Formalizing Dependability Mechanisms in B: From Specification to Development Support

F. Tartanoglu, V. Issarny
INRIA Rocquencourt
France
Galip-Ferda.Tartanoglu@inria.fr
Valerie.Issarny@inria.fr

N. Levy
Laboratoire PRISM
Université de Versailles
Saint-Quentin, France
Nicole.Levy@prism.uvsq.fr

A. Romanovsky
School of Computing Science
University of Newcastle upon Tyne
UK
Alexander.Romanovsky@newcastle.ac.uk

Abstract

The CA action concept has been proven successful for building dependable distributed systems due to its support for error recovery for both competitive and cooperative concurrent actions. This paper introduces the formal specification of dependability mechanisms offered by CA actions using the B formal method, from which an XML-based language is derived. The resulting language then allows developing dependable systems, where the B formal specification is refined to obtain an implementation of the associated runtime support.

1. Introduction

Dependability of systems is defined by the reliance that can be put on the service they deliver. Developing distributed systems that are dependable is recognized as a complex task, requiring adequate mechanisms for dealing with the occurrence of failures. Coordinated Atomic Actions (CA actions) [8] provide a general structuring mechanism for developing dependable systems through the exploitation of atomic actions and transactions. The composition of CA actions [6] further extends the base CA action model for developing open distributed systems.

Several applications have proven that CA actions are effective for building dependable concurrent systems [9, 2]. Formalization of applications based on CA actions, using Petri nets and temporal logic, further enables to prove the applications' dependability properties through model-checking [7, 5]. However, dependability properties are proved with respect to a specific application.

Our approach aims at providing a language for developing distributed systems using dependability mechanisms that are formally specified and implemented. Towards that goal, we provide a formal specification of the dependability mechanisms associated with the CA action concept using the B formal method, from which we derive an XML-based language to be used to develop dependable applications.

The paper is structured as follows. Section 2 briefly presents CA actions and their composition. Sections 3 and 4 then introduce the B formal specification of CA actions, discussing in particular the specification of dependability mechanisms offered by CA actions. Definition of the resulting XML-based development support follows in Section 5. Finally, Section 6 concludes, summarizing our contribution and discussing areas for future work.

2. Architecting Dependable Systems with Coordinated Atomic Actions

CA Actions

The CA actions [8] are a structuring mechanism for developing dependable concurrent systems through the generalization of the concepts of atomic actions [3] and transactions [4]. Atomic actions are used for controlling cooperative concurrency among a set of participating processes and for realizing coordinated forward error recovery using exception handling; transactions are used for maintaining the coherency of shared external resources that are competitively accessed by concurrent actions. Each CA action is designed as a multi-entry unit with roles activated by action participants, which cooperate within the action. A transaction is started upon each first access to a given external object by a CA action participant and it terminates at the end of the CA action. A CA action terminates with a normal outcome if no exception has been raised or if an exception has been raised and handled successfully; all transactions on external objects are then committed. If a participant raises an exception inside an action and if the exception cannot be handled locally by the participant, the exception is propagated to all the other participants of the CA action for *coordinated error recovery*[1]. If coordinated recovery fails, the CA action terminates with an exceptional outcome. An exception is then signalled by the CA action and transactions on external objects are aborted.

CA actions can be designed in a recursive way using action nesting. Several participants of a CA action can enter into a *nested CA action*, which defines an atomic operation inside the embedding CA action. Accesses to external objects within a nested action are performed as nested transactions so that if the embedding CA action terminates exceptionally, all sub-transactions that were committed by nested actions are aborted as well. A CA action participant can only enter one nested action at a time. Furthermore, a CA action terminates only when all its nested actions have completed. Note that if the nested action terminates exceptionally, an exception is signalled to the containing CA action.

As an illustration, Figure 1 depicts a CA action *A1* that is entered by participants *P1-P3* and that comprises two nested CA actions, *A11* and *A12*; transaction are further executed on external objects. An exception raised by participant *P2* causes the CA action to enter an exceptional state, as showned by the greyed box, where the participants cooperate for handling the exception.

