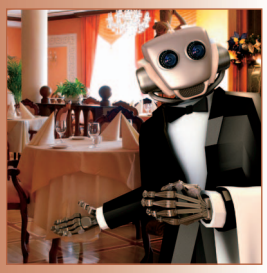

QoS Assessment via Stochastic Analysis

Using a stochastic modeling approach based on the Unified Modeling Language and enriched with annotations that conform to the UML profile for schedulability, performance, and time, the authors propose a method for assessing quality of service (QoS) in fault-tolerant (FT) distributed systems. From the UML system specification, they produce a generalized stochastic Petri net (GSPN) performance model for assessing an FT application's QoS via stochastic analysis. The ArgoSPE tool provides support for the proposed technique, helping to automatically produce the GSPN model.

Simona Bernardi
Università di Torino, Italy

José Merseguer
Universidad de Zaragoza, Spain

Critical software systems distributed over the Internet need high dependability to avoid unacceptable failures, such as crashes or attacks, despite the presence of faults. One way to guarantee dependability is via fault tolerance (FT). Software FT, in particular, involves embedding — within either the middleware or application layers — techniques aimed at system failure avoidance,[1] such as error-detection and recovery mechanisms coordinated according to an FT strategy.

Providing software systems with FT capabilities has its costs, however, and a trade-off is usually required between system overhead and an FT strategy's effectiveness, which is directly related to system dependability. Thus, FT distributed systems' quality of service (QoS) depends on basic metrics related to the adopted strategy's effectiveness (the time required for error-detection and recovery, for example) as well as on classical performance metrics (such as response time, throughput, or the probability of dropping packets due to network congestion) that let us quantify computation or communication overhead. We can assess such systems' QoS before development using fault-forecasting techniques,[1] which let us predict system behavior in the presence of faults. In particular, stochastic modeling and analysis techniques — in which we can represent system activity durations and delays as random variables and model logic conditions with probabilities — let us derive probabilistic estimates of QoS metrics.

To this end, we propose a stochastic modeling technique built on the Unified Modeling Language[2] and enriched with annotations conforming to the UML profile for schedulability, performance, and

 Published by the IEEE Computer Society 1089-7801/06/$20.00 © 2006 IEEE

Related Work in UML Specification for QoS Assessment

Andrea Bondavalli and colleagues presented one of the first attempts at enriching a UML design to specify dependability aspects for QoS assessment with fault-tolerant (FT) systems.[1] Their UML extension mechanisms don't conform to any OMG profile, but the authors exploit them to automatically produce stochastic Petri net (SPN) models amenable to dependability analysis. In particular, the authors derive high-level SPN models from UML structural diagrams (such as class, deployment, and use cases) under the assumption of standard behaviors by system components; they obtain more detailed SPN models from UML behavioral specifications (such as statecharts and sequence diagrams).

Vittorio Cortellessa and colleagues devised a methodology for estimating the performance risk factor — that is, the probability that a system won't meet certain performance requirements.[2] Although not specifically developed for the FT domain, this approach is based on using UML sequence diagrams to represent system scenarios and deployment diagrams to represent hardware restrictions. From the UML system specification — properly annotated with risk-related attributes according to the schedulability, performance, and time (SPT) profile — the authors derive a formal model (the execution graph) from which they can estimate the probability of performance failures and identify possible bottleneck components by computing asymptotic bounds.

References

1. A. Bondavalli et al., "Dependability Analysis in the Early Phases of UML-based System Design," *Int'l J. Computer Systems Science & Eng.*, vol. 16, no. 5, 2001, pp. 265–275.
2. V. Cortellessa et al., "Model-Based Performance Risk Analysis," *IEEE Trans. Software Eng.*, vol. 31, no. 1, 2005, pp. 3–20.

time (SPT).[3] We then assess QoS by analyzing a generalized stochastic Petri net (GSPN)[4] performance model, automatically derived from the UML-SPT specification. (For related work in combining UML and performance models for QoS assessment, see the sidebar.)

We've successfully applied our approach to assessing QoS in mobile-agent software systems,[5] and now adopt it for assessing QoS in FT Internet-based distributed systems. We take our modeling and analysis guidelines from our experience with verification and validation activities in the European Dependability for Embedded Automation Systems in Dynamic Environments project (www.depaude.org), and illustrate them via the Backbone case study (developed within Depaude). These guidelines are useful with a wide range of FT systems, which share important commonalities. Here, we present only a simplified version of the original UML models and the QoS assessment carried out in Depaude, focusing instead on the most relevant QoS aspects of Backbone.

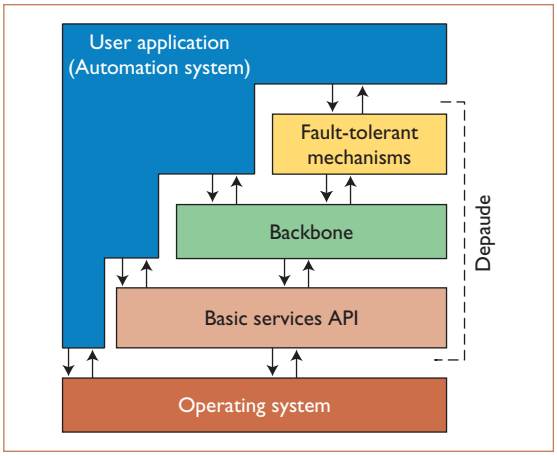

Figure 1. Processing-node architecture in Depaude. The Backbone element within the Depaude middleware is a distributed application that gathers and maintains error-detection information and coordinates fault-tolerant mechanisms.

