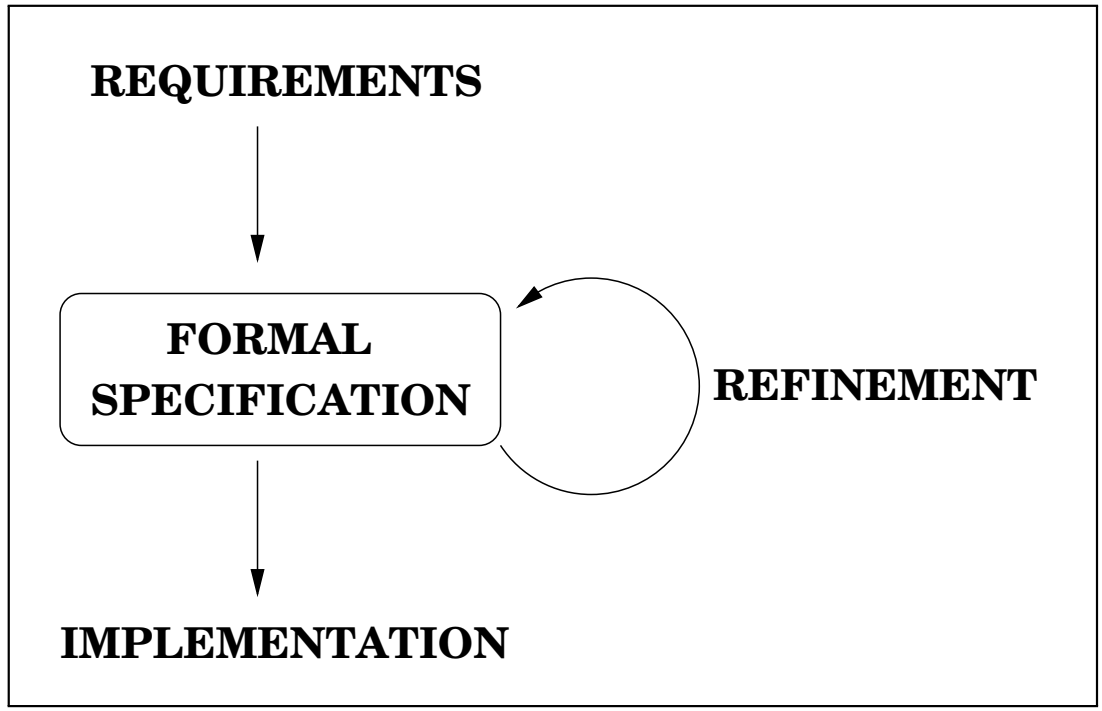
Why use Formal Methods?

Thus, for 'important' (i.e. critical/expensive) systems, Formal Methods may be appropriate, because:

- they provide a formal (exact and unambiguous) notation in which the required properties of the system can be *specified*;
- they provide mechanisms for *developing* specifications toward implementation, whilst still retaining formal properties; and
- they provide mechanisms for *checking* that the code produced really does implement the requirements.

However, there are many Formal Methods, often developed/used for very different classes of system.

Typical Formal Methods Cycle



- Requirements capture: informal $\rightarrow$ formal.
- Refinement: often involves a proof step.
- Implementation: may use formal semantics of the target programming language.

Transformational Systems

It is useful to try to categorize different classes of system.

Transformational systems are essentially those whose behaviour can be described in terms of each component's input/output behaviour.

Each component in a system receives some input, carries out some computations (typically on internal data structures), and terminates producing some output.

Z Specifications

Operations might include: arithmetic, database manipulation, data structure modification, etc.

Specification notations particularly relevant to this type of system (e.g. VDM, Z) were developed in the late 1960's and came to prominence in the 1970's.

Typically, the operations are specified using pre- and post-conditions.

Reactive Systems (1)

Approaches such as VDM and Z have been very effective.