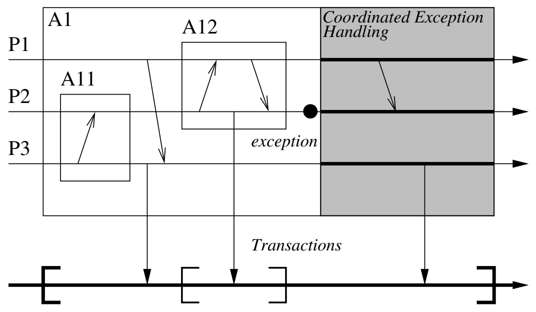

Figure 1. Coordinated atomic actions

CA actions mainly focus on structuring concurrent systems and on providing their fault tolerance by exception handling. One of the main intentions behind CA actions is to employ them as the mechanism for structuring complex distributed applications: they promote recursive view on system execution with abstracting away both normal and abnormal behaviour of the low level software.

CA Actions Composition

Composing CA actions allows the design of open distributed systems built out of several CA actions. Unlike classical action nesting where a subset of action participants enters into a nested action, *composed CA actions* are autonomous entities with their own participants and external objects. In this model, a participant of a CA action can dynamically initiate the creation of a composed CA action (or dynamically nested action).

The internal structure of a composed CA action (i.e., set of participants, accessed external objects and participants' behavior) is hidden from the calling CA action, which only has an access to the composed CA action's interface. A participant that calls a composed CA action enters a waiting state in a way similar to a synchronous RPC. The participant then resumes its execution according to the outcome of the composed CA action. If the composed CA action terminates exceptionally, its calling participant raises an internal exception which is possibly locally handled. If local handling is not possible, the exception is propagated to all the peer participants for coordinated error recovery. Note that unlike static nesting, when a composed CA action has terminated with a normal outcome, an abort operation of the containing CA action does not automatically compensate effects of the composed one; specific handling must be performed at a higher level, e.g., a composed action can be initiated to abort/compensate actions on external objects if needed.

Figure 2 illustrates the use of nested and composed CA actions, considering a travel agency system. The top-level CA action comprises the *User* and the *Travel* participants; the former interacts with the user while the latter achieves joint booking according to the user's request. The CA action has further access to the *Banking System*. In a first step, the *User* participant requests the *Travel* participant to search for a trip. This leads the participants to enter the nested action *SearchTrip* in which the *Travel* participant invokes a composed action comprising the *Hotel* and the *Flight* participants. The external objects accessed by those participants are the hotel and flight booking system. The *SearchTrip* action, if successful, returns a list of possible trips. Then, according to the *User*'s selection, the *BookTrip* nested action is executed, where another composed CA action is initiated to book the given trip. If an exception is raised within the composed CA action (e.g., *no_flight_available* for a given destination) and if it cannot be handled internally, the composed action terminates exceptionally by aborting all transactions on external objects and signals a failure exception to the higher level.

3 Specifying CA Actions in B

The B method

B is a complete formal method [1] that supports a large part of the development life cycle, from abstract specification to implementation. The B formal method is a model-based method, which is based on set theory and predicate

[1]If several exceptions have been raised concurrently they are resolved using a resolution tree imposing a partial order on all action exceptions, and the participants handle the resolved exception [3].

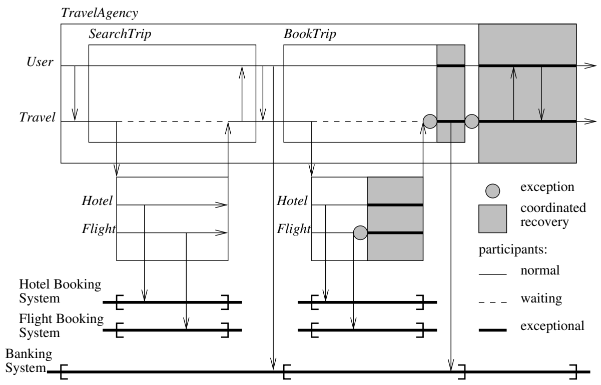

Figure 2. CA actions composition

logic and extended by generalized substitutions. B specifications are represented by abstract machines encapsulating operations and states. Generally speaking, the B method allows us to define *abstract machines* and *refinements* over them. During the refinement nondeterminism is reduced and preconditions are relaxed, but the interfaces of the operations remain the same. At the end of the refinement process, an *implementation* can be written, which corresponds to an executable code.

