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

Jeff Magee

Department of Computing
Imperial College London
180 Queen's Gate
London SW7 2BZ UK
+44 20 75948269

jnm@doc.ic.ac.uk

Tom Maibaum

Department of Computing and Software
McMaster University
1280 Main St. West
Hamilton, Ontario
Canada L8S 4K1
+1 905 525 9140 ext 26627

tom@maibaum.org

ABSTRACT

In this paper we describe initial ideas about modeling and analyzing 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 can be systematically constructed from the specifications in LTSA, 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.

Categories and Subject Descriptors

D.2. [**Software Engineering**]: Requirements/specifications. design tools and techniques. Software/Program verification. Design. Software architectures. F.3. [**Logics and Meanings of Programs**]: Specifying and verifying and reasoning about programs.

General Terms

Design, Reliability, Languages, Theory, Verification.

Keywords

Fault tolerance, Self healing systems, Self managed systems, Software architectures, Program models, Specification, Analysis, Software engineering.

1. INTRODUCTION AND OVERVIEW

At an abstract, or intuitive, level, *self healing*, as part of *self management*, is to do with a system's (application's) capability to recover from anomalous behaviour, or situations, autonomously, without outside intervention (by an operating system, monitoring system, supervisory system, etc). Hence, it is analogous, in some sense, to a fault tolerant system that is capable of recognising the occurrence of a certain category of anomalous situations and has prescriptions for (partial) recovery from such situations. See [19-21]. So, it would appear that the capability to model fault tolerance, to analyse such models and to implement the models (via appropriate recovery algorithms) are important ingredients for understanding self healing and self management. (We do not intend in this work to address issues for self healing systems related to cognitive modelling.)

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 fulfil their obligations and never perform actions that are prohibited (e.g., [1,2]). For systems constructed by a single designer and operating on a stable and reliable platform, this may well be a perfectly reasonable assumption. There are at least two main circumstances in which the assumption must be abandoned. In open agent societies, where agents are programmed by different parties, where there is no direct access to an agent's internal state, and where agents do not necessarily share a common goal, it cannot be assumed that all agents will behave according to the system norms (laws) that govern their behaviour. Agents must be assumed to be untrustworthy, because they act on behalf of parties with competing interests, and so may fail, or even choose not to conform to the society's norms in order to achieve their individual goals. An example of the latter case is in the execution of e-contracts, where a party may break the terms of a contract because it is to that party's advantage to do so. (The agent may not be able to fulfil all of its obligations at the time and manages the penalties involved in not fulfilling some contracts. Or the agent is acting maliciously and wants to subvert the contract.) It is then usual to impose sanctions to discourage norm violating behaviour and to provide some form of reparation when it does occur. (This in itself may be seen as a form of fault tolerance, using a probabilistic/penalty based approach to discouraging faulty behaviour.)

The second circumstance is where agents may fail to behave as intended because of factors beyond their control. This is likely to become commonplace as multi-agent systems are increasingly deployed on dynamic, distributed environments. Agents in such circumstances are unreliable, but not because

SEAMS'06, May 21–22, 2006, Shanghai, China.

they deliberately seek to gain unfair advantage over others. Imposition of sanctions to discourage norm violating behaviour is pointless, though there is a point to specifying reparation and recovery norms. There is a third, less common, circumstance, where deliberate violations may be allowed in order to deal with exceptional or unanticipated situations. An example of discretionary violation of access control policies in computer security is discussed in [3]. In all these cases, it is meaningful to speak of obligations and permissions, and to describe agent behaviour as governed by norms, which may be violated, accidentally or on purpose. In addition to analysing system properties that hold if specifications/norms are followed correctly, it is also necessary to predict, test, and verify the properties that hold if these norms are violated, and to test the effectiveness of introducing proposed control, enforcement, and recovery mechanisms.

Although there is much work on fault tolerance mechanisms and models in the literature, there is a dearth of research addressing analysis and verification of same. In the work described in this paper, we plan to address exactly these issues.

2. MODELLING FAULT TOLERANCE

Key elements of specifying fault tolerance in such systems include the ability to: describe component behaviour, describe component interaction mechanisms (connectors and co-ordination), describe how systems are configured from components and connectors; describe hierarchical component (connector) structuring, characterise behaviour and states that are considered anomalous, describe recovery activity, etc.

