Towards Specification, Modelling and Analysis of Fault Tolerance in Self Managed Systems

Tom Maibaum

(joint work with Jeff Magee, ICL)

McMaster University

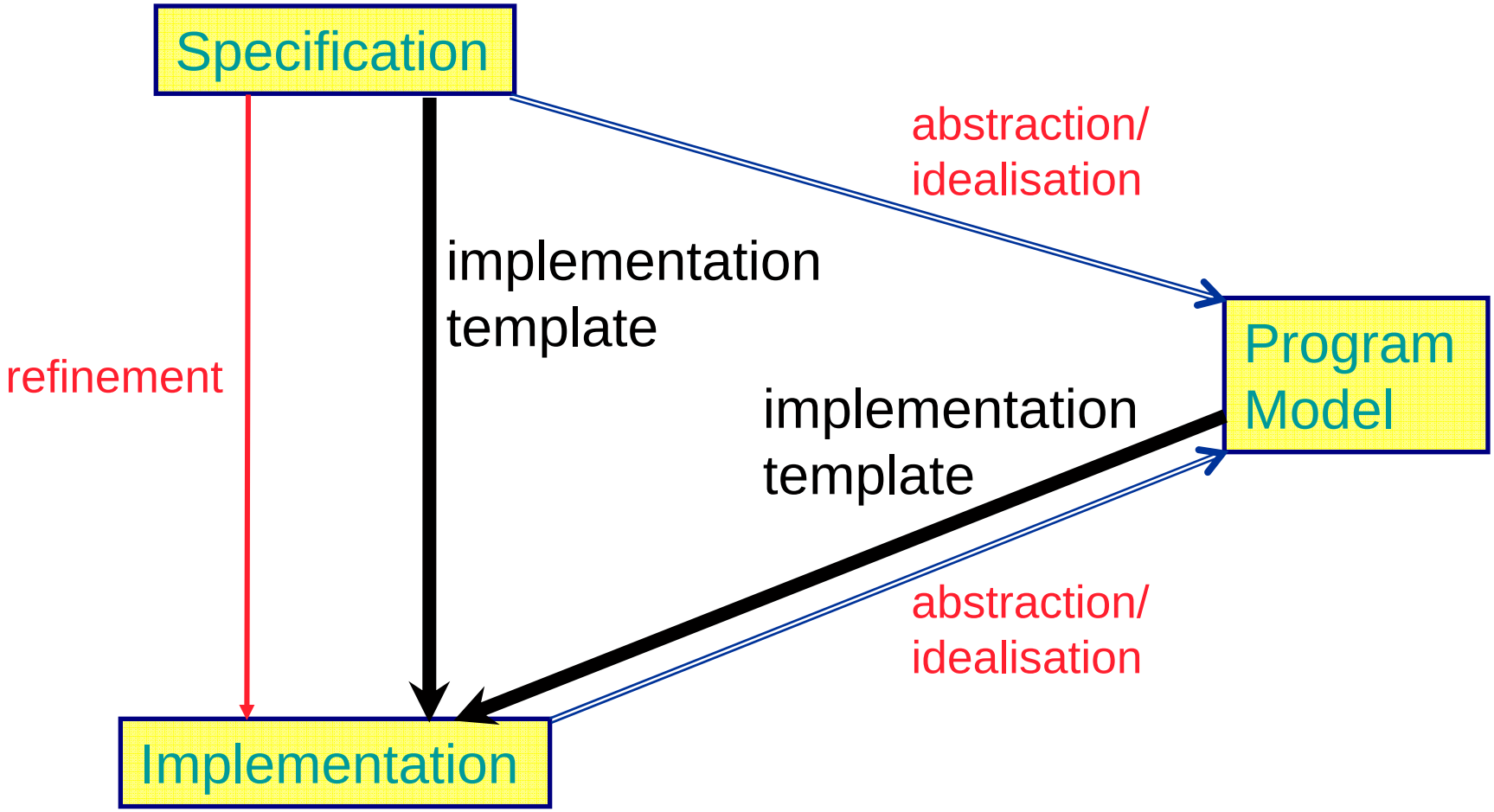
Department of Computing and Software

tom@maibaum.org

Introduction

- We describe initial ideas about an engineering method for modeling and analysing fault tolerance mechanisms in self managed/self healing systems.
- Specifications are component based, with coordination mechanisms for building systems from components.
- A modal action logic is augmented with deontic operators to describe normal vs abnormal behaviours.
- Fault tolerance mechanisms can be specified in terms of the kind of abnormality encountered and the desired recovery route.
- Abstract programming models in LTSA can be systematically constructed from “typical” specifications, a finite state, process algebra based modeling tool.
- LTSA then enables us to check that various properties do or do not hold for the specified fault tolerance mechanisms.
- Templates for translation to (Java) code are used to realise the designs.