Proofs are an essential part of the model: it should be proven that all operations preserve the invariants of the machine, and that the implementations and refinements preserve the invariants and the behaviour of the initial abstract machine. There are various tools that help writing and proving B specifications. The main of them are B-Tool[2] and Atelier B[3]. Both tools include a type checker, an animator, a proof obligation generator, theorem provers, code translators and documentation facilities. Atelier B has been used in our investigation, however the notation we used is compatible with B-Tool.

Modelling CA Actions

Our goal in providing the B specification of CA actions is to offer a general framework that can be instantiated to describe the implementation of a specific system that is developed using CA actions. The framework thus defines the dependability properties associated with CA actions, which will be enforced for any system based on them.

The B formal specification of CA actions is given by the *CAACTIONS* abstract machine. The machine extends machines *OBJECTS* and *PARTICIPANTS*, which respectively describe external objects that can be accessed or modified by a CA action, and the participants of a CA action (see Figure 3). The *CONST* machine further contains global declarations and is seen by all the other machines. In the remainder, we introduce the main elements of the B specification, focusing on dependability properties associated with CA actions; the interested reader may find the overall specification at `http://www-rocq.inria.fr/~tartanog/dsos/`.

[2]http://www.b-core.com/btool.html
[3]http://www.atelierb.societe.com

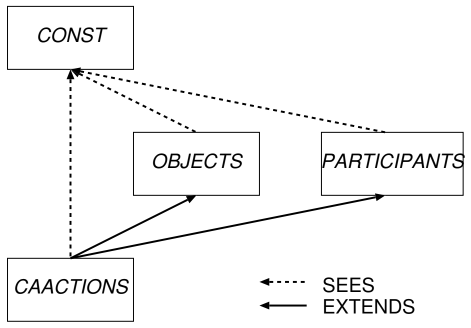

Figure 3. Structure of the B specification

The state of the *PARTICIPANTS* abstract machine characterizes the participants of a CA action as follows:

- The *PARTICIPANT* set is declared in the *CONST* machine and represents all possible participants that can be involved in a CA action.

- A participant that is activated by entering in a CA action is included in the subset *participant* of *PARTICIPANT* and removed at the end of the action:

  $participant \subseteq PARTICIPANT$

- Each participant enters in a sequence of modes (referred to as *state*), which can be normal, exceptional when an exception has been raised, or waiting if the participant invokes a composed CA action and is blocked until the action's termination:

  $participant\_state \in PARTICIPANT \rightarrowtail\!\!\!\!\!+ \ \mathbf{seq}(PARTICIPANT\_STATE)$

- Each participant has a value (local variables) that is logged for later use in case of backward recovery:

  $participant\_value \in PARTICIPANT \nrightarrow VALUE \wedge$
  $initial\_values \in PARTICIPANT \nrightarrow \mathbf{seq}(VALUE)$

The *CAACTIONS* abstract machine defines operations associated with the execution of CA actions: creation, termination, nesting and composition of CA actions, message exchange between participants, and exception handling. An abstract set *CAACTION* of all possible CA actions is introduced together with subset *caaction* of the CA actions that are running at a given state of the system. Three types of CA actions are distinguished: the top-level, nested and composed CA actions; two variables are used to memorize the nested and the composed CA actions:

$is\_nested \in caaction \leftrightarrow caaction \land$
$is\_composed \in participant \nrightarrow caaction$

The state of the *CAACTIONS* abstract machine is defined by the following attributes:

- CA actions have a mode (referred to as *state*) that can be normal (if all the participants are in normal mode) or exceptional (if all the participants are in exceptional mode).