The Backbone Case Study

The Backbone application is part of an FT middleware devoted to guaranteeing high-dependability for automation systems distributed over both intra- and internetworks (Ethernets and the Internet, respectively). Figure 1 shows the architecture of a processing node from an automation system integrated with the Depaude framework.

Backbone itself is a distributed application in which agents run on multiple processing nodes. Its main functionalities are gathering and maintaining error-detection information in replicated databases and coordinating the FT mechanisms at runtime by interpreting the user-defined FT strategy. Given that Backbone is the Depaude architecture's core element, its developers devised a self-check algorithm specifically so it could tolerate crash failures and temporary partitioning.

The Backbone agents — each of which runs on a different node — can act as either *masters* or *backups*. At startup, and in normal situations, the full system has only one master. In faulty situations, however, more than one master can exist —

that is, a temporary partitioning occurs in which the set of agents is divided into as many subsets as the number of current masters. The master is the only component that can initiate error recovery.

The agents execute the self-check algorithm with support from an alarm manager (AM). The algorithm involves a message exchange among participants: each backup agent expects a "master is alive" (MIA) message from the master every T_{t2r} seconds, at most, whereas the master expects a "backup is alive" (BIA) message from each backup every T_{t2r} seconds, at most. Each agent sends its "I am alive" message as soon as it receives the proper time-out expiration notification, which the AM running on that particular node sends every T_{t2s} seconds.

In a normal situation, the master and backups continue to receive and send MIA and BIA messages; however, if an agent doesn't receive such a message within the specified deadline, it detects a *late-timing failure* and views the sender agent as "failed." At this point, a transitory phase begins in which Backbone isolates the failed agent and, if this agent was a master, elects a new one from among the backups.

From the user's viewpoint, the main QoS requirement is to guarantee high Backbone availability without congesting the network.

Vincenzo De Florio and his colleagues implemented a heavier (that is, more complex) version of the self-check algorithm on a Backbone prototype running on a Parsytec Xplorer — a multiple-instruction, multiple-data engine — using four PowerPC nodes.[6] They tested this system via fault-injection and determined that it could tolerate crash failures and system partitioning. We analyze the lighter version of the Backbone self-check algorithm developed in Depaude, which is scalable with respect to the number of processing nodes.

As with the Depaude framework's other FT mechanisms, users must configure Backbone when defining the FT strategy; in particular, they must set the alarm durations T_{t2s} and T_{t2r}. From the user's viewpoint, the main QoS requirement is thus to establish reasonable values for these durations that guarantee high Backbone availability (99.99 percent) without congesting the network (the maximum acceptable overhead being 5 percent). In particular, users should satisfy the inequality $T_{t2s} \leq T_{t2r}$ to minimize the frequency of false alarms (such as when the master agent suspects a failed backup but the latter is actually working correctly).

So, how do we quantify Backbone's QoS? Following Daniel Menascé's proposal,[7] we can express it as a function of two metrics: the time required to detect a failed agent and isolate it (T^{td}), which is directly related to Backbone's availability, and the communication overhead (O^{net}) generated from executing the self-check algorithm. We can then define Backbone's QoS as

$$QoS = \frac{1}{2}\left(\frac{T^{td}_{\max} - T^{td}}{T^{td}_{\max}} + \frac{O^{net}_{\max} - O^{net}}{O^{net}_{\max}}\right), \quad (1)$$

where T^{td}_{max} and O^{net}_{max} are QoS user requirements. Once they're established, Equation 1 becomes a function of the system's input parameters, which characterize both the network (such as network speed) and the Backbone application itself (the number of execution nodes, size of exchanged messages, or activity and alarm durations, for example). We can then use the QoS formula during stochastic analysis to find and assign values to the alarm durations for a given system characterization that will satisfy the QoS user requirements.

UML Specification of FT Systems

UML is currently the primary choice for software engineers developing design system models. Following this current trend, we designed our FT applications using the UML statechart and sequence diagrams to model system behavior and the deployment diagram to model system architecture. Moreover, we move a step forward in the use of UML by proposing a technique that lets us the exploit these diagrams in verifying QoS system properties — that is, for system analysis purposes.

UML-SC

Most FT distributed applications, such as Backbone, exhibit discrete, event-driven behavior. To accurately model this behavior, UML provides the *statechart* formalism (UML-SC) as a primary notation. UML-SCs are, in fact, suitable for modeling system components that react to stimuli generated inside or outside the system boundaries. Such components carry out their system responsibilities

(services) by exchanging messages — a behavior the Backbone agents clearly exhibit.

Moreover, UML-SCs are useful for modeling specific FT behaviors such as affected system components' anomalous states, system-recovery actions, and different types of fault behavior (permanent, transient, or intermittent).

