Timed Probabilistic Reasoning on UML Specialization for Fault Tolerant Component Based Architectures

Jane Jayaputera, Iman Poernomo and Heinz Schmidt
{janej,ihp,hws}@csse.monash.edu.au
CSSE, Monash University, Australia

ABSTRACT

Architecture-based reasoning about reliability and fault tolerance is gaining increasing importance as component-based software architectures become more widespread. Architectural description languages (ADLs) are used to specify high-level views of software design. ADLs usually involve a static, structural view of a system together with a dynamic, state-transition-style semantics, facilitating specification and analysis of distributed and event-based systems. The aim is a compositional syntax and semantics: overall component behavior is understood in terms of subcomponent behavior. ADLs have been successful in understanding architecture functionality. However, it remains to be investigated how to equip an ADL with a compositional semantics for specification and analysis of extra-functional properties such as reliability and fault-tolerance.

This paper combines architecture definition with probabilistic finite state machines suitable to model reliability and fault-tolerance aspects. We present a compositional approach to specifying fault tolerance through parameterization of architectures. Using Probabilistic Real Time Computational Tree Logic (PCTL) we can specify and check statements about reliability of such architectures.

1. INTRODUCTION

In distributed systems, fault-tolerance can be aided by replication mechanisms. Central to these mechanisms is the notion of *fail-over*: a backup server takes over the job from a crashed server after a short timeout period and sends data back to the client directly, without human reconfiguration, as if the original server is still operating. There is a range of possible replication algorithms for achieving fail-over. We would like to systematically apply similar kinds of fail-over to design more reliable component-based software architectures. Given particular client and server components, we wish to create a fault tolerant architecture by associating a replication algorithm with calls to the server from the client.

Architectural description languages (ADLs) are used to specify high-level views of software design. ADLs usually involve a static, structural view of a system with a dynamic, state-transition style semantics, facilitating specification and analysis of distributed and event-based systems. The implementation is compositional: component behavior is understood in terms of subcomponent behavior.

However, the compositionality of component specifications cannot be taken for granted, especially when extra functional properties of the dynamic behavior are modeled [18]. In previous work [14] we have developed a compositional ADL-based approach to reliability using Markov chains.

In this paper, we extend some of those ideas, focusing on representing a range of fail-over replication strategies in the syntax and semantics of a compositional ADL. The novelty of our approach is the combined use of

- parameterization of architectures to treat replication strategies systematically, and
- a probabilistic semantics to facilitate fault-tolerance analysis of resulting architectures.

We can then use our work to reason about reliability properties of software component architectures. Our approach is simple, combining three formalisms:

1. We define an ADL with probabilistic finite state machine (PFSM) semantics. The semantics tells us about the dynamic behavior of a component. Specifically, it permits us to define how a call to a component interface method will result in calls to other required components. Our semantics is probabilistic, so it permits our models to relate usage profiles of method calls (the probability that particular sequences of methods will be called), and also to model method reliability (i.e., the probability of method execution success). Overall component reliability is then given as a cumulative function of method reliability over all component interfaces.
2. We apply a parameterization mechanism to add fault tolerant features automatically into the ADL. The parameterization involves choosing one of the provided replication algorithms. These algorithms replicate the server process to enable fail-over.
3. We use Probabilistic Real Time Computational Tree Logic (PCTL) to specify and check statements about such architectures. This is possible because PCTL statements have truth values that are determined according to the ADLs probabilistic finite state machine semantics, if we associate logical properties with particular states. To check statements against architectures, we use the compositional semantics to build a machine for the architecture, preserving the logical properties known to hold for subcomponents. This yields a larger PFSM over which PCTL statements may be checked.

2. FAULT TOLERANCE IN DISTRIBUTED SYSTEMS

Reliability and fault-tolerance are some of the key issues of current distributed systems. Large scale, widely distributed systems contain large numbers of network nodes and connections. There is a likelihood that some nodes or some connections will be unavailable. Because many connections and

intermediate nodes are needed to enable a client-server communication, this temporary unavailability can significantly decrease the overall reliability of a system. This is of particular concern in distributed enterprise systems.

Replication combined with a fail-over logic is a common fault tolerance mechanism, overcoming some of these problems and hence increasing overall availability and reliability.

Replication mechanisms try to overcome system failure by duplicating processes and resources. Then clients can access resources without having to worry about server crashes and unpredicted downtime. This is possible because requests made to a crashed server are now diverted to another server (or *replica*). The replica then sends information back to clients directly, as if the original server itself was performing the data transfer.