An approach to this problem was mooted in the old (1983-90) FOREST requirements engineering projects, which focused on formal requirements specification of multi-component systems. The specification language developed in the project, Modal Action Logic (MAL) [4,5,11], was based on a multi modal action logic, a variant and precursor of dynamic logic, and included a novel use of deontic operators (actions being obliged, permitted or forbidden). The semantics of these operators enabled a distinction in the formalism itself (and in the semantics) between normal or 'good' behaviour and abnormal or bad behaviour. For example, the permission operator is analogous to an enabledness condition. In a conventional implementation, it is impossible to execute an action whose enabledness condition is false. In MAL, an action could be executed anytime, but some executions were defined as normal and others as abnormal. (In fact, there are good reasons for teasing apart enabledness (when is an action available for execution) and permission (when is an action's execution conforming to rules of behaviour prescribed for the system), as well as preconditions (when is an action's effect well defined), but we will not pursue that discussion here.)

The idea in MAL is that states of the system are divided into 'good' states and 'bad' states. Actions executed under the right circumstances lead from good states to good states; we call this normative behaviour. Bad actions lead to bad states, from which recovery is desirable in order to re-enter a normative behaviour mode. Bad situations can also occur when the environment puts the system in a bad state, i.e., it is not the system's 'fault'. For example, the failure of a node in a ring may be modelled in this way: suddenly, something has caused the node to cease functioning. (Compare [4,5] and its notion of *violation* markers.) We need to cover such situations, as self healing systems are certainly going to be 'open', not closed, systems. (The environment may not be able to modify the system, in the sense of adding to it, but it must have the ability to interact with the self healing system in some ways.)

Recovery actions can be specified using the premise that something bad has happened, so some action (or sequence of actions) is necessary (obliged) so as to recover. (In deontic logic, these are called *contrary to duty* structures ([8,9]).) The 'bad' state can be subdivided so that different bad things can be recognised. A later extension of the MAL work began to address fault tolerance specifically ([10]) and demonstrates one way of doing this. For example, the common fault model of {OK, BAD, REALLY BAD} with the appropriate ordering can be modelled with OK corresponding to normative, BAD and REALLY BAD to non normative and the appropriate ordering between them. So, REALLY BAD $\Rightarrow$ (implies) BAD but not vice versa. By modelling abnormal situations with appropriate categories of abnormal states and ordering the bad states logically in an appropriate manner, one can model different fault tolerance mechanisms/models. Then REALLY BAD may imply one set of recovery actions (perhaps to shut the system down), while BAD situations can be recovered by executing appropriate actions. Of course, the detection of what has gone wrong, to determine, for example, whether the system is in a BAD state or in a REALLY BAD state, is itself an interesting and important issue.

The work in [10] extends this idea by using a categorisation of abnormal states that records the non occurrence of obliged actions, the occurrence of forbidden actions, etc. This enables one to build quite sophisticated fault tolerance mechanisms and models and to explore their logical properties. A defect in the approach is that the properties and nature of the logic were not studied. Another is that the formalism used was not component based, but focused on a global system specification. These defects made the usefulness of the approach quite limited. (One might wish to compare this approach to the law governed systems approach of Minsky, [1,2]. The laws in the law governed systems work are unbreachable and act as absolute requirements on system behaviour, so almost like laws of nature. Our idea is that laws are made to be broken, as in human systems of law, and that detecting and punishing/rehabilitating offenders is an important issue. In fact, deontic logic had its origins in the modelling of legal reasoning.)

We need to be able to analyse fault tolerance/self healing models to demonstrate that the mechanism is effective. This is some sort of progress property (say *ProgNorm*): if nothing else bad happens, then eventually normative behaviour is resumed. A variation of this is necessary in which normative is replaced by some appropriate (fault tolerance model dependent) subnormative notion (e.g., from MEDIUM BAD you can only recover to BAD; but if it was only BAD to begin with, then you can recover to OK). The logic to do this kind of reasoning (appropriate inference rules and so on) needs to be developed and attempts made to support the logic in tools like STeP [12] needs to be undertaken.