When modeling an event-driven FT system, we must identify those system components that cooperate to both carry out actual system functionalities (their "normal" behavior) and implement the inherent FT techniques (error detection and recovery). We can easily model each cooperating component's behavior in UML with a UML-SC; in our case study, these components were the backup and master agents and the AM. Using the master agent's UML-SC as an example, Figure 2 illustrates four guidelines (M1 through M4) common to modeling FT components:

- *M1*. Each component-provided service executes when agents properly react to the event (external stimuli) that requests that service. The occurrence of a `notify` event, for example — invoked by either the user application or an FT mechanism from the Depaude framework — moves the master to the `update&recovery` state, from which it performs the service (updating the database and starting user-application recovery). Once it finishes these activities, the master returns to its `waiting` state.
- *M2*. We can model self-checking between FT components using the UML-SC's message-exchange mechanism. The notation supporting message exchange labels UML-SC transitions with *trigger–effect* clauses. For example, when the master receives a time-out event `T2SM` (time to send MIA) from the AM, it notifies the backup of an MIA event. The transition `T2SM/Bck.MIA` represents the master accepting the `T2SM` event (the trigger) and the MIA event generation (the effect). The backup agent will manage MIA events to realize the master's healthy state (that is, that the master is properly working). Consequently, we model each agent as a self-checking component defining an *error-confinement area*, where failures in the system can occur.
- *M3*. The system must discover anomalous states, such as failure states, internally in order to execute proper actions. For the master agent, we model the failure state (`backup failed`) that the master agent reaches when it processes the AM-generated `T2RB` (time to receive BIA) event and realizes that the backup has failed. Because backup and master behaviors are similar, the agents and the AM concurrently implement error-detection policies. The AM's UML-SC in Figure 3 depicts how the system concurrently manages the alarms. We chose alarms as the mechanism to test system components' availability. Each alarm begins a count down and, when this count down expires, fires the alarm event to the proper agent. In the first concurrent region, for example, the AM sends the master a `T2SM` event, then waits for the master to receive a `reset_T2SM`.
- *M4*. Once the system has detected the anomalous states, it must take the proper recovery actions by implementing the FT strategy. We model this situation with an activity in the failure states (`backup failed` and `master failed`) that should initiate a chosen recovery policy, such as *reinitializing* the failed agent — that is, restarting this agent's computation.

We can easily model each cooperating component's behavior in UML with a UML state chart.

Observe that the representation of the FT strategy could be more or less complex than M4 indicates, depending on the model-abstraction level. In some cases, we must explicitly model the FT strategy mechanisms' behaviors via UML-SC. Moreover, depending on the objective of the analysis, a fault-generator model might be necessary (when assessing system availability, for example). UML-SC is suitable for modeling different types of fault behavior with respect to persistence (permanent, transient, or intermittent) as well as fault occurrences' effects on the system components.

UML-SD

The UML *sequence diagram* (UML-SD) provides the proper notation for representing scenarios of interest. Figure 4a (p. 38) shows the execution of a normal Backbone scenario, which includes the concurrent interactions representing the mutual

