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 \([\text{init}](\text{master}=a \land \text{val}=0 \land \neg \text{error})\)
2 \((\text{ready_to_write} \land \text{master}=m \land \neg \text{error}) \rightarrow [\text{write}(\text{val},m)]\neg \text{ready_to_write}\)
3 \((\neg \text{ready_to_write} \land \text{master}=m \land \neg \text{error} \land \text{val}=x) \rightarrow [\text{read}(y,m)]((x \neq y \rightarrow \text{error}) \land (x=y \rightarrow (\text{ready_to_write} \land \text{val}=x+1)))\)
4 \(\text{master}=a \rightarrow [\text{switch}]\text{master}=b\)
5 \(\text{c_master}=b \rightarrow [\text{switch}]\text{master}=a\)
6 \(\neg \text{ready_to_write} \rightarrow [\text{switch}]\neg \text{normal} \land \text{Obl}('\text{abort}')\)
7 \(\neg \text{normal} \rightarrow [\text{abort}](\text{ready_to_write} \land \text{normal})\)
Reasoning about fault tolerance

ë 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:

\[(\square \text{normal}) \rightarrow (\square \neg \text{"error"})\]

ë We want to say, to demonstrate 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 (◊) we get back to a normal state.

\[\neg \text{normal} \land \text{"no\_further\_violation"} \rightarrow \diamond \text{normal}\]

ë This is a kind of stability property.
**LTSA models**

```plaintext
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.

\[
\text{CLIENT} = (\{a,b\}.\text{write}[v:\text{Int}] \rightarrow (\{a,b\}.\text{read}[u:\text{Int}] \\
\rightarrow \text{if } (u=v) \text{ then } \text{ERROR } \text{ else } \text{CLIENT} \\
\mid \text{abort } \rightarrow \text{CLIENT }))
\]
LTSA analysis

Such a system is described by the following parallel composition:

$$\langle a: \text{Server(True)} || b: \text{Server(False)} || \text{CLIENT} \rangle /\{ a.\text{put}/b.\text{get}, b.\text{put}/a.\text{get}, \text{failover}/\{a,b.\text{failover}\} \}.$$ 

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

\[
\text{fluent UPDATING } = \\
\langle \{a,b\}.write[\text{Int}], \{a,b\}.put[\text{Int}], \text{abort} \rangle
\]

\[
\text{assert BAD } = (\text{UPDATING } \&\& \text{ failover})
\]

The fluent \text{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 \text{UPDATING} is true, then the system is in a \text{¬normal} state as described in the forgoing.
LTSA analysis

- We can simply prohibit the system from entering this state by adding the following constraint:
  \[
  \text{constraint } NO\_BAD = [] ! BAD \\
  || CON\_SYS = (\neg SYS || NO\_BAD).
  \]

- 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.
  \[
  \text{constraint } REC\_BAD = [] (BAD \rightarrow X \text{ abort}) \\
  || REC\_SYS = (SYS || REC\_BAD).
  \]

- 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 \[[a.init](a.master \land \neg a.updating)\]
   and for b.Server:
   \[[b.init](\neg b.master \land \neg b.updating)\]

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

3 \((a.master \land a.updating)\) \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

 Açiom 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.

 Açiom 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.

 Açiom 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.

 Açiom 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.
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.