More recently, the ideas of normative vs non normative behaviour were used in the context of the abstract program design language CommUnity [13,14] to model 'may' and 'must' like conditions on the execution of actions, without using the

normal/abnormal distinction in the semantics [14]. The LTSA tool [16,17], used to model concurrency in the abstract setting of a process algebra, can generate state machines depicting the possible state transitions of the modelled system. In some models, a state labelled *-1* is used to denote a state representing an error condition, from which there is no recovery, i.e., there are no outgoing transitions from this state. This is representative of the usual approach to error, namely that such conditions are relegated to undefinedness in models. One way of seeing deontic logic approaches is to 'open up' this *-1* state and allow it to form some subsystem of the whole state machine, with the addition of transitions back out of this subsystem to 'normal' states.

Closely related work is reported in [18,3]. C+$^{++}$ is an extended form of the action language C+ of [15], a formalism in AI for specifying and reasoning about the effects of actions and the persistence ('inertia') of facts over time. An 'action description' in C+ is a set of C+ rules which define a transition system of a certain kind. Implementations supporting a range of querying and planning tasks are available, notably in the form of the 'Causal Calculator', CCalc. C+$^{++}$ provides two main extensions. The first is a means of expressing 'counts as' relations between actions, also referred to as 'conventional generation' of actions. The second extension is a way of specifying the permitted (acceptable, legal) states of a transition system and its permitted (acceptable, legal) transitions. This second extension is clearly related to the focus of attention in this paper.

3. PROPERTY CHECKING

One of the properties that we might wish to demonstrate of a system is then akin to the need in a conventional LTSA model that the error state, *-1*, is never entered. In the case of our models, this becomes (say *Norm*): in case nothing ever goes wrong, i.e., if only normative actions and scenarios are encountered, then behaviours have specific nice properties, and general ones like never entering the 'error' state. (One might be able to perform an analysis of these so called normative behaviours by reducing the corresponding state machine to one with a single error state, the *-1* state, with no transitions out of it (since they will not be needed in a putative normative scenario) and demonstrating in the usual way that *-1* is never entered. *ProgNorm* may be seen as symmetric: if one collapses all normative states in a model into a single 'good' state called *+1* and looses all transitions out of 'good' states, then the recovery from a bad state may be seen as the property: eventually the system enters the *+1* state. These scenarios present interesting issues to explore with an extended LTSA.

One of the key aspects of modelling and analysing self healing systems is that of dynamic reconfiguration. Recent work, reported in [12], has developed an approach and supporting formalisms for specifying dynamically reconfigurable, multi component, and hierarchically organised systems. (The work can actually be seen as a further extension of the family of languages initiated by MAL.) This language for specifying and reasoning about software architectures is a good starting point for the proposed work. One can extend the language and its semantics with the mechanisms for specifying normal and abnormal behaviours. Interesting problems include the exploration of what it means to combine/co-ordinate two components in this setting. An underlying principle here is that components are co-ordinated by identifying an action in one with an action in the other (defining a single action from the system's point of view). What happens to the deontic constraints associated with the individual actions? It would appear reasonable to say that permission for the co-ordinated single action requires permission for each of the constituent actions. Exploring this space and defining the required reasoning mechanisms is a major goal of the research described in this paper.

So the objective here is to extend the architecture description language proposed in [12] with such general mechanisms, analysing how component descriptions with this fault tolerant aspect can be put together to build systems. Typically, two components that synchronise on an action will have joint permission to execute the action if they each individually have permission to execute it separately; they will have an obligation to do the joint action if at least one of them has an obligation to execute it. Looking at the running examples and modelling various fault tolerance mechanisms, reasoning about them and perhaps coming up with new ones would be good things to try.

4. AN ILLUSTRATIVE CASE STUDY

4.1 An engineering Method

[16] proposes an engineering model for constructing concurrent systems. The method consists of building LTSA models of the system, analyzing the models for desirable properties and then using standard transformations from the LTSA models to corresponding Java programs implementing the models. In fact LTSA models are approximations since they abstract data values and focus on interaction. They are event-based models of the intended system behaviour.

What we are proposing is more akin to the ideas behind the Model Driven Architecture (MDA) approach ([22]). We propose an engineering approach based on the following main steps:

1. Build a software architecture based specification of the architecture of the system, based on some well defined requirements, including appropriate fault tolerance (and self management) mechanisms. Analyse the specification.
2. Build an LTSA model of the specification (or crucial parts of the specification) as an intermediate design, actually an abstract program design, lying between the architectural specification and the code level implementation. The model may be built in a standard way, given the style of architecture used in step 1. Analyse the model to check that it has desirable properties, including properties that demonstrate that the model conforms to the specification.
3. Use standard 'transformations' to turn the high level design into corresponding programs in the chosen programming language. Analyse the program for conformance with previous models.