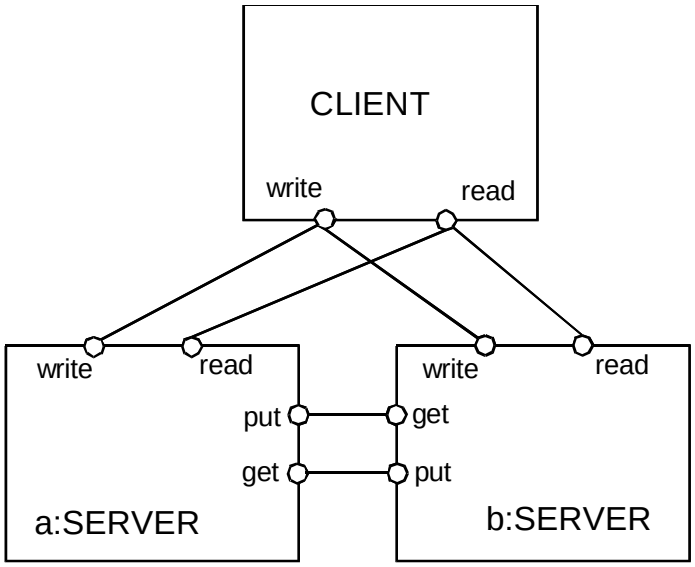
Program models



Normal vs abnormal behaviours

- It is a common assumption in many multi-agent/pervasive systems that *agents/components will behave as they are intended to behave*.
  - Even in systems where the language of ‘obligation’ and ‘permission’ is employed in the specification of agent behaviour, there is an *explicit, built-in assumption that agents always fulfill their obligations and never perform actions that are prohibited*.
- To reason about fault tolerance and self management, we need to *internalise this distinction between normal and abnormal behaviour* to:
  - describe what a fault is
  - to specify recovery mechanisms

Building the specification



component Client

Attributes

val:**int**, master:{a,b},
ready_to_write:**bool**, error:**bool**

Actions

init, write(**int**, master), read(**int**, master), switch, abort

Axioms

1 [init](master=a ∧ val=0 ∧ ready_to_write ∧ ¬error)

2 (ready_to_write ∧ master=m ∧¬error) → [write(val,m)]¬ready_to_write

3(¬ready_to_write ∧ master=m ∧ ¬error ∧ val=x) → [read(y,m)]((x≠y → error) ∧ (x=y → (ready_to_write ∧ val=x+1)))

4 master=a → [switch]master=b

5 c_master=b → [switch]master=a

6 ¬ready_to_write → [switch](¬**normal** ∧ **Obl**(abort))

7 ¬**normal** → [abort](ready_to_write ∧ **normal**)

Reasoning about fault tolerance

- Now, if all goes well, then we should expect that **normal** always holds in the `Client` and we have no error state, i.e.., no fault:

  $(\Box \mathbf{normal}) \rightarrow (\Box \neg \text{"error"})$

- We want to say, to demonsrate that our fault tolerance design works, that if we are *in an abnormal state* (assuming that we have got there by the failover happening in the middle of a transaction by the master server) and *nothing else bad happens*, then eventually ($\Diamond$) we get back to a normal state.

  $\neg \mathbf{normal} \wedge$ "no_further_violation" $\rightarrow \Diamond \mathbf{normal}$

- This is a kind of stability property.

LTSA models

```
const False = 0
const True = 1
range Bool = False..True
range Int = 0..2

SERVER(M=0) = SERVER[M][0][0],
SERVER[master:Bool][val:Int][updating:Bool]
= ( when (master)
    write[v:Int]-> SERVER[master][v][True]
  | when (master && updating)
    put[val]-> SERVER[master][val][False]
  | when (master && !updating)
    read[val]-> SERVER[master][val][updating]
  | when (!master)
    get[u:Int]-> SERVER[master][u][updating]
  | failover -> SERVER[!master][val][False]
  ).
```

LTSA models

- The client offers to read or write to either server “a” or server “b”
  - only the master server will accept these actions
- A CLIENT may be aborted
  - effectively causes it to ignore the effect of the write before abort
- The client contains the simple consistency check that it must read the value it has previously written; if this is not true, then any system in which the CLIENT is included moves irrevocably into an error state. Again, this reflects the behaviour of the client specification above.

```
CLIENT = ({a,b}.write[v:Int] ->({a,b}.read[u:Int]
                -> if (u!=v) then ERROR else CLIENT
          |abort -> CLIENT )).
```