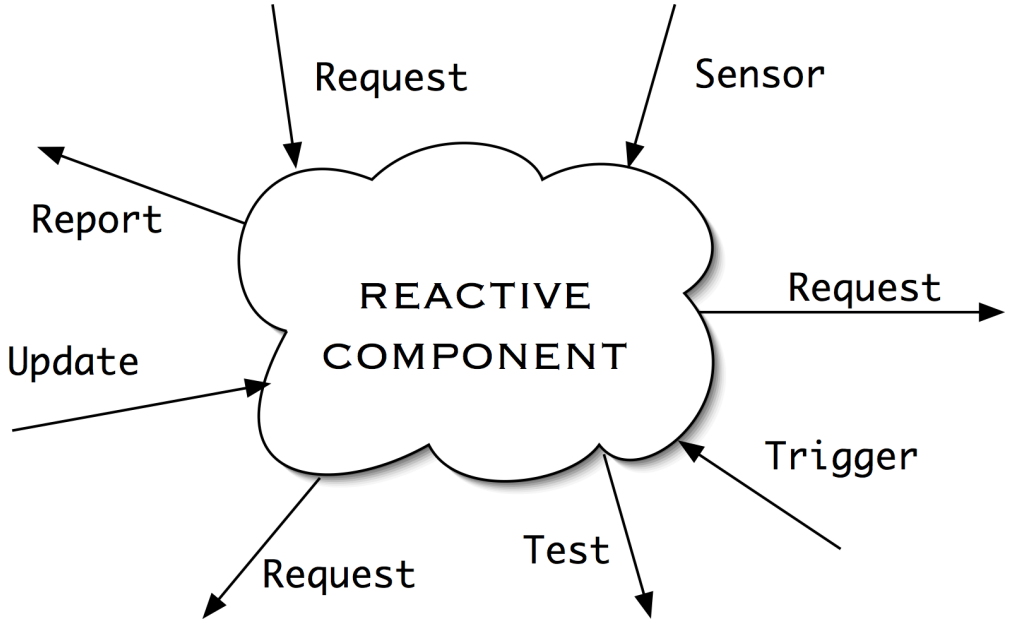
However, it became clear (in the 1970's) that increasingly many systems could not easily be categorised as being 'transformational'.

These systems are typically

- non-terminating,
- continuously reading input
  (not just at the *beginning of computation*),
- continuously producing output
  (not just at the *end*) and
- regularly interacting with other concurrent or distributed components.

Reactive Systems (2)

Such systems are often described as *Reactive Systems*.



Formal Methods for these *reactive* systems indicated that something more than pre- and post-conditions might well be needed.

Why Temporal Logic?

Temporal Logic — an extension of classical logic.

Temporal logics had been used in linguistics (to represent tense) since the 1960s.

Modal logics also provide some of formal foundations.

Temporal logics then began to be applied to formal methods within computer science in late 1970s.

Now temporal techniques are widely used for the specification of more complex *reactive* (i.e. concurrent, dynamic, distributed) systems.

Temporal Logic (1)

An inherent problem with classical logic is its essentially *static* nature.

We can express statements such as

> "*if it is Tuesday and we are in Liverpool, then it is raining*"

but have much more difficulty with dynamic statements such as

> "*if it is Tuesday, then it will continue raining while we remain in Liverpool*"

In classical logic there is no inherent concept of properties changing over time.

Temporal Logic (2)

But there are *many* different temporal logics.

We will concentrate on one very popular variety that is:

- *propositional*, with no explicit first-order quantification;
- *discrete*, with the underlying model of time being isomorphic to the Natural Numbers (i.e. an infinite, discrete sequence with distinguished initial point); and
- *linear*, with each moment in time having at most one successor.

These constraints ensure that each moment in time has *exactly* one successor, hence the use of just one form of the "next step" operator ('$\bigcirc$').

If we allow several immediate successors, then we typically require additional (or at least, modified) operators.

Temporal Logic (3)

Basically, temporal logic allows us to add an implicit *temporal* dimension and provides additional temporal operators such as:

$\Box A$ is true if 'A' is true at *all* times in the future;

$\Diamond A$ is true if 'A' is true at *some* time in the future;

$\bigcirc A$ is true if 'A' is true at the *next* moment in time.

Temporal Logic was developed to be able to represent statements that vary over time and, in the 1970's, such logics were successfully applied to the specification of reactive systems.

Recap

Why are Formal Methods important?

1. As systems become more expensive/critical, increased assurance will be required.
2. Increasingly, we will require ‘proof’ that the systems we buy do actually do what we want.
3. We need to be able to *trust* complex software.

For example, when you buy some software (e.g. a word processor) there is no guarantee that it will actually do *anything*, never mind what you expect it to do!

A systems become more complex (dynamic, distributed, etc), different formal methods have been developed — temporal logic forms the basis for many of these.

Online

Note that, further notes/links related to this material is available online at

`http://www.csc.liv.ac.uk/~michael/TLBook`

Please let me know about any mistakes, lack of clarity, or places where more explanation is required.