A range of fail-over replication algorithms has been proposed. *Passive replication* (also known as primary-backup) [2] and *active replication* (also known as state machine) [20] approaches are probably the most important and well-used ones. Most of the remaining approaches are extension of these two replication methods. There are several important extensions. *Active client in passive replication* approach, abbreviated here as *active client replication*) [3], extends the primary-backup approach by making a client actively choose a server to contact. Recently, [4, 17] specified a new replication technique known as *semi-passive replication* mechanism. For reasons of scope, we only describe passive replication and active client replication algorithms in this paper. Interested parties are referred to, for example, [7] for details on the other algorithms.

The passive replication approach mainly works as follows. At any one time, there is at most one primary server to serve requests from clients. Other servers act as backups. These backups receive the updated data from the primary server and do not interact directly with clients. If the primary server fails, one of the backups takes over the serving role and acts as the new primary server [2].

In order to overcome the main drawback of passive replication approach, active client replication was invented. It extended the passive replication algorithm by allowing client to contact a backup server directly if the primary server does not function correctly (crash) [3]. One of the current authors has extended the approach by allowing busy server to be handled as well as crashed primary server and message omission failure [7].

Our probabilistic fail-over model uses the following probabilities (illustrated in Fig. 1).

- $P_{primary}(S)$ is the probability of server S to be chosen as primary server when a client makes a request.
- $P_{busy}(S)$ is the average probability of server S being busy for a call at any time.
- $P_{bottleneck}(S)$ is the failure probability of server S.
- $P_{transfer}(S, S')$ gives the probability of server S' being chosen as backup server in lieu of server S.

Without loss of generality, for simplicity of the examples, we assume that these probabilities are independent of S.

3. ARCHITECTURAL DESCRIPTION LANGUAGE

Architectural Description Languages (ADLs) are used to specify coarse-grain components and their overall interconnection structure. ADLs are compositional, permitting the

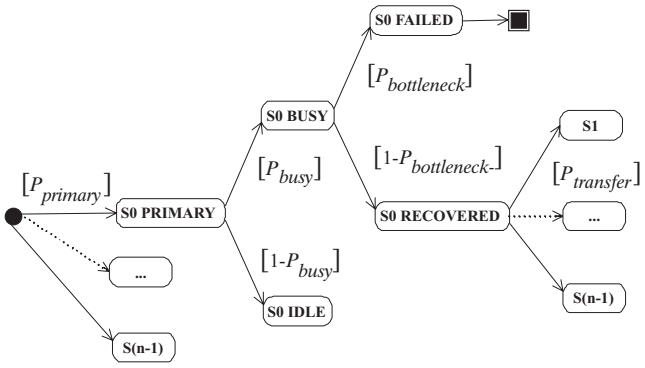

Figure 1: Probabilities.

specification of components in terms of smaller components. Examples of ADLs are Darwin [10], Wright [1] and radl [19].

In this section, we define a simplified version of radl, a language for describing and analyzing both functional and nonfunctional properties of architectures. Like many ADLs, radl consists of a visual and textual notation for defining the static composition of a system, and a state transition semantics for analysis of dynamic aspects. We describe the former and then the latter.

The basic elements of our language are components – referred to as *kens* in radl.[1] The functionality of a component is defined by a set of provided and required ports. The ports are referred to as *gates* in radl. Gates are to be regarded as interfaces of the component, which can be accessed by an external client. The internal functions of a ken are specified by internal ports, which can be thought of as internal methods used to implement an interface, not available to an external client. Provided and required gates express the external functionality that a ken provides and needs to use, respectively.

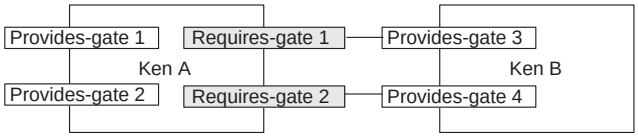

Figure 2: Example of radl.

4. FAULT TOLERANCE PARAMETERIZATION IN ADL

Replication is usually described as communications between a client with some servers in distributed systems. We propose a method to abstract over these communications in component-based software architecture. We encapsulate the detailed design of server replications using an abstraction concept, added into the ADL. Instead of expecting a software designer to know the details of a particular algorithm, we provide a mechanism to automatically produce a fault tolerant architecture, given client and server components and a chosen algorithm. The mechanism is called *Parameterized Replication*.

[1]We use the term *ken* to specify the elements of our ADL, as these elements are often more general than traditional system components, representing a range of other architectural building blocks, such as transactional boundaries or, in our case, parameterized fault-tolerant architectures.