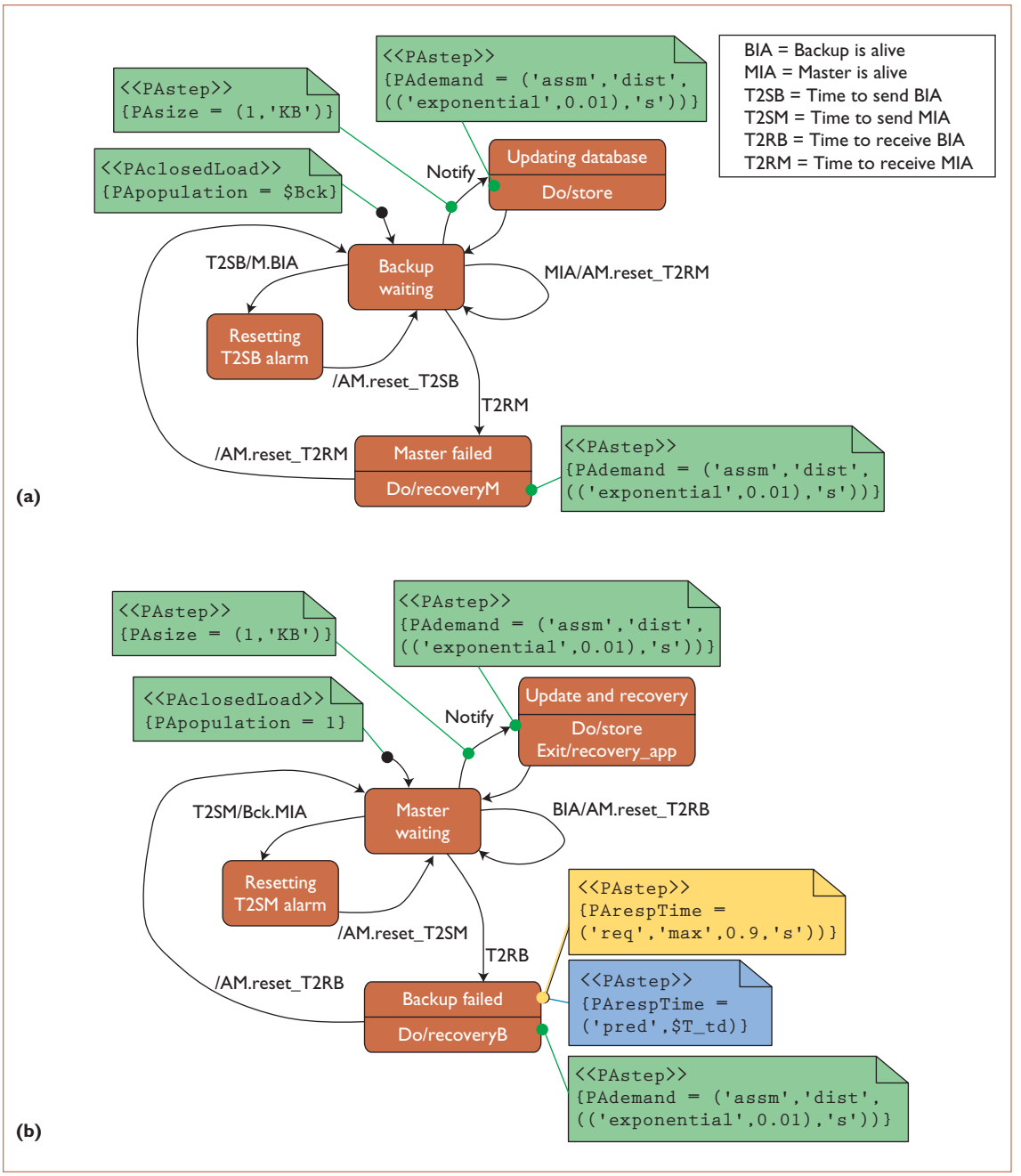

Figure 2. UML statechart specification. (a) The backup component is a software replica of a master agent that takes over the master's role if the latter isn't working correctly. (b) The master component is the primary agent for the Backbone application's self-check algorithm. The schedulability, performance, and time (SPT) annotations support the specification of different performance measures: required values (yellow), assumed values (green), estimated values (blue), and measured values (not used).

checking of the Backbone components and the main Backbone functionalities. In the same figure, the self-check scenario emphasizes the order among the exchanged messages. In the FT system domain, UML-SDs are useful for representing not only normal system executions but also faulty scenarios – that is, sequences of exchanged messages that lead to system failures or accomplish the coordinated recovery actions defined by the FT strategy. We could easily model a backup crash and its recovery, for example, by ordering `T2RB` and `reset_T2RB` events.

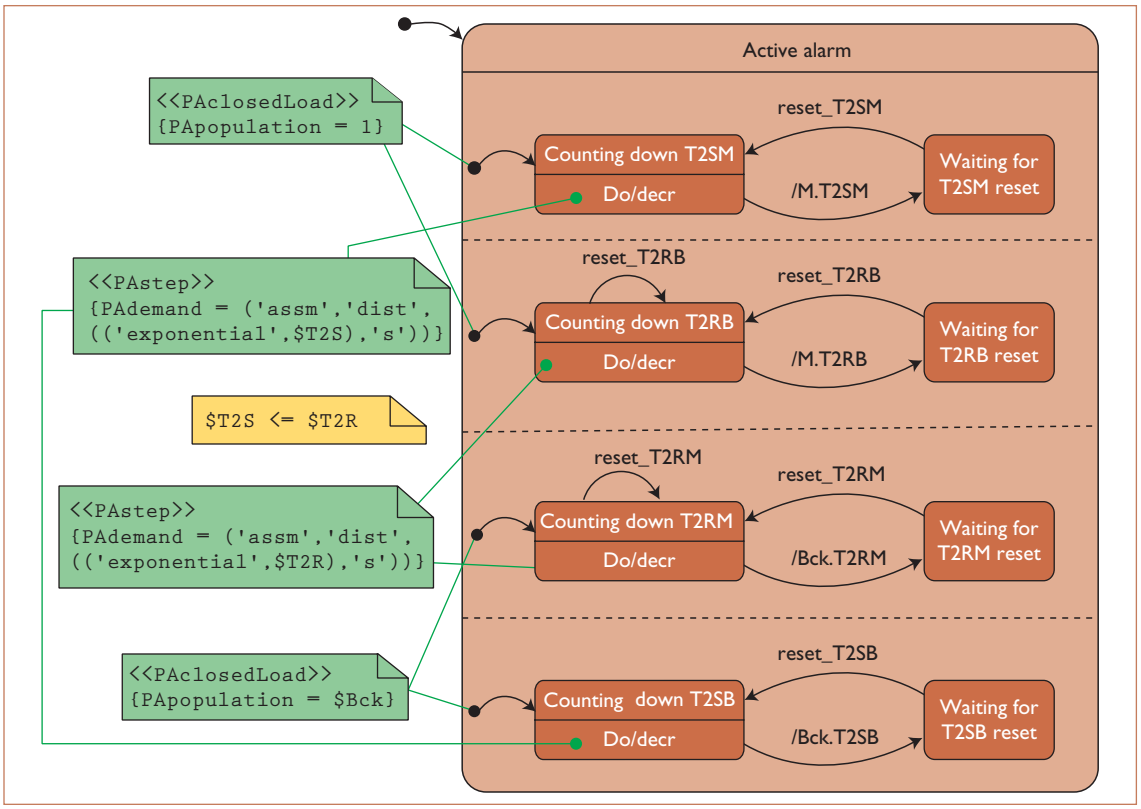

Figure 3. UML statechart (SC) specification for the alarm manager. The SC's concurrent regions are useful for modeling how the system can concurrently manage multiple alarms, including time-to-send (T2S) and time-to-receive (T2R) master and backup alarms. The schedulability, performance, and time (SPT) annotations help set the alarms' durations.

UML-DD

UML *deployment diagrams* (UML-DD) let us represent system architectures — particularly hardware and software redundancies, which are among the most-applied FT techniques (see Figure 4b). Backbone itself has been implemented as a redundant application consisting of software replicas — that is, the master and backup agents, which run on separate nodes, together with their associated AMs. Each backup is a replica of its master, and running on different nodes guarantees that when one (master or backup) is down the other is alive and can continue computation.

Enriching UML Models with Stochastic Information

The standard procedure for stochastic modeling and analysis follows four main steps:

- construct the stochastic model,
- establish the QoS metrics of interest,
- set the model parameters, and
- derive probabilistic estimates of QoS metrics using numerical, analytical, or simulation techniques.