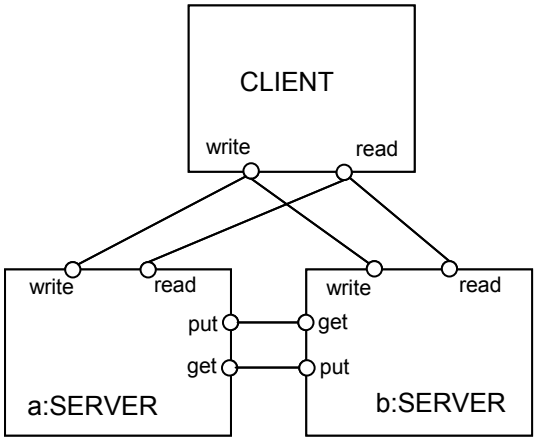
We will illustrate here only the first two steps. There is a lot of work yet to be done to turn these initial ideas into a proper engineering method, as noted in the conclusions. We intend that the method be quite general. Any system in which we need to distinguish between normal or ideal behaviour and abnormal or subideal behaviour is fair game for the method. For traditional

fault tolerance, we are not proposing magic solutions to fault tolerance challenges. We are proposing a way of characterizing and analyzing proposed mechanisms, together with the possibility of standardizing implementations, cookbook solutions, via the modeling and coding steps. The same might be said about ideal versus subideal behaviour, as in performance tuning.

As with any approach based on formal reasoning, there are practical limitations due to limitations of theorem provers and model checkers, like LTSA, so another important challenge is to make useful analysis feasible. Certainly, since LTSA was never intended was never intended for analysis of completely faithful models of systems, there is some hope of building models of fault tolerance mechanisms that are useful and feasible.

4.2 The Case Study

We present a 'simple' example of fault tolerance. The diagram below represents a model of a resilient server in which a slave server keeps a replica of the state, which is updated by the master server in response to client write requests. The state maintained by the system is abstracted to a single integer value (which will be further restricted to a bounded value in the LTSA representation below).



The client is able to read and write to the master server. In this simple example, the client will know which server it is talking to, by maintaining an internal variable pointing to it. (Of course, in a more sophisticated version of this example, we would model a dynamically reconfigurable architecture, where the client was talking to 'the' server and which server it was actually talking to would be determined by the topology of the reconfigurable architecture.)

To build this architecture, we now need to specify the components and then describe their interconnection. In the MAL approach, a component is described in terms of its local memory (attributes), local operations (actions) and MAL axioms describing the properties of the attributes and actions, such as the effect of actions on attributes. These properties may also include deontic axioms describing what should normally happen (or not happen). There may also be axioms describing what happens when we are in an abnormal situation. So, in the example we are describing, we want the master server to copy any updates by the client to its internal variable to the slave server before allowing any other operations to be executed. We will model a possible failure of the master server by an operation called failover, which has the effect of switching the roles of master and slave. If this 'action' occurs after a write by the master but before a put to the slave, then the client will read the wrong value and something will have gone seriously wrong. Of course, a failure can happen between the write and the put, and if it does, corrective action needs to be taken. We model this corrective action simply by requiring an immediate execution of the failover operation, again switching the servers in their master/slave roles. So, the client cannot get to read the wrong server because it will be 'forced' to participate in the switchover before it does anything else. (Of course, the client may abnormally do something wrong, creating an even worse situation, but we do not model here what happens.)