LTSA analysis

- Such a system is described by the following parallel composition:

```
||SYS = (a:Server(True)
        || b:Server(False)
        || CLIENT
        ) /{ a.put/b.get,
             b.put/a.get,
             failover/{a,b.failover}.
```

- Note that the failover action causes an atomic switch from master to slave, as in the spec. But, the client consistency check fails in the following situation:

```
Trace to property violation in CLIENT:
  a.write.1
  failover
  b.read.0
Analysed in: 0ms
```

LTSA analysis

- the client can read the new master state before an update has occurred. We can characterise this situation in FLTL as:

```
fluent UPDATING =
  <{a,b}.write[Int],{{a,b}.put[Int],abort}>

assert BAD = (UPDATING && failover)
```

- The fluent `UPDATING` is true between the point that a write actions occurs changing the master server state and a put action occurs to register that change in the slave. If the action failover occurs while `UPDATING` is true, then the system is in a ¬**normal** state as described in the forgoing**.**

LTSA analysis

- We can simply prohibit the system from entering this state by adding the following constraint:

```
constraint NO_BAD = []! BAD
||CON_SYS = (SYS || NO_BAD).
```

- The constraint is imposed by composing the system with the constraint. The LTSA generates an automaton for the constraint.
- An alternative, and fault tolerant, approach is to let the system get into a bad state and then do some compensating action before the client puts the system directly into the irrecoverable ERROR state. We accomplish this by specifying a constraint that states if we arrived at the BAD or not normal state, then we must immediately (next action) abort.

```
constraint REC_BAD = [](BAD -> X abort)
||REC_SYS = (SYS || REC_BAD).
```

- The use of the next time operator X here is to express the idea that the obliged abort action must be done before anything else.

LTSA analysis

- What this model does not do is reflect the possibility implicit in the spec that other things may then go wrong
  - it would appear that we can model the idea of recovery in the absence of other things going wrong via LTSA, *up to some degree* constrained by both the expressiveness of the temporal logic used and also, of course, by the usual state explosion model checking problem for complex systems
  - the more complex the situation being described, the less is the likelihood that LTSA can cope with it
- So modeling the fault tolerance mechanisms in stages would seem to be an effective process of analysis for complex mechanisms and specifications.

Building the specification

component {a,b}.Server

Attributes

{a,b}.val:**int**, {a,b}.master:**bool**, {a,b}.updating:**bool**

Actions

{a,b}.init, {a,b}.write(**int**), {a,b}.read(**int**),
{a,b}.put(**int**), {a,b}.get(**int**), {a,b}.failover

Axioms

1 [a.init](a.master $\wedge$ $\neg$a.updating)
and for b.Server:
[b.init]($\neg$b.master $\wedge$ $\neg$b.updating)

2 (a.master $\wedge$ $\neg$a.updating)
$\rightarrow$ [a.write(val)](a.val=x $\wedge$ a.updating)

3 (a.master $\wedge$ a.updating) $\rightarrow$ [a.put(val)]$\neg$a.updating

4 a.master $\rightarrow$ [a.failover]$\neg$a.master

5 $\neg$a.master $\rightarrow$ [a.get(x)]a.val=x

6 (For b.Server, we have axioms 2-5 with ‘a’ replaced by
‘b’.)

Building the specification

- Axiom 2 says that if the server is in master mode and it is not in the middle of a ‘write-put’ transaction, then doing a `write(val)` starts a transaction.
- Axiom 3 says that, if a `write` has been done and a `put` immediately follows, then the master is no longer in the middle of the transaction.
- Axiom 4 says that a failover causes a change in the master/slave roles.
  - The action will be coordinated with the `failover` action of the slave, so that the two servers flip roles symmetrically. It will also be coordinated with the `switch` action of the `Client`, so that it ‘knows’ about the changeover.
- Axiom 5 says that if the server is in slave mode and it does a `get`, then the value it reads is put into its local `val`.

LTSA

- The Labelled Transition System Analyzer (LTSA) is a finite state verification tool for modelling and analyzing the behaviour of systems represented by labelled transition systems.
  - a system is modelled as a set of processes described in Finite State Processes (FSP), a process algebra notation
  - permits the analysis of systems with respect to propositional linear temporal logic properties specified in Fluent Linear Temporal Logic (FLTL)
- In the models below:
  - attributes in the specifications above become parameters of the corresponding state machine definition
  - types like **int** have to be made into bounded versions, as LTSA is a finite state analyzer

LTSA models

- When a server is master, it accepts write requests and responds to read requests and, in addition, propagates state changes using `put`.
- When a server is slave, it does not respond to client requests and accepts state changes from the master using `get`.
- The failover action causes a master to become a slave and a slave a master. This reflects the specification of the master server given above.