  $CAACTION\_STATE = \{caa\_normal, caa\_exceptional\}$
  $caaction\_state \in caaction \rightarrow CAACTION\_STATE$

- Each CA action has a set of participants and each of them participates to a sequence of nested CA actions:

  $participant\_of\_caaction \in caaction \rightarrow \mathcal{P}\ (participant)$
  $caaction\_of\_participant \in participant \rightarrow \mathbf{seq}(caaction)$

- CA actions access several external objects:

  $caaction\_ext\_objects \in caaction \leftrightarrow objects$

Several invariant properties of the CA actions have been identified and specified. They are written as constraints on the variables of the abstract machines. The state transformations associated with the execution of CA actions are further defined by the following preconditioned operations[4]:

- create_{main,nested,composed}_caaction: initiates a CA action, either top-level, nested or composed;
- {send,recv}_message(participant,participant,message): sends or receives a message from one participant to a peer participant;
- {read,write}_object(participant,object,{function}): reads, writes the value of an external object;
- raise_exception(participant,exception), propagate_exception(participant): raises, propagates an exception;
- abort_{main,nested,composed}: aborts a CA action, sending an abort message to all its external objects;
- terminate_{main,nested,composed}_{normal,exceptional} (caaction): terminates a CA action, either in a normal or in an exceptional state.

Invariant properties and operations are specified in such a way that properties of the dependability mechanisms associated with CA actions are enforced.

[4]Braces are used to denote multiple distinct operations.

4 Dependability Properties

The dependability mechanisms embedded within CA actions fall into three categories: (i) transactional access to external objects, (ii) atomicity of CA actions, and (iii) coordinated exception handling.

Transactions on External Objects

Access to external objects within CA actions are performed according to classical nested transaction rules. The operation that creates a top-level CA action initiates the transaction on external objects associated to the CA action:

```
    add_objects(obj) =
PRE
    $obj \subseteq OBJECT \land obj \cap object = \emptyset$
THEN
    $values := values \lhd obj \times \{begin\}\ ||$
    $object := object \cup obj$
END;
```

Participants *setpar* of nested CA action *caa1* can only access subset *setobj* of external objects associated to containing CA action *caa2*. This constraint is ensured with the following precondition of the *create_nested_caaction(caa1,caa2,setpar,setobj)* operation:

$$\forall\ obj.(obj \in setobj \Rightarrow obj \in caaction\_ext\_objects[\{caa2\}])$$

Then, the operation initiates a nested transaction on the external object (*add_nested_object(setobj)*). When a (possibly nested) CA action terminates its execution normally (*terminate_caaction(caaction)*), it commits transactions on external objects:

$$terminate\_transaction(caaction\_ext\_objects[\{caaction\}],commit)$$

On the other hand, if the CA action terminates exceptionally or aborts, all the transactions that it initiated on external objects are aborted as well:

$$terminate\_transaction(caaction\_ext\_objects[\{caaction\}],abort)$$

Note that nested transactions are aborted recursively by the underlying transactional support of external objects.

Atomicity of CA Actions

Cooperation of participants is encapsulated inside atomic computation units using nested or composed CA actions.

The following invariant property states that participants of nested CA action *caa1* are also participants of containing action *caa2*:

$\forall$ (*caa1*, *caa2*).((*caa1* $\in$ *caaction* $\wedge$ *caa2* $\in$ *caaction* $\wedge$
(*caa1*,*caa2*) $\in$ *is_nested*)
$\Rightarrow$ *participants_of_caaction*(*caa1*)
$\subseteq$ *participants_of_caaction*(*caa2*))

In the case when a participant invokes a composed CA action, participants *setpar* of the composed CA action must not be involved in any other CA action:

setpar $\cap$ *participant* = $\emptyset$

Communication between participants *p1* and *p2* within a CA action is realized by message exchanges. Preconditions of the *send_message* operation set the rules of message exchange that is only allowed between participants of the same (possibly nested) CA action. The participants must be in the same state (normal or exceptional). Finally, a participant that is in a waiting state (i.e., waiting for a composed CA action to terminate) cannot send or receive a message:

caaction_of_participant(*p1*) = *caaction_of_participant*(*p2*) $\wedge$
last(*participant_state*(*p1*)) $\neq$ *waiting* $\wedge$
last(*participant_state*(*p1*)) = **last**(*participant_state*(*p2*)) $\wedge$

Rules of nesting and composition are further specified with the following preconditions of the CA action termination operation stating, that a CA action terminates when all embedded nested and composed CA actions have terminated:

caaction $\notin$ **ran**(*is_nested*) $\wedge$ *caaction* $\notin$ **ran**(*is_composed*)