We allow kens to be parameterized by different replication strategies, through use of the *ParRepl* construct. Like UML template classes (and template components), we draw a parameter as a box in the corner of a ken to illustrate the parameterized replication notation.

The first abstraction represents a black-box of parameterized replication. The parameterization shows the name of the replication algorithm that is used (see for example Fig. 3). In this abstraction, an architect does not have to know the algorithm details. The architect need only specify a name of replication algorithm that will be used (in this case AR stands for active replication).

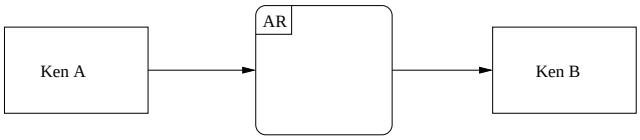

Figure 3: Abstraction 1.

In radl, the first abstraction corresponds to a new construct

$$\texttt{ParRepl}(C, S, algo, probs, n)$$

This element wraps all interactions between a client C and server S according to a replication algorithm defined by *algo*. The algorithm ranges over fail-over mechanisms, replicating n copies of the server S by n. We assume that $\{S_0, S_1, ..., S_{n-1}\}$ are a set of servers. We denote the algorithms mentioned above by `PassiveReplAlgo`, `ActiveReplAlgo`, `SemiPassiveReplAlgo`, `ActiveClientReplAlgo` with their obvious meaning. We use probabilities *probs* specified at the end of section 2 for reliability measurement of fault tolerant architecture.

Fig. 2 presents two basic kens $Ken\ A$ and $Ken\ B$ with two bindings between required and provided gates (shaded and white rectangles, respectively). $Ken\ A$ is a client of $Ken\ B$, a server. Fig. 4 shows a variation of Fig 2. $Ken\ B$ has been parameterized by the formal parameter of the replication algorithm. A particular replication algorithm, Passive Replication, has been chosen as an actual parameter.

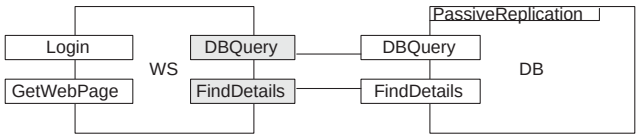

Figure 4: Example of syntax for fault tolerant radl.

The second abstraction is concerned with the actual communication of client and servers according to a particular algorithm. The active replication algorithm is shown in Fig. 5 (a), where a client communicates with all servers. Fig. 5 (b) shows passive replication algorithm, where a client can contact a primary server only. This abstraction is elaborated in section 5.

5. SEMANTICS FOR DYNAMIC ANALYSIS

Architectures of radl are equipped with probabilistic finite state machine (PFSM) semantics. Our semantics is compositional, in the following sense. Each basic ken is associated with a set of PFSMs, defining how calls to provided gates

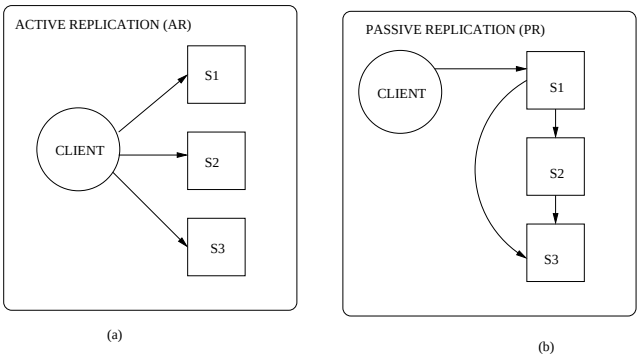

Figure 5: Abstraction 2.

result in internal gate calls and outgoing signals to required gates. Then, our semantics defines how larger compositions of kens result in larger PFSMs from the individual kens' PFSMs.

A PFSM may be formally defined:

Definition 5.1 (PFSM). *A Probabilistic Finite State Machine (PFSM) is a tuple*

$$D = \langle E_D, A_D, Z_D, \delta_D, Prob_D, initial_D, Final_D, failed_D \rangle$$

E_D is called event alphabet, A_D is the action alphabet, Z_D denotes the set of states. $initial_D, failed_D \in Z_D$ are designated initial and fail state, respectively and $Final_D \subset Z_D$ is the set of final states.

$$\delta_D : Z_D \times E_D \cup A_D \rightarrow Z_D$$

and

$$Prob_D : Z_D \times E_D \cup A_D \rightarrow [0, 1]$$

are the transition function and the transition probability, respectively, where for all $z \in Z_D$:

$$1 = \sum_{x \in E_D \cup A_D} Prob_D(z, x)$$

In other words $Prob_D$ induces a probability distribution on the set of transitions for each state.