We can repeat the latter two steps — for example, to study how performance changes if parameter values change (sensitivity analysis) or find the parameter configuration that guarantees best performance (optimization analysis).

The Depaude modeling experience, partially reported here, shows that UML-SCs, UML-SDs, and UML-DDs have the artifacts necessary for modeling most FT systems' intricacies. Thus, we can consider UML a very acceptable tool for modeling FT systems. However, UML lacks support for stochastic modeling, which enables fault forecasting and, consequently, QoS assessment via predictive quantitative analysis.

The UML-SPT profile provides a framework within UML that introduces new modeling capabilities as annotations. These SPT annotations *stereotype* core UML elements — that is, they add semantics in the time domain. Each stereotype has a list of tagged values describing time-related attributes. Figure 2 depicts several examples attached to UML notes. SPT annotations support the specification of different performance mea-

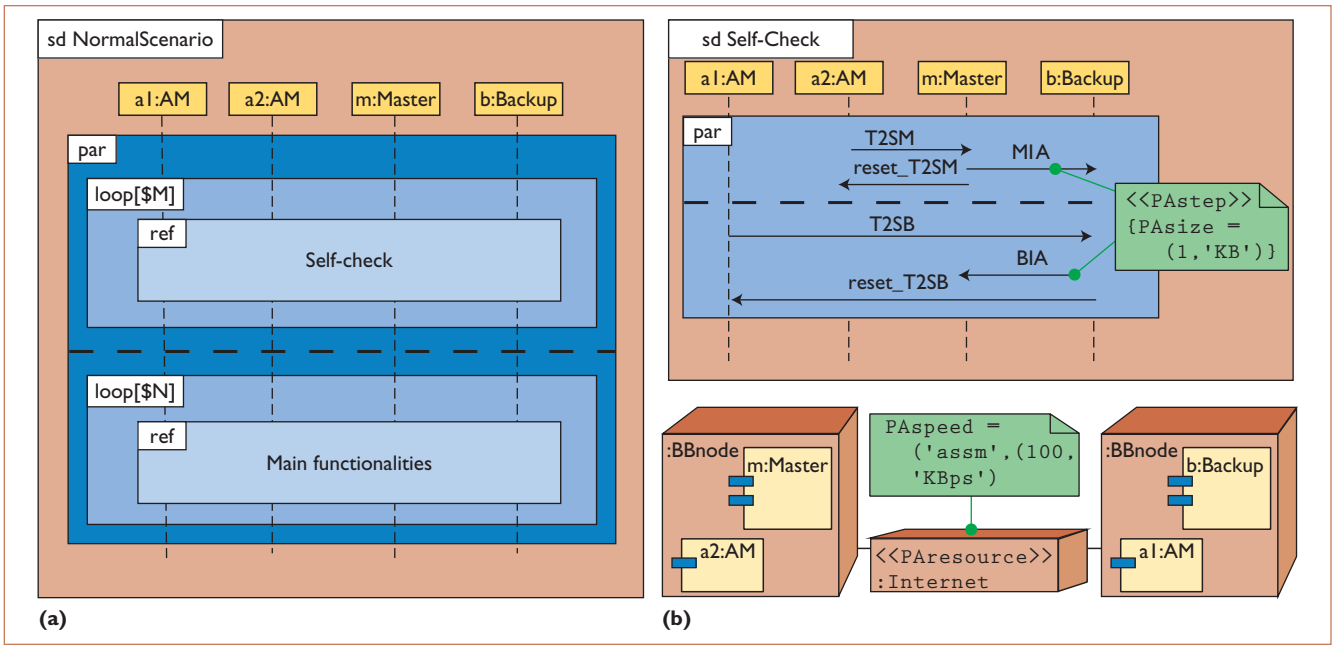

Figure 4. UML specification. (a) Sequence diagrams (SDs) let us represent scenarios of interest. (b) Deployment diagrams (DDs) let us represent system architectures, particularly hardware and software redundancies.

sures: required values (annotated in yellow), assumed values (green), estimated values (blue), and measured values (not used).

SPT models are the basis from which we derive performance models, which we can use for our quantitative analysis, adopting existing model-analysis techniques and tools. (We express performance models in terms of GSPNs, so we use the GSPN formalism's analysis tools. The "Stochastic Modeling using Petri Nets" sidebar has more information.) Moreover, the SPT doesn't hamper the way software engineers use UML to construct their models. Thus, we can follow the FT component-modeling guidelines (M1 through M4) within the SPT framework to model FT applications' performance measures and QoS requirements. SPT annotations let us address three main concerns in stochastic modeling: time, probability, and load.

FT applications carry out activities to provide, among others, either actual services (M1) or system recovery actions (M4). A `Time` annotation models the time spent on these activities. Figure 2a shows examples of time annotations on the `store` and `recoveryM` do-activities (that is, computations in the system). The first annotation states that updating the database service takes 0.01 seconds, whereas the second states that a backup spends 0.01 seconds recovering its master agent.

Sometimes, an FT component must probabilistically select a behavior from a set of possible ones. A `Prob` annotation models this in the FT component's UML-SC, in which more than one transition leaves the state from which the component selects the behavior. We stereotype these transitions as `<<PAstep>>` (the transition is a step in the system computation) with a `PAprob` tag.

The `load` annotation models the system workload — that is, the number of FT components or execution threads expressed as a closed load in the corresponding UML-SC. Figures 2 and 3 show an example of this in the `PApopulation` tags.

