Modeling and Analysis of Redundancy Management in Distributed Object-Oriented Systems by Using UML Statecharts

Gábor HUSZERL, István MAJZIK
Budapest University of Technology and Economics
Department of Measurement and Information Systems
Pázmány Péter sétány 1/d., H-1521 Budapest, Hungary

E-mail: `[huszerl|majzik]@mit.bme.hu`

Abstract

The paper presents techniques that enable the modeling and analysis of redundancy schemes in distributed object-oriented systems. The replication manager, as core part of the redundancy scheme, is modeled by using UML statecharts. The flexibility of the statechart-based modeling, which includes event processing and state hierarchy, enables an easy and efficient modeling of replication strategies as well as repair and recovery policies. The statechart is transformed to a Petri-net based dependability model, which also incorporates the models of the replicated objects. By the analysis of the Petri-net model the designer can obtain reliability and availability measures that can be used in the early phases of the design to compare alternatives and find dependability bottlenecks. Our approach is illustrated by an example.

1 Introduction

The increasing need for distributed technologies have led to several distributed object-oriented (OO) middlewares. These systems target the problems of transparent invocation of objects as well as services that are necessary in a distributed environment (ordering of messages, multicast communication etc.). The need for highly available applications lead also to fault tolerant extensions like replication management, checkpointing and recovery.

This trend can be presented by the example of the most important open object-oriented middleware, the Common Object Request Broker Architecture (CORBA). CORBA was specified by the Object Management Group (OMG). It provides an object-oriented infrastructure that allows objects to communicate, regardless of the specific platforms and languages used to implement them. CORBA defines the basic mechanisms for remote object invocation through the Object Request Broker (ORB), as well as a set of services for object management, e.g. Transaction Service or Trader Service. Originally, neither the ORB nor the existing services provided means for building reliable and highly available applications. Different approaches were elaborated to incorporate the necessary extensions into the CORBA framework. Among others Orbix+Isis [8], Electra [10] and Eternal [11] can be mentioned. Recognizing the need for applications that provide high availability, OMG started a standardization process to define fault tolerance in CORBA. The Fault Tolerant CORBA (FT-CORBA) [13] was proposed by leading information technology companies and reached the level of a Final Adopted Specification in 2000.

The fault tolerant distributed OO systems typically share similar concepts. Server-like objects are replicated on different nodes of the distributed system, thus forming a replica group. The clients can transparently invoke one or more objects of the replica group depending on the redundancy scheme (e.g. active or passive redundancy). The fault model assumes object crash failures. It is the responsibility of a replication manager to keep the necessary number of object replicas, i.e. to recover the replica objects or create new replica instead of the crashed one. Accordingly, the behavior of the replication manager has crucial impacts on the availability of the replica group.

The designer of the system needs tool support to construct optimal fault tolerance schemes and to parameterize these schemes in terms of deployment, number of replicas, fault monitoring interval, repair policy etc. Dependability modeling proved to be a useful technique in the early design phases when comparison of the alternative architectural solutions and identification of dependability bottlenecks is necessary. Stochastic dependability models using Markov-chains or Petri-nets can provide numerical reliability and availability figures that can be used to analyze the sensitivity of the system-level measures to component parameter values.

Dependability modeling is usually a manual task requiring expertise and some experience. However, if a detailed model of the system is available, automatic dependability modeling is a promising approach. Nowadays, as the Unified Modeling Language (UML) becomes the de facto standard modeling language of object-oriented systems, the system model is usually available in UML. Accordingly, UML-based automatic model transformation can provide the stochastic dependability model required for the analysis [3, 4].

The paper presents a technique that enables the modeling and analysis of redundancy schemes in distributed object-oriented systems. The replication manager, as core part of the redundancy scheme, is modeled by using UML statecharts. The flexibility of the statechart-based modeling, which includes event processing and state hierarchy, supports an easy and efficient modeling of replication strategies as well as repair and recovery policies. The statechart of the replication manager, as well as simplified behavior of the replica objects, is transformed to a stochastic Petri-net. By the analysis of the Petri-net model the designer can obtain reliability and availability measures that can be used to compare alternatives and analyze sensitivity to parameters. Our analysis is based on an (early) architectural view of the system. The behavioral description of the replication manager is used only to derive the replication management and repair strategy, in this way determine the structure of the dependability model.