For the sake of conformance to standards for interchangeability within CASE tools, our visual notation for PFSMs borrows from UML statechart diagrams and extends them with probabilities. Initial, failed and final states are marked using a solid circle, a solid rectangle and a circle marked with a dot, respectively. In our extension, a transition from state s_1 to s_2 is of the form:

$$s_1 \xrightarrow{e[p]/a} s_2$$

where

- e is an optional *event*. We use events to denote calls to a ken's provided or internal gates, resulting in the state transition
- a is an optional *action*. We use actions to denote internal activity or calls to a ken's required gates.
- p is a probability value (the probability of the given transition). ¿From a practical viewpoint, p is often the product $u \cdot r$ where u is the usage probability (of

the event e) and r the reliability of the action a. For details see [14].

A PFSM D then gives rise to a (finite-state discrete-time) Markov chain (transition probability matrix) M by summing the (multi-edge) probabilities between a given pair of states and solving the resulting model analytically [14].

Composing Fault Tolerant PFSM

Our ADL has been defined as a Meta Object Facility (MOF) meta-model through UML specialization [12]. The details are topic for another paper. Here it suffices to mention that the MOF/UML permits to treat architectural and behavioral considerations in tandem. Kens are treated as metaclasses that contain a meta-method *semantics*() which returns a PFSM model (as a first class object) accessible to other methods in our implementation.

We define a fault tolerant parameterization semantics as follows:

1. There is a UML meta-model for describing our ADL and its semantics, mainly defined by `StateMachine`, `Transition`, and `State` [11]. They are similar to `PFSM`, δ_D and Z_D in Def. 5.1, respectively.

2. The replication algorithms are given by a metaclass ReplAlgo, used to generically compute semantics for a given replication choice. This metaclass is equipped with a virtual (meta-)method *execAlgo* that denotes the algorithm. The range of replication algorithms may then be represented by subclassing ReplAlgo and redefining this function.

   Some algorithms inherit properties of ReplAlgo abstract class: *PassiveReplAlgo*, *ActiveReplAlgo*, *Semi-PassiveReplAlgo*, and *ActiveClientReplAlgo*. This approach is efficent from a meta-modeling perspective, because it enables us to treat all algorithms as of the metaclass type ReplAlgo.

3. A *ParRepl* is given as a parameterized (meta-)class that takes four arguments. Given client PFSM and server PFSM, it calls an abstract class called *ReplAlgo* and outputs a combined PFSM as the result.

Another novelty of this approach is that we use MOF and UML meta-modeling to generically treat architectures over particular replication algorithms.

We outline a compositional semantics for fault tolerant architectures, based upon PFSMs. We refer the interested reader to [13] for a detailed formal description of a similar semantics using deterministic finite state machines without probabilities and [8] with probabilities. We define passive replication algorithm in this paper to illustrate composition of fault tolerant architectures.

Binding. Given ken C with required gate pp connected to ken S with provided gate pp,

$$Bind(C, S, pp)$$

we build a larger set of PFSMs associated with the provided gates of $Bind(C, S, pp)$

$$[\![Bind(C, S, pp)]\!]$$

according to the following algorithm. First, let

$$Sem_0 = [\![S]\!] - \{PFSM_{pp}\}$$

1. Set $i := 0$.

2. For each

   $$PFSM_{epp} \in [\![C]\!]$$

3. If $PFSM_{epp}$ does not involve pp as an action, then let $Sem_{i+1} = Sem_i \cup \{PFSM_{epp}\}$, let $i := i + 1$, take the next $PFSM_{epp} \in [\![C]\!]$ and go to step 3.

   Otherwise, go to step 4.

4. Take every transition in $PFSM_{epp}$ that involves pp as an action,

   $$s_1 \overset{e[call*rel]/pp}{\longrightarrow} s_2 \quad (1)$$

   and let $RPFSM_{pp}$ be the $PFSM$ for the provided gate pp in $[\![S]\!]$. We define $Change(RPFSM_{pp}, s_2)$ to be $RPFSM_{pp}$ with its *initial* state replaced by $s_2.pp.start$ and $final$ state with $s_2.pp.end$. Then, we insert $Change(RPFSM_{pp}, s_2)$ between s_1 and s_2, in the sense that we delete the transition 1 and replace it with

   $$s_1 \overset{e[call*rel]/pp}{\longrightarrow} s_2.pp.start$$

   and add the transition

   $$s_2.pp.end \overset{pp.end[1]}{\longrightarrow} s_2$$

   We call the resulting machine $PFSM'_{epp}$.