Furthermore, a participant can only enter one sibling nested CA action at a time, which means that all participants in *setpar* willing to enter a nested CA action are in the same containing CA action:

card(**ran**({ *p*, *c* | *p* $\in$ *setpar* $\wedge$ *c* $\in$ *CAACTION* $\wedge$
c= **last**(*caaction_of_participant*(*p*)) })) = 1

Finally, the participants willing to enter a nested CA action must all be in the same state, normal or exceptional:

card(**ran**({ *p*, *state* | *p* $\in$ *setpar* $\wedge$
state $\in$ *PARTICIPANT_STATE* $\wedge$
state= **last**(*participant_state*(*p*))})) = 1 $\wedge$
$\forall$ (*p*).(*p* $\in$ *setpar* $\Rightarrow$ **last**(*participant_state*(*p*)) $\neq$ *waiting*)

Coordinated Exception Handling

The following invariant ensures that a CA action is set to an exceptional state if all of its participants are in the exceptional state. Note that the participant can be in a waiting state following a call to a composed CA action, in which case case the test is performed on the last state of the participant before the call:

$\forall$ (*caa*). (*caa* $\in$ *caaction* $\wedge$ *caaction_state*(*caa*)= *caa_exceptional*
$\Rightarrow$ $\forall$ (*p*).(*p* $\in$ *participant_of_caaction*(*caa*)
$\Rightarrow$ ((**last**(*participant_state*(*p*)) $\in$ *EXCEPTIONAL_STATE*) $\vee$
((**last**(*participant_state*(*pa*)) = *waiting* $\wedge$
last(**front**(*participant_state*(*p*))) $\in$ *EXCEPTIONAL_STATE*)))))

Exception raising and propagation (to other participants) is realized by two operations defined in the *CAACTIONS* machine. The *raise_exception* operation requires that the participant and the CA action are in the normal state, and sets the participant's state to exceptional:

```
raise_exception (p, exception) =
PRE
    p ∈ participant ∧
    exception ∈ EXCEPTIONAL_STATE ∧
    last(participant_state(p)) = normal ∧
    caaction_state(last(caaction_of_participant(p))) = caa_normal
THEN
    set_participants_state({p}, exception)
END;
```

The *propagate_exception* operation is then called to propagate the exception to all participants of the CA action.

If a CA action terminates in an exceptional state, all transactions on external objects are aborted. If this CA action is a nested or composed one, then the participant in the containing CA action raises an exception by calling the *raise_exception* operation (i.e., the exception is *signalled* to a higher level).

5. From the B Specification to the Development Support

In order to have an implementation of the CA action's run-time support, the abstract machines are refined. At the end of the refinement process, we have a set of executable codes that correspond to the implementation of the operations defining the B machines, offered as a programming library. Note that, when implementing the CA actions, some existing libraries such as drivers for running transactions are used. For all these libraries, what is usually known is the interfaces of the offered methods. In order to be able to prove the correctness of the implementation it would be necessary: (i) to have in addition the formal specification of the behavior of these methods and (ii), to prove that the refinements of the machines that use these methods are correct (in the B sense). During the refinement, the nondeterminism will be reduced (e.g., by introducing of message queues).

The preconditions have to be relaxed in order to take into account all the possible cases.

We introduce an XML-based language derived from the B specification in order to provide to the developer, which may not necessarily have a B knowledge, a convenient declarative language for building CA action-based systems.

Each XML document defines a main CA action which contains composed and nested CA actions, where composed CA actions are defined in a distinct document. The external objects are also declared.

```
<caaction name="nmtoken"?  >
  <composedActions> ?
    <action name="qname" /> *
  </composedActions>
  <nestedActions> ?
    <nested name="nmtoken" /> *
  </nestedActions>
  <external> ?
    <object name="nmtoken" /> *
  </external>
```

Each participant is then declared within the CA action, defining its local variables, which correspond to values used in the B specification (*participant_value*), and its behaviour composed of a normal and an exceptional parts. A participant executes this exceptional part when the CA action is in an exceptional state, which means that the exception raised by a participant has been automatically propagated to all of them.

```
<participants>
  <participant name="nmtoken"> +
    <var>
      <element name="nmtoken" type="qname" /> *
    </var>
    <behavior>
      <normal>
        Statements *
      </normal>
      <exceptional handle="qname"> *
        Statements *
      </exceptional>
    </behavior>
  </participant>
</participants>
```