We also annotate the QoS requirement T^{td}_{max} and the predicted measure T^{td}, from Equation 1, in the master agent's `Backup Failed` state, as Figure 2 shows. Note that the SPT lets us use variables (`$T_td`), allowing us to parameterize the model. Unfortunately, the SPT lacks the syntax to address certain performance measures and QoS requirements that might be necessary for configuring an FT distributed system. Thus, we've extended the SPT with new annotations:

- *Message-size* and *network-speed* tags are useful for computing message-transmission delay (see the UML-SD and UML-DD annotations in Figure 4).
- QoS requirements that represent relationships

Stochastic Modeling using Petri Nets

Carl Adam Petri introduced Petri nets (PNs) in the early 1960s as a formal and graphical language suitable for modeling concurrent systems. PNs are structurally characterized by places and transitions connected via arcs. PNs incorporate the notion of (distributed) state, called *marking*, and their dynamic behavior is governed by transition-enabling and firing rules. A marking is a distribution of tokens; a transition is enabled if its input places contain at least as many tokens as the multiplicity of the corresponding input arcs. An enabled transition can "fire" to remove as many tokens from its input places as the multiplicity of the corresponding input arcs and add to its output places as many tokens as the corresponding output arcs.

As originally defined, PNs didn't include time concepts; researchers introduced temporal specification in PN models several years later, with different approaches, based first on the use of deterministic timing and then on stochastic timing. They introduced temporal specification in PNs mostly by associating delays with transitions. With stochastic Petri nets (SPNs),[1] in particular, the transition-firing delays are negative exponentially distributed random variables; thus, we can view an SPN as a high-level model that generates a Markov chain.

Although classical PNs are useful for investigating logical properties (such as boundedness, fairness, and the absence of deadlocks), SPNs present a powerful modeling formalism that lets us conduct quantitative analysis and then performance-metric prediction. The typical steps in performance evaluation with SPNs are

- modeling the given system via an SPN;
- generating the Markov process and corresponding Markov chain;
- computing the steady-state probability distribution of the marking process's states; and
- obtaining the required performance metrics from the steady-state probabilities.

Several tools are available for SPNs, including the Graphical Editor and Analyzer for Timed and Stochastic Petri Nets (GreatSPN; www.di.unito.it/~greatspn/index.html), which lets us model, validate, and evaluate distributed systems' performance using generalized SPNs and their colored extensions (stochastic well-formed nets). GSPNs are well-known extensions of SPNs,[2] in which certain transitions fire in zero time (immediate), with priorities over stochastic transitions (timed). We've used this class of SPN as the target for our performance model in this article.

References

1. M.K. Molloy, "Performance Analysis using Stochastic Petri Nets," *IEEE Trans. Computers*, vol. 31, no. 9, 1982, pp. 913–917.
2. M. Ajmone Marsan et al., *Modeling with Generalized Stochastic Petri Nets*, Wiley, 1995.

among variables are expressed as constraints, (see the red annotation in Figure 3).

- Nonstandard QoS requirements, such as communication overhead (O^{net} in Equation 1), can't stereotype a concrete UML core concept. The `$N` and `$M` annotations in the UML-SD are useful, for example, for defining O^{net} as `$M`/(`$N` + `$M`). Observe that messages have the same size in this case. We can thus express O^{net} as the percentage of MIA/BIA messages sent with respect to the total messages Backbone sends and receives over the Internet.
- When we use UML-SC to explicitly model faults, we can specify their quantitative characterization via fault frequency and latency tags, which are associated, respectively, with `<<FTstep>>` transitions (representing fault occurrences) and do-activities (representing fault-latency duration).

The new annotations, together with the UML-SPT annotations, give us a framework for modeling the system's stochastic view. It complements the system's functional view, defined using UML diagrams. Thus, we homogeneously model the system using UML and its extension mechanisms, stereotypes, and tagged values.

QoS Assessment via Stochastic Analysis

Once we have a UML system specification, enriched with QoS annotations, we can produce a performance model for assessing an FT application's QoS. The "Generating Performance Models from UML Designs" sidebar (p. 41) describes the general principles that we followed to derive a GSPN performance model from the Backbone UML specification.

Once we generated the GSPN model, we were able to compute estimated values and compare them against the required values. Each estimated value corresponds to a metric prediction in the performance model. The mean return time in the `Backup Failed` state, for example, corresponds to T^{td} and leads us to evaluate the inverse of a transition throughput in the GSPN model.

For effective analysis, we turned to the ArgoSPE tool (http://argospe.tigris.org). Implemented as a plug-in of ArgoUML, the tool lets us