5. We let $Sem_{i+1} = Sem_i \cup \{PFSM'_{epp}\}$, take the next $PFSM_{epp} \in [\![C]\!]$, set $i := i + 1$ and go to step 3.

ParRepl. We define the semantics of the fault-tolerant replication

$$[\![ParRepl(C, S, algo, probs, numServers)]\!]$$

according to the algorithm defined in *algo*. Here we only give a sketch of the semantics. The probabilities $primary(S)$ are added in the transition between client PFSM to the PFSM for each servers S, by defining a new state $S.current$ which intercedes calls from the client with $primary(S)$ as the probability of transition from the client call. Probability $busy(S)$ is added in the transition after $S.current$, by defining transition $busy(S)$ to a new state $S.BUSY$. $idle(S)$ is the probability of server S being idle (does not fail). The probability $bottleneck(S)$ is added in the transition between $S.BUSY$ to the root PFSM. Probabilities $transfer(S, S')$ are added in the transition between $S1.transferControl$ to other servers' PFSM starting state $S'.current$. Probability $otherFailures(S)$ is being added in the transition between $S.BUSY$ to a new state $S.transferControl$.

ReplAlgo. As an illustration, we define the semantics for $ParRepl$ where *algo* is set to `PassiveReplAlgo`.

1. Set n := 0 and do all steps in $Bind(C, S, pp)$ up to step 3. Instead of calling step 4, refer the call to this algorithm in step 2.

   Let us add some probabilistic transitions to each of ken S's $PFSM$s in steps 3 for correct processes and 4 for busy processes.

2. For each $PFSM_n$
   s.t. $n < numServers$ and $PFSM_{pp} \in [\![S]\!]$:

3. Go to step 4 in $Bind(C, S, pp)$ to compose a larger PFSM in normal case (no failure). After that, we do a composition for failure cases. We define a new function $Add(FTPFSM_{pp}, s_1, s_2)$ to add a new state, namely $s_2.pp.current$, so that we can put a probability in choosing a primary server $probs[n][primary]$ for all $numServers$ servers. Then, we insert the function $Add(FTPFSM_{pp}, s_1, s_2)$ so that two more transitions are added between s_1 and s_2 (besides the transitions in step 4 of $Bind(C, S, pp)$) with

$$s_1 \overset{probs[n][primary]}{\longrightarrow} s_2.pp.current$$

and

$$s_2.pp.current \overset{probs[n][idle]}{\longrightarrow} s_2.pp.start$$

4. Add a transition to a new state $S_n.BUSY$ if busy processes happen:

$$s_2.pp.current \overset{probs[n][busy]}{\longrightarrow} S_n.BUSY$$

In the case of bottleneck (crashes) without having a chance to transfer the control to a new primary:

$$S_n.BUSY \overset{probs[n][bottleneck]}{\longrightarrow} final$$

Add a transition in the case of failures other than bottleneck:

$$S_n.BUSY \overset{probs[n][other]}{\longrightarrow} S_n.transferControl$$

And add other transitions to the starting state of other backup servers:
For $((0 \leq backup < numServers) \wedge (backup \neq n))$:

$$S_n.transferControl \overset{probs[n][transfer][backup]}{\longrightarrow} s_2.pp.current$$

Then set $n := n + 1$ and repeat step 3 until $n = numServers$.

5. We call the resulting machine $PFSM'_{epp}$ and go to step 5 in $Bind(C, S, pp)$.

6. PROBABILISTIC COMPUTATIONAL TREE LOGIC

Probabilistic Computational Tree Logic (PCTL) (see for instance [6]) is used to specify and check timing and probabilistic properties on our architectures. We define PCTL in terms of *structures* comprising PFSMs and associating additional atomic propositions with states:

DEFINITION 6.1 (STRUCTURE). *A* structure *is a tuple*

$$S = \langle D_S, Props_S, h_S \rangle$$

where D_S is a PFSM, $Props_S$ is a finite set of atomic propositions and $h : Z_{D_S} \rightarrow \mathcal{P}(Props_S)$ is a function decorating states with propositions sets.

The idea is to extend each basic ken PFSM by associating atomic propositions with states. We use the architectural semantics of the previous section, so that compositions of kens have appropriately expanded structures. Formulae can then be specified about the provided gates of any given architectural composition, and then checked against the associated structures.