In our previous works methods of transformation from full UML statecharts to Petri-nets were proposed [7]. Now we apply this method to the analysis of replication management, present the necessary UML extensions and show the usefulness of the approach. We investigate the modeling of client fail-over, repair and recovery policies and fault management. To do this, we adopt the system structure that is proposed by the FT-CORBA specification.

The paper is structured as follows. In Section 2 we discuss the typical replication strategies in OO systems. Section 3 describes the modeling approach. The model transformation is outlined in Section 4. The last section presents an illustrative example. The paper is closed by a short conclusion.

2 Replication in distributed OO systems

The architecture presented in the FT-CORBA standard clearly separates the typical tasks in a redundancy structure and assigns these tasks to individual objects. In this way the responsibilities and the interfaces of the objects can be clearly defined. In our architectural model, however, we do not restrict the interfaces, mechanisms and other implementation details that are specified by the standard.

In this framework fault tolerance is provided by entity redundancy, i.e. by the replication of objects, fault detection and error recovery. Client objects can invoke the methods of the replicated server objects thus avoiding single point of failures normally caused by single server objects. The client objects should not be aware of the fact that the server objects are replicated (replication transparency) and should not be aware of faults in the server replicas or of recovery from faults (failure transparency). Redundant objects belong to *object groups*, and several object groups can be managed together in a *fault tolerance domain*. In each domain, the creation and maintenance of the object replicas is provided by the *replication manager* (RM) (Figure 1). We do not separate application-controlled and infrastructure-controlled schemes and assume that the replication manager is solely responsible for these tasks. The replica objects are continuously monitored by *local fault detectors* that are deployed on each host. If a replica object fails (crashes) then the local fault detector reports the error to the *fault notifier*. The fault notifier filters and analyzes the incoming error reports and sends a notification to the replication manager. The local fault detectors are monitored by a *global fault detector* that detects when a local fault detector is not available (e.g. in the case of a host failure). When the replication manager receives a notification about the crash of an object replica, then it can initiate the recovery of that replica (by invoking a specific method), or it can remove the crashed object from the object group and create a new replica (by invoking a factory object that is deployed on each host).

The replication style can be one of the standard ones like active, warm or cold passive, stateless etc. but also application-specific style can be programmed. The (transparent) connection between the client and the object group is the responsibility of a *gateway* (GW). Direct connection is handled as a degenerate case of the gateway. The client fail-over strategy determines the behavior of the client when it does not receive the requested service. Usually, a retry mechanism is implemented. In our case we delegate this mechanism to the gateway and we say that the system fails when the necessary number of server replica as required by the gateway is not available.

In order to increase reliability, the global components of the infrastructure, i.e. the replication manager, the global fault detector, the fault notifier and the gateway, can be replicated as well. We assume an active replication (an object fails if there are no available replicas).

The fault model is *object crash*. It means that in the case of an error the object will not provide any response (service) to the clients and will not return to normal operation until an explicit recovery. In this paper we do not model commission faults.