The statements declared in the behaviour part of the participant's definition describe a sequence of operations to be executed. Each operation corresponds to the implementation of an operation of the *CAACTIONS* machine

```
Statements:
<invoke action="qname" input="qname"?
      output="qname"?  />
⇒ create_composed

<send rcpt="qname" input="qname" />
⇒ send_message

<recv from="qname" output="qname" />
⇒ recv_message

<call rcpt="qname" input="qname"?
      output="qname"?  />
⇒ {read,write}_object

<assign element="qname" value="XPATH" />
⇒ set_value

<raise exception="qname" message="qname"?  />
⇒ raise_exception

<nest nestedaction="qname">
  <behavior>
    <normal>
      Statements *
    </normal>
    <exceptional handle="qname"> *
      Statements *
    </exceptional>
  </behavior>
</nest>
⇒ create_nested
```

The above language enables the development of systems using CA actions. It hides the details of the dependability mechanisms such as automatic exception propagation, and more generally the behavior of the operations described in the B specification and that will be executed during runtime. Furthermore, it enables static analysis to be performed in order to verify the structural properties of a given system described in the invariant of the specification.

6. Conclusion

This paper has presented both how to specify dependability mechanisms using the B formal method and a development support relying on an XML-based language and on the refinement process of B.

We have considered the use of CA actions that have been proved useful for building dependable systems. We have defined a generic formal specification using the B method, defining systems composed of several CA actions that make concurrent accesses to external objects. B was chosen because of its powerful theorem proving ability and because of availability of a number of mature tools. We have shown how to specify the following dependability mechanisms of CA actions: (i) constraints related to the atomic access to external transactional objects, (ii) encapsulation of computations inside atomic action units ensured through action nesting and composition and (iii), properties related to the behaviour of the system in case of exception occurrences.

The XML-based language is to be used for describing a specific system instance such as a travel agency system, by giving the behavior of each participant. Refinement of the B specification is exploited for offering a correct implementation of the language. This includes static analysis and run-time support, whose correct implementation further depends on the one of third-party libraries.

Up to now several implementations of CA actions have been proposed and experimented with, but mainly on closed systems [9, 2]. We are working on an implementation of CA action-based systems to be defined as a composition of Web services such as the travel agency. This kind of systems clearly needs new dependability properties (e.g., relaxed atomicity properties for accessing external objects). We intend to use this initial B specification to study such properties.

Acknowledgments
This research was partially supported by the European IST DSoS (Dependable Systems of Systems) project (IST-1999-11585)[5].

References

[1] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[2] D. Beder, A. Romanovsky, B. Randell, C. Snow, and R. Stroud. An Application of Fault Tolerance Patterns and Coordinated Atomic Actions to a Problem in Railway Scheduling. *ACM, Operating Systems Review*, 34(4):21–31, October 2000.

[3] R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, SE-12(8), 1986.

[4] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[5] D. Schwier, F. von Henke, J. Xu, R. Stroud, A. Romanovsky, and B. Randell. Formalization of the CA Action Concept Based on Temporal Logic. Design for validation (deva) basic esprit project. second year report. part 2, LAAS, France, 1997.

[6] F. Tartanoglu, V. Issarny, N. Levy, and A. Romanovsky. Dependability in the Web Service Architecture. In *Proceedings of the ICSE Workshop on Architecting Dependable Systems*, Orlando, USA, May 2002.

[7] J. Vachon, N. Guelfi, and A. Romanovsky. Using COALA to Develop a Distributed Object-Based Application. In *In the 2nd Int. Symposium on Distributed Objects and applications (DAO'00), P. Drew, R. Meersman, Z. Tari, R. Zicari (Eds.)*, pages 195–208, Antverp, Belgium, 2000.

[8] J. Xu, B. Randell, A. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Proceedings of the Twenty-Fifth IEEE International Symposium on Fault-Tolerant Computing*, 1995.

[9] J. Xu, B. Randell, A. B. Romanovsky, R. J. Stroud, A. F. Zorzo, E. Canver, and F. W. von Henke. Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions. In *Symposium on Fault-Tolerant Computing*, pages 68–75, 1999.

[5] http://www.newcastle.research.ec.org/dsos/