We extend the traditional definition of PCTL which uses transition probabilities without associated events or actions. To this end we use finite sequences in $(E \cup A)^*$, infinite sequence in $(E \cup A)^\omega$ and bounded sequences in $(E \cup A)^{n<k}$ for some bound k. Such a sequence $a_0 \ldots a_{n-1}$ is also called a finite, infinite or bounded *trace*, respectively, of the PFSM D, if there is an associated sequence of states $s_0, \ldots, s_{n-1} \in Z_D$ such that $initial_D = s_0 \overset{a_0}{\rightarrow} s_1 \overset{a_1}{\rightarrow} \cdots s_{n-1}$ are legitimate transitions, i.e., if $\delta_D(s_i, a_i) = s_{i+1}$ for all for the respective step i. The probability of this trace is the product of the single transition probabilities: $p(a_0 \ldots a_{n-1}) = \prod_i Prob_D(s_i, a_i)$. The trace is also called *accepted* if $s_{n-1} \in Final_D$.

As computation tree paths in PCTL are sequences of states without transition symbols, in order to forget the transition symbols we map a given trace $t = a_0 \ldots a_{n-1}$ to its underlying state sequence $s = s_0 \ldots s_{n-1}$. We denote by s_t the underlying state sequence of t. Since different symbols can make different transitions between the same pair of states, then different traces can have the same underlying state sequence. By $T_s = \{t \mid s_t = s\}$ we denote the set of traces with the same underlying state sequence s.

Now the paths in the sense of traditional PCTL are the underlying state sequences of the traces in T_D. Their probability can be computed as

$$p(s) = \sum_{t \in T_s} p(t)$$

This means we sum the trace probabilities given in the PFSM over all traces with the same underlying state sequence to derive the probability of the state sequence.

PCTL now permits model checking with temporal formulae that are composed from atomic propositions and include modal operators with optional lower reliability bounds and upper time bounds. Regarding the time bounds, such operators are interpreted over all traces (and state sequences) up to the given length in a time bound. Bounded reliability is defined using the probabilities above, where we sum the probabilities over all traces (implicitly over all underlying state sequences) satisfying the given formulae, i.e. over a corresponding state 'computation tree'.

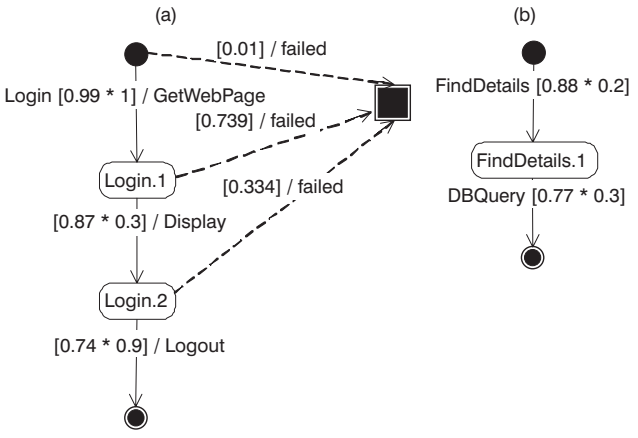

Figure 6: PFSMs associated with basic kens of our example: (a) $PFSM_{\texttt{Login}} \in [\![\texttt{WS}]\!]$, (b) $PFSM_{\texttt{FindDetails}} \in [\![\texttt{DB}]\!]$, and The failed state and its incoming transitions are displayed in chart (a) and are left implicit in the other chart.

Without loss of generality and for the sake of simplicity in the following examples, we now assume that the above probabilities are all independent of the chosen server(s). In

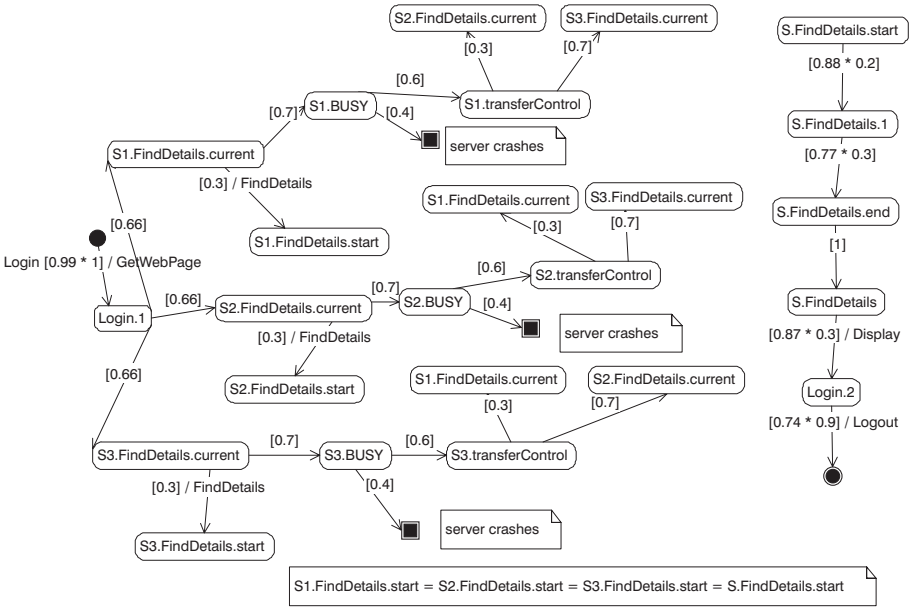

Figure 7: Fault tolerant PFSM using passive replication.

other words, we assume a homogeneous pool of servers with symmetric fail-over policy.