- derive GSPN models from UML- SC, DD, and

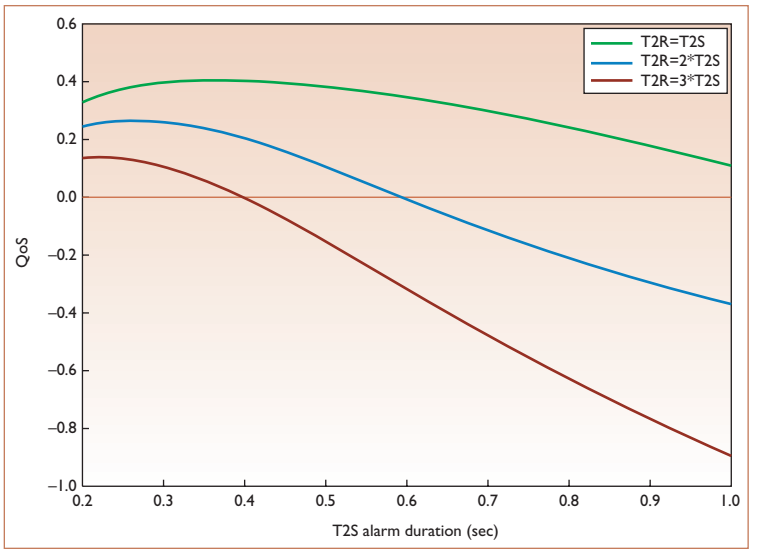

Figure 5. Quality of service in a Backbone application running on two nodes. The three curves represent varying alarm durations for different relations between time-to-send (T_{t2s}) and time-to-receive (T_{t2r}) alarms. When we set all the alarms to the same value, we obtain better performance results (green curve). We reach the maximum QoS when $T_{t2r} = T_{t2s} = 0.34$ seconds.

collaboration diagrams (which are semantically equivalent to SDs);

- submit queries that the tool maps automatically to the corresponding metrics, which the GSPN solver computes on the underlying GSPN model; and
- visualize the predicted metric values directly onto UML diagrams.

For this article's purposes, we used ArgoSPE to automatically produce — from the UML-SCs and the UML-DD — the GSPN model leading the experiments. Given that ArgoSPE doesn't support UML-SDs, we didn't consider them in our analysis; instead, we annotated all the message-size specifications to the UML-SCs' transitions.

We first analyzed a Backbone application running on two nodes. Using sensitivity analysis, we found values for the alarm durations that maximized the QoS function defined in Equation 1: the maximum value gives information on predicted metrics' *goodness* with respect to the given QoS requirements. That is, the closer the maximum is to 1, the smaller the predicted metric values. On the other hand, if the maximum is negative, at least one QoS requirement will remain unsatisfied.

Figure 5 plots the QoS versus the duration of time-to-send (T2S) alarms (T_{t2s}, as described in the Backbone Case Study section): the three curves represent the QoS for different relations between T2S and the duration of the time-to-receive (T2R) alarms (T_{t2r}, as described in the Backbone Case Study section). When setting T2R = T2S (green curve), we reach the maximum QoS at T2S = 0.34 seconds — in this case, Backbone satisfies both QoS requirements (T^{td} = 0.51 seconds ≤ 0.9 seconds and O^{net} = 2.9% ≤ 5%).

If T2S = 0.34 seconds, we can still satisfy the QoS requirements when T2R is twice as long as T2S (blue curve), but the predicted T^{td} value is worse than in the previous case (T^{td} = 0.78 seconds). Finally, when T2R is three times longer than T2S (brown curve) and T2S = 0.34 seconds, the time for detecting and isolating a failed backup agent is equal to 1.09 seconds, which is more than the maximum acceptable threshold.

Having established the values for the alarm durations that guarantee satisfied QoS requirements for a two-node Backbone application, we investigated whether such durations were still acceptable for a Backbone application running on N nodes ($N \geq 2$). We considered, for example, N = 3, 6, and 11, and the experiments showed that none of the considered cases satisfied both QoS requirements when T2S = T2R = 0.34 seconds. Indeed, we must set different maximum thresholds for T^{td} and O^{net} depending on the number of nodes.

We applied the modeling and analysis approach illustrated here during validation and verification activities in Depaude. In particular, we adopted it for the performance and dependability evaluation of the FT mechanisms, implemented within Depaude, to provide a benchmark for software engineers, with optimal parameter configurations that guarantee certain QoS levels for automated systems using the Depaude framework.

From our experience with verification and validation activities in Depaude, we learned that one of the main modeling issues is deciding on the specification's level of abstraction. Considering all the system aspects in stochastic modeling isn't a good approach because the performance model becomes too detailed and its analysis intractable.

In fact, model construction should be driven by analysis goals. (What are the necessary QoS requirements for validation? What are the metrics to compute?) Indeed, different models might be necessary in different scenarios. Modeling faults affecting the Backbone components, for example, wasn't necessary for us to validate the QoS

Generating Performance Models from UML Designs

The literature presents several approaches to generating performance models from UML designs. Simonetta Balsamo and colleagues present a survey of the main contributions to such work.[1] In the work we present in the main text, we follow one method that generates a generalized stochastic Petri net (GSPN) model from a UML-annotated design — such annotations include statecharts (SCs) and sequence and deployment diagrams (SDs and DDs).[2] The approach exploits GSPNs' compositional features, letting us master complexity in both defining and implementing the generation process, which consists of three main steps. First, the approach involves translating SCs separately into GSPN models characterized by labeled places and transitions. Labeled places represent mailboxes associated with event types, whereas labeled transitions can represent event generation or event consumption. The second step translates the SD into a GSPN model with labeled transitions, capturing the causal relations between the represented scenario's events as well as the message-transmission delays. The last step presents two choices: we can compose the SC GSPN models over labeled places to produce the performance model of the system, `SysModel`; or we can compose `SysModel` with the GSPN model, over labeled SD transitions, to gain a performance model for a system scenario `ScenarioModel`.

During the translation steps, we use the SPT annotation specified in the UML diagrams to define the performance model's input parameters. In particular, we use `PAdemand` tagged values to define the firing rates of GSPN timed transitions representing the SC's do-activities (that is, computations in the system). The message-size annotation `PAsize`, together with the network-speed annotation `PAspeed` (specified in the DD), define the rate of GSPN timed transitions that model the message-transmission delays. We map the `PApopulation` tagged values into the GSPN's initial marking parameters and use `PAprob` tagged values to assign weights to the GSPN's immediate transitions representing the choices among a set of possible behaviors. Figure A shows a sketch of Backbone's `SysModel` emphasizing the main translation features.