We first describe the Client:

```
component Client
Attributes
    c_val:int, c_master:{a,b},
    ready_to_write:bool, error:bool
Actions
    c_init, c_write(int,c_master),
    c_read(int,c_master), switch, abort
Axioms
1. [c_init](c_master=a ∧ val=0 ∧
   ready_to_write ∧ ¬error)
2. (ready_to_write ∧ c_master=m ∧ ¬error)
   → [write(val,m)]¬ready_to_write
3. (¬ready_to_write ∧ c_master=m ∧ ¬error
   ∧ val=x) → [read(y,m)]((x≠y → error) ∧
   (x=y → (ready_to_write ∧ val=x+1)))
4. c_master=a → [switch]c_master=b
5. c_master=b → [switch]c_master=a
6. ¬ready_to_write → [switch](¬normal ∧
   Obl(abort))
7. ¬normal → [abort](ready_to_write ∧
   normal)
```

The specification above has attributes `val` for recording the current value to be sent to the server, `c_master` to record which server the client is talking to, `ready_to_write` to record the fact that a write has occurred and a read now needs to happen, and `error` to record that a value has been read that was not the one last written. The actions are `c_init` to initialise the client component (we ignore issues related to this having to occur only at the beginning), `c_write` to write a value (to the master, as we will see below where we define how the actions in the various components are coordinated), ditto `c_read` and `switch` to record the switch of the master slave relationship. There is also an `abort` action that is used in recovery mode.

The axioms describe the required behaviour. Axiom 1 simply says that, after the initialisation action, the values of the various attributes are set to default values. Axiom 2 that 'if the client is ready to write and the master is m and the client is not in an error state, then the effect of `write(val,m)` is simply to set `ready_to_write` to false. (Of course, its 'real' effect will be achieved by coordinating this action with the write action in the server, resetting the local value stored there to that of `val`).

Axiom 3 describes the effect of a read operation, saying that `ready_to_write` must be false in order for the `read`'s effect to be defined. The 'post condition' requires that, if the value read is not the one expected, then an error is recorded, and otherwise that the value stored in the client is incremented and the client is again ready to start the write/read cycle again. Axioms 4 and 5 say that when switch occurs, it causes a change in the server to which the client is talking. So, the client recurrently writes and reads, going into an error state if the wrong value is read and allowing server switches at any time.

Axioms 6 and 7 deal with the abnormal situation when a switch occurs between a write to the master server and a consequent read by the client, intended to be from the same server. Then, we enter an abnormal state, recorded by setting **`normal`** to false. **normal** is a logical connective, like conjunction or disjunction, but having no arguments. It is used to capture violations of deontic constraints. (The ability to capture different kinds of faults and recovery mechanisms would require a refinement of this connective into a richer, structured set of such connectives, to enable us to distinguish different fault scenarios from each other.) If the Client is in an abnormal state, then the recovery action is to do an `abort`, so that a write to the (new) master is forced. Hence, no bad value should be read (as long as nothing else goes wrong).

We now describe the server components. We describe both at the same time by using the usual 'dot' notation. The descriptions are exactly the same for the two servers, except for the effect of the initialisation actions, setting one to be the default master and the other to be the default slave.

```
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 ∧ ¬a.updating)
    (and for b.Server:
     [b.init](¬b.master ∧ ¬b.updating)
2.  (a.master ∧ ¬a.updating) →
    [a.write(val)](a.val=x ∧ a.updating)
3.  (a.master ∧ a.updating) →
    [a.put(val)]¬a.updating
4.  a.master → [a.failover]¬a.master
5.  ¬a.master → [a.get(x)]a.val=x
6.   (For b.Server, we have axioms 2-5 with
    'a' replaced by 'b'.)
```

So, when we initialise `a.Server`, it is made the master and it is not in the middle of a 'write-put' transaction. (Symmetrically, `b.Server` is set to be the slave.) 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`.

We can now configure the system by defining how the various components are coordinated. The formal definition of this is in terms of the colimit construction in the category of component specifications and property preserving morphisms between them. Two (or more) components are coordinated via this mechanism by: identifying a subset of the attributes of one component with corresponding attributes of the other component (shared memory) and by identifying a subset of the actions of one component with corresponding actions of the other component (synchronised actions, as in synchronised message passing where sends in one component are synchronised with receives in the other). (In our server example, we do not use the shared memory idea.)

For example, the client action `switch` is synchronised with `a.failover` in `a.Server` and with `b.failover` in `b.Server`. (Again, this simple mechanism obviates the need for describing the system in terms of reconfigurable architectures.) So both servers and the client do the switchover together (and never separately). Similarly, we coordinate `c_write(x,a)` and `c_read(x,a)` of `Client` with `a.write(x)` and `b.read(x)`, respectively, of `a.Server`. Ditto for `b`. Now the servers are coordinated by synchronising `a.put` with `b.get` and *vice versa*. This reflects the box and line diagram above.