EXAMPLE 6.1. *Using* `PassiveReplAlgo`*, let* $NewDBReq$ *be an atomic proposition, standing for the fact that a new database connection that fault tolerant needs to be added to the pool in our example. Assume that* $h_{\mathtt{DB}}(FindDetails.1) = h_{\mathtt{WS}}(Login.2) = \{NewDBReq\}$*, so that*

$$h_{\mathtt{ARCH}}(S1.FindDetails.1) = h_{\mathtt{ARCH}}(C.Login.2) = \{NewDBReq\}$$

We make the following specification about the behavior of a call to the `Login` *gate of* `ARCH`*. In the best case, there is a probability of at least* 0.2% *that we will require a new database connection in less than or equal to* 7 *times steps in server* S_1*.*

$$True\ U^{t\leq 7}_{p\geq 0.002}\ NewDBReq \tag{2}$$

After the third attempt, we get a probability of at least 0.37% *that we will require a new database connection in less than or equal to* 14 *times steps in server* S_3*, using the same formulae above.*

For the overall architecture, we get a reliability of at least 82% *if we use one server only. The reliability is increased by* 16.3% *if we use three servers instead.*

$$True\ U^{t\leq 8}_{p\geq 0.82}\ final \tag{3}$$

Using the algorithms of [6], we can verify that this specification is true of `ARCH`*.*

EXAMPLE 6.2. *Assuming active client replication algorithm is used, we use the same atomic proposition as in Example 6.1. For the best case, we have probability of at least* 0.2% *that we will require a new database connection in less than* 7 *times steps in server* S_1 *using Equation (2). After the third attempt, we get a probability of at least* 0.38% *that we will require a new database connection in less than or equal to* 17 *times steps in server* S_3*, using the same formulae.*

For the overall architecture, at least 60% *reliability is achieved if we use one server only. By replicating the number of servers to three, we get a greater reliability of at least* 98.2%*. We use the formulae in Equation (3) for both measurements.*

The PFSM in Fig. 7 is the result of combining individual PFSMs in Fig. 6, using the composition algorithm for passive replication defined in Section 5.

In relation to composing PFSMs using the active client approach in Example 6.2, more states need to be used compared to Example 6.1. This is due to the need to choose a new primary server in case the old primary fails. Thus, the client has to resend a request to the new primary and the corresponding client PFSM has to be included again in the composition. As the number of states increases, the number of steps also increases (21 steps in Example 6.1 compared to 30 in Example 6.2).

Updating backup servers (as part of the passive replication algorithm) is not included in the PFSM composition. We can avoid this because our reliability measurements are only the sequences leading to the client receiving a response regardless of server replication. It can be seen from Example 6.1 and Example 6.2 that reliability of architecture is increased by around 0.01% if we use three servers instead of one server only. Since a crashed server can often be restarted after a failure, there is an additional element of availability 'built-in' that we are not even accounting for, since our simplified model treats failure as terminal.

7. IMPLEMENTATION

The compositional semantics for our version of radl have been implemented. The software, called *FSMComb* [5], implements our methods in Java. FSMComb interfaces to the PRISM [9] model checking tool which permits to check PCTL specifications against our architectures. Generally, FSMComb combines two or more PFSMs for individual kens into one for the given architecture. The PFSMs are read from text files and then extended by fault-tolerance constructs as described. Finally the composite models are exported to the PRISM checker together with fault-tolerance assertions. PRISM then verifies these models and reports the results.

8. RELATED WORK AND CONCLUSIONS

Little work has been done in using ADLs to specify and analyze fault tolerant properties.

In [16], non-functional properties such as dependability, safety, reliability and availability are defined formally using a predicate logic with some extensions. Then, components and replication methods are also defined formally using the non-functional properties that have formally been defined previously. However, that work does not relate a compositional ADL-style view of architecture to replication techniques. The relation of fault tolerance to executable architectures was investigated in [15]. The approach adds fault tolerant supports into SOFA component framework. The SOFA framework is based on component oriented programming. Thus, SOFA is similar to OMG's CORBA, Sun's Enterprise Java Beans and Microsoft's COM. Although they use replication methods such as active and passive replication as our approach, there are some differences.