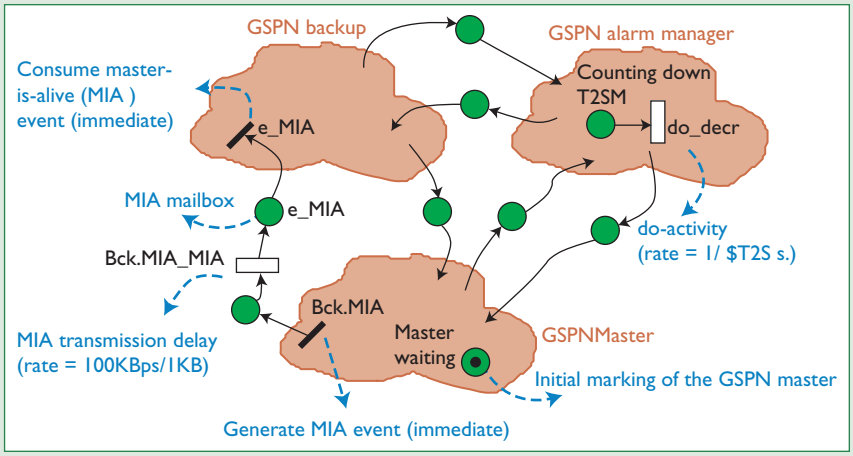

Figure A. Abstract view of the Backbone performance model. Backbone statecharts (master, backup, and alarm manager) are translated into component generalized stochastic Petri nets (GSPNs), which communicate via mailbox places. Inside the GSPNs, you can see the relevant places and transitions obtained by the translation. They represent activities, generated events, and initial marking.

References

1. S. Balsamo et al., "Model-based Performance Prediction in Software Development: A Survey," *IEEE Trans. Software Eng.*, vol. 30, no. 5, 2004, pp. 89–107.
2. S. Bernardi, S. Donatelli, and J. Merseguer, "From UML Sequence Diagrams and Statecharts to Analyzable Petri Net Models," *Proc. 3rd Workshop Software and Performance* (WOSP 02), ACM Press, 2002, pp. 35–45.

requirements we consider in this article; however, to evaluate Backbone availability – which depends highly on fault frequency and latency – faults should be explicitly modeled.

Using tool support is a must when analyzing realistic case studies. ArgoSPE's automatic generation of the performance model shortens this stage considerably, but we could shorten it even more if ArgoSPE computed the metrics for Backbone.

Finally, we must verify several QoS requirements because it isn't easy to keep all the derived predicted metrics under control. Defining QoS functions[7] facilitates system QoS assessment. To this end, we've defined a QoS function for analysis with Backbone that depends on the two equally important requirements discussed at the beginning of this article.

We plan to extend our approach of deriving performance models from UML specification to consider object identities. For some systems, failing to do so in the performance analysis could generate misleading results. For this purpose, we believe that the colored extension of GSPN (stochastic well-formed nets) is a more suitable performance model than GSPN, given that it allows us to generate a scalable model with respect to the system population.

Acknowledgments

Projects TIC2003-05226 of the Spanish Ministry of Science and Technology and IBE2005-TEC-10 of the University of Zaragoza supported this work. We thank the anonymous referees and guest editors who reviewed the article and helped improve it, and Diego Rodriguez, who helped us with the numerical analysis.

References

1. A. Avizienis et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable and Secure Computing*, vol. 1, no. 1, 2004, pp. 11–33.
2. G. Booch, I. Jacobson, and J. Rumbaugh, *The Unified Modeling Language*, Addison Wesley, 1999.
3. *UML Profile for Schedulability, Performance, and Time Specification*, Object Management Group Adopted Specification, Jan. 2005; www.omg.org/docs/formal/03-09-01.pdf.
4. M. Ajmone Marsan et al., *Modeling with Generalized Stochastic Petri Nets*, Wiley, 1995.
5. J. Merseguer, J. Campos, and E. Mena, "Analysing Internet Software Retrieval Systems: Modeling and Performance Comparison," *Wireless Networks*, vol. 9, no. 3, 2003, pp. 223–238.
6. V. De Florio, G. Deconinck, and R. Lauwereins, "An Algorithm for Tolerating Crash Failures in Distributed Systems," *Proc. 7th Int'l Conf. and Workshops in the Eng. of Computer-based Systems*, IEEE CS Press, 2000, pp. 9–17.
7. D.A. Menascé, "Automatic QoS Control," *IEEE Internet Computing*, vol. 7, no. 1, 2003, pp. 92–95.

Simona Bernardi is a researcher at the University of Torino, Italy. Her research interests include software performance and dependability engineering, stochastic modeling, and UML. Bernardi has a BS and an MS in mathematics, as well as a PhD in computer science, all from the University of Torino. She has served as a referee for international journals and conferences. Contact her at bernardi@di.unito.it.

José Merseguer is an assistant professor in the Department of Computer Science and Systems Engineering at the University of Zaragoza, Spain. His research interests include performance and dependability analysis of software systems, UML semantics, and object-oriented software engineering. Merseguer has a BS and an MS in computer science from the Technical University of Valencia and a PhD in computer science from the University of Zaragoza. He has served as a referee for international journals and as a program committee member for several international conferences and workshops. Contact him at jmerse@unizar.es.