Now, if all goes well, then we should expect that **normal** always holds in the `Client` and we have no error state. So the property

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

says that if we never have a violation of the deontic constraints, then we never encounter an error state in the client. Moreover,

$$\neg\textbf{normal} \wedge \textbf{no\_further\_violation} \rightarrow \Diamond\textbf{normal}$$

says 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. (Again, we do not get into the definition of this no further violation marker in this paper.)

So, having described our system specification and reasoned about its properties, we may wish to implement the resulting specification. We can take a step in this direction by building a programming model, albeit abstract, of the specification and gaining confidence about a design. LTS and the LTSA tool provide just such a modelling environment.

4.3 The LTS Model

The Labelled Transition System Analyzer (LTSA) [16] is a finite state verification tool for modelling and analyzing the behaviour of systems represented by labelled transition systems. In the LTSA, a system is modelled as a set of processes described in Finite State Processes (FSP), a process algebra notation. The tool permits the analysis of systems with respect

to propositional linear temporal logic properties specified in Fluent Linear Temporal Logic (FLTL) [17].

Note that, in the models below, attributes in the specification above become parameters of the corresponding state machine definition. Also note that types like **int** have to be made into bounded versions, as LTSA is a finite state analyzer. The behavior of a SERVER process is modelled in FSP in the following, where "->" means action prefix and "|" means choice.

```
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]
  ).
```

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.

We model a CLIENT as shown below:

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

The client offers to read or write to either server "a" or server "b", however, only the master server will accept these actions. A CLIENT may be aborted which 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. This local property is, of course, only preserved for the situation in which there is only a single client. 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 specification. Despite this, 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
```

This happens because, whereas the switch from master to slave and slave to master is atomic, the state as seen by the client is not. In particular, as the above trace shows, 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 action 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**.** We can simple 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. What this model does not do is reflect the possibility implicit in the specification 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.

5. CONCLUSIONS

We have outlined an approach for modeling and analyzing fault tolerance (and self management) mechanisms in multicomponent, distributed systems. We use a modal action logic formalism, augmented with deontic operators, to describe

normal and abnormal behaviour. By using the operators introduced in the deontic component of the formalism, we can model the existence of abnormal behaviour and prescribed recovery mechanisms for specific abnormal situations. Our objective is to provide a systematic way of turning these specifications into (approximating) abstract programming models. We have shown in the preceding example a translation of component specifications into finite state models that can be model checked for properties related to fault tolerance. This is very much an initial step that shows how some of the ideas we have proposed for describing normal and abnormal behaviour can be expressed in a conventional model checking tool.

Clearly, we are at the beginning of a long road in developing an appropriate version (or versions) of the action logic, extending LTSA to deal directly with the encoding of the required deontic constraints (and, hence, the related fault tolerance mechanisms). The relationship to analyzers like that supporting C++ are also of interest, as they may have features that can be usefully incorporated into LTSA (and *vice versa*).

Acknowledgements: This work was partially supported by an EPSRC grant to the1st author to support a Visiting Fellowship for the 2nd author. McMaster University and NSERC (Canada) also generously supported this research, as did the EU via the SENSORIA project.

6. REFERENCES

[1] Ao, X., Minsky, N., Nguyen, T., Ungureanu V. Law-Governed Communities Over the Internet. In *Proceedings of Coordination 2000: Fourth International Conference on Coordination Models and Languages*, LNCS 1906, Springer-Verlag, 2000, 133-147.

[2] Minsky, N., Ungureanu V. Law-Governed Interaction: A Coordination & Control Mechanism for Heterogeneous Distributed Systems. *ACM Transactions on Software Engineering and Methodology* (*TOSEM*), 9, 3, (July 2000) 273-305.

[3] Rissanen, E., Sadighi Firozabadi B., Sergot M. J. Towards a mechanism for discretionary overriding of access control (position paper). In *Proceedings of 12th International Workshop on Security Protocols*, Cambridge, 2004.

[4] Khosla, S., Maibaum, T. S. E. The Prescription and Description of State Based Systems. In *Proceedings of the Workshop on Temporal Logic in Specification*, Banieqbal, B., Barringer, H., Prueli, A., eds, LNCS 298, Springer Verlag, 1989, 243-294.

[5] Jeremaes, P., Khosla, S., Maibaum, T. S. E. A Modal (Action) Logic for Requirements Specification. In *Proceedings of Software Engineering 86*, IEE Computing Series 6, Barnes, D., Brown, P., eds, Peter Peregrinus, 1986, 278-294.

[6] Wieringa, R., Meyer, J.-J. Ch., Weigand, H. Specifying Dynamic and Deontic Integrity Constraints. *Data Knowl. Eng.* 4, 1989, 157-189.

[7] Meyer, J.-J. Ch., Weigand, H., Wieringa, R. A Specification Language for Static, Dynamic and Deontic Integrity Constraints. *MFDBS 89, 2nd Symposium on Mathematical Fundamentals of Database Systems*, LNCS 364, Springer-Verlag, 1989, 347-366.

[8] Prakken, H., Sergot, M. Contrary-to-Duty Obligations. *Studia Logica* 57,1, 1996, 91-115.

[9] Carmo, J., Jones, A. Deontic logic and contrary-to-duties. In *Handbook of Philosophical Logic - Second Edition, Volume 3: Extensions to Classical Systems 2*, Gabbay, D., Guenthner, F., eds, Kluwer, 2000.

[10] Kent, S. J. H., Maibaum, T. S. E., Quirk, W. J. Formally Specifying Temporal Constraints and Error Recovery. In *Proceedings of the 1st IEEE International Symposium on Requirements Engineering*, IEEE CS Press, 1993, 208-215.

[11] Ryan, M., Fiadeiro, J. L. L., Maibaum, T. S. E. Sharing Actions and Attributes in Modal Action Logic. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, Ito, T., Meyer, A., eds, LNCS 526, Springer-Verlag, 1991, 569-593.

[12] Aguirre, N., Maibaum, T. S. E. Some Institutional Requirements for Temporal Reasoning on Dynamic Reconfiguration of Component Based Systems. In Proceedings of the International Symposium on Verification (Theory and Practice), Dershowitz, N., ed, Festschrift celebrating Zohar Manna's 64th Birthday, Taormina, Italy, LNCS 2772, Springer-Verlag, 2003.

[13] Fiadeiro, J. L. L., Maibaum, T. S. E. Categorical Semantics of Parallel Program Design. *Science of Computer Programming*, 28, 1997, 111-138.

[14] Barreiro, N., Fiadeiro, J. L. L., Maibaum, T. S. E. Politeness in Object Societies. *Information Systems – Correctness and Reusability*, Wieringa, R., Feenstra, R., eds, World Scientific, 1995, 119-134.

[15] Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H. Nonmonotonic Causal Theories. *Artificial Intelligence*, 153, 2004, 49-104.

[16] Magee, J., Kramer, J. *Concurrency: State Models & Java Programs,* John Wiley and Sons, 1999.

[17] Giannakopoulou, D., Magee, J. Fluent model checking for event-based systems. Proceedings of ESEC/SIGSOFT FSE 2003, ACM Press, 2003, 257-266.

[18] Sergot, M. J. Modelling Unreliable and Untrustworthy Agent Behaviour. In *Monitoring, Security and Rescue Techniques in Multiagent Systems*, Dunin-Keplicz, B., Jankowski, A., Skowron, A., Szczuka, M., eds, Springer-Verlag, 2005, 161-178.

[19] Guerra, P. A. de C., Rubira, C., Romanovsky, A., de Lemos, R. A Fault-Tolerant Software Architecture for COTS-Based Software Systems. In *Proceedings of the Joint European Software Engineering Conference (ESEC) and 11th SIGSOFT Symposium on the Foundations of Software Engineering (FSE-11)*, ACM Press, 2003, 375-382.

[20] Guerra, P. A. de C., Rubira, C., de Lemos, R. *A Fault-Tolerant Software Architecture for Component-Based Systems*. LNCS 2677, Springer-Verlag, 2003, 129-149.

[21] Romanovsky, A., Dony, C., Knudsen, J. L., Tripathi, A. (eds). *Advances in Exception Handling Techniques*, LNCS 2022,Springer-Verlag, 2001, 289 p.

[22] Lano, K. *Design for Change: Advanced System Design with Java, UML and MDA*, Butterworth, 2005.