The approach does not replicate a component automatically. Also, the primary goal of that work is to implement fail-over algorithms directly in SOFA source code, without describing it abstractly in architecture.

In [14], we developed a compositional approach to reliability models where PFSMs are associated to hierarchical component definitions and to connectors. Markov chain semantics permits hierarchical composition of these reliability models. However the paper does not develop a fault-tolerance model. The work presented here is in part based on that semantics.

This paper presented a compositional approach to fault-tolerant component-based architectures. We modeled fail-over mechanisms in a pool of replicated servers. PFSMs were associated with components and connectors to define the behavior of hierarchical component-based architectures. We sketched a formal semantics using PCTL.

Parameterized architectural patterns are used in our approach, in which the chosen fault-tolerance mechanisms becomes a parameter. The actualization of the parameter includes probabilities for weaving the PCTL fault-tolerance model into the PCTL models of the client-server interface functionality.

Finally, the paper developed an example for the special case of a pool of symmetric servers - although our approach is more general. The example also illustrated our prototype implementation of the PCTL architecture weaver.

9. REFERENCES

[1] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, May 1997.

[2] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed Systems*, chapter 8, pages 199–215. ACM Press, 2nd edition, 1993.

[3] P. Chundi. *Protocols for Achieving Consistency and Reliability in Replicated Database Systems that Utilize Asynchronous Updates*. PhD thesis, Department of Computer Science, University of Albany - State University of New York, August 1996.

[4] X. Défago and A. Schiper. Specification of replication techniques, semi-passive replication and lazy consensus. Technical Report IC/2002/007, École Polytechnique Fédérale de Lausanne, Switzerland, Feb. 2002.

[5] Fsmcomb - finite state machine combinator. See `http://www.csse.monash.edu.au/~janej`.

[6] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[7] J. Jayaputera. *Fault Tolerance in Active Client/Passive Replication: Tolerating Faulty or Slow Servers and Handling Network Partitions*. Honours Thesis, School of CSIT, RMIT University, Oct 2001.

[8] J. Jayaputera, I. Poernomo, R. Reussner, and H. Schmidt. Timed probabilistic reasoning on component based architectures. In H. Sondergaad, editor, *Third Australian Workshop on Computational Logic*. ANU, Canberra, Dec 2002.

[9] M. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker. In *PAPM/PROBMIV'01 Tools Session*, 2001.

[10] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf.*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[11] OMG. *UML Specification v1.5*, March 2003.

[12] OMG. *UML Superstructure v2.0*, April 2003.

[13] R. Reussner, I. Poernomo, and H. Schmidt. Using the TrustME Tool Suite for Automatic Component Protocol Adaptation. In P. Sloot, J. Dongarra, and C. Tan, editors, *Computational Science, ICCS 2002, The Netherlands, 2002*, volume 2330 of *LNCS*, pages 854–862. Springer-Verlag, Berlin, Germany, Apr. 2002.

[14] R. Reussner, H. Schmidt, and I. Poernomo. Reliability prediction for component-based software architectures. *Journal of Systems and Software – Special Issue of Software Architecture - Engineering Quality Attributes*, 66(3):241–252, 2003.

[15] J. Rovner. Fault tolerant support for sofa. Technical report, Dept. of CSE, University of West Bohemia in Pilsen, Czech Republic, 2001.

[16] T. Saridakis and V. Issarny. Developing dependable systems using software architecture. In *Proceedings of the 1st Working IFIP Conference on Software Architecture*, pages 83–104, San Antonio, TX, USA, Feb 1999.

[17] A. Schiper. Failure detection vs. group membership in fault-tolerant distributed systems: Hidden trade-offs. In *PAPM-PROBMIV 2002*, LNCS 2399, pages 1–15, Denmark, July 2002. Springer Verlag. Invited talk.

[18] H. Schmidt. Trustworthy components – compositionality and prediction. *Journal of Systems and Software – Special Issue of Software Architecture - Engineering Quality Attributes*, 65(3):215–225, 2003.

[19] H. Schmidt, I. Poernomo, and R. Reussner. Trust-By-Contract: Modelling, Analysing and Predicting Behaviour in Software Architectures. In *Journal of Integrated Design and Process Science*, volume 4(3), pages 25–51, 2001.

[20] F. B. Schneider. Replication management using the state-machine approach. In S. Mullender, editor, *Distributed Systems*, chapter 7, pages 169–197. ACM Press, 2nd edition, 1993.