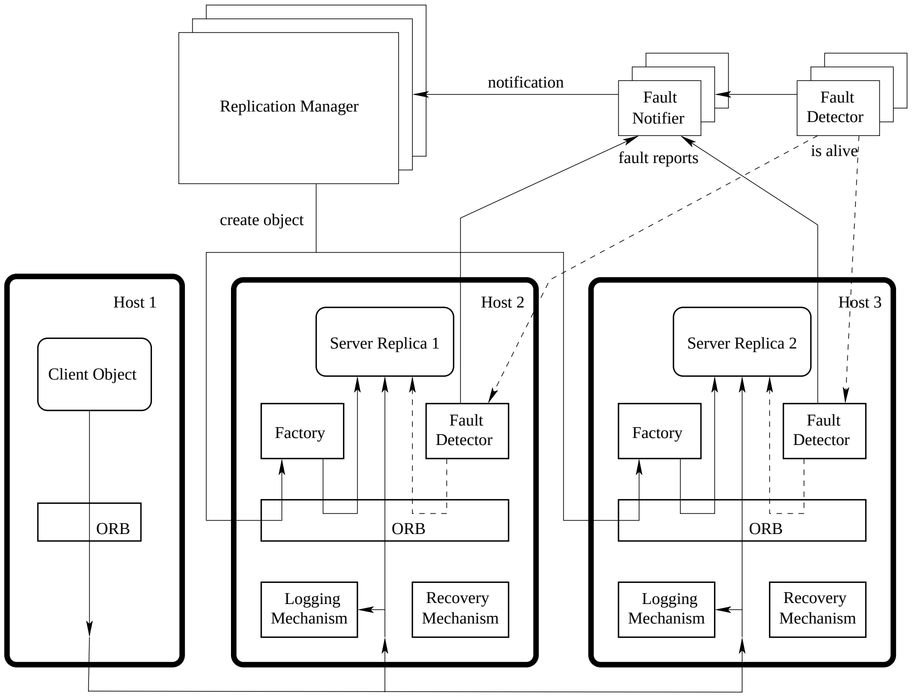

Figure 1. The architecture of the FT-CORBA redundancy structure

3 Modeling of redundancy structures

We are interested in the dependability model available in the early phases of the system design when decisions can be made about the architecture of the system and the applied redundancy scheme (including replication, recovery and repair strategy). Accordingly, we focus on the architecture of the system and abstract from implementation details of the replica consistency, checkpointing and recovery. The main source of information is the object model of the system (available as UML class, object and deployment diagrams) and the dynamic model of the core parts of the system (available as UML statechart diagrams).

The availability of the object group (presented in the previous section) is determined by several factors and parameters. In the following, we outline how they can be represented in (and then extracted from) the UML model.

- Reliability parameters of the replica objects, the hosts and the local fault detectors: The dynamic behavior of these objects is not analyzed. The designer assigns the *failure rate* and the *average time required for recovery* to these objects as *UML tagged values*.

- Reliability parameters of the global components of the infrastructure: They are modeled in a similar way like in the case of the replica objects. The roles of the objects are identified by *UML stereotypes*. Active replication of these objects is taken into account by assigning to them a fault tree consisting of an AND gate (i.e. an infrastructure object fails if all of its replicas fail). Note that this replication style can be refined.

- Replication style: It is the responsibility of the gateway which processes the request of the client and the response(s) of the server object(s). The statechart of the gateway determines the *condition of the system failure*. Usually, this function can be represented by a fault tree.

- Repair and recovery strategy: It is the responsibility of the RM which processes events from the fault notifier and from the factory, and sends messages to the factory and to the replica objects. The *processing of the failure/repair events*, which is defined by the statechart diagram of the RM, determines the repair and recovery strategy.

Accordingly, the most sophisticated model in the redundancy structure is that of the RM. The full modeling power of statecharts is required to be able to describe the conditions and sequences of events/actions that determine when a failed object is recovered, how many replicas are maintained, what is the condition of object removal from the replica group etc. Note that the statechart model of the RM describes only dependability-related behavior, no application-specific details are included. Thus, there is no need to filter out irrelevant states or transitions.

Event	From (sender)	To (receiver)	Function
create	RM	factory	Create an object
initialized	factory	RM	Object is created and initialized
remove	RM	factory	Remove an object
removed	factory	RM	Object was removed
object_fault	fault notifier	RM	Failure of an object
host_fault	fault notifier	RM	Failure of a host
recover	RM	replica	Initiate recovery
recovered	fault notifier	RM	Object is recovered
unrecoverable	fault notifier	RM	Object can not be recovered

Table 1. Events in the redundancy structure

Without restricting the interfaces and the implementation of the system, we assume that the events presented in Table 1 are processed by the RM.

4 Dependability analysis by model transformation

4.1 Dependability sub-models

Our dependability model consists of several sub-models as follows.

- The core part of the dependability model that represents the replica management is generated by an automatic model transformation from the statechart of the RM to a stochastic Petri-net model.
- The replica objects and the global components of the infrastructure are represented by simplified models that are stored in a library of pre-defined sub-models. Note that the simplified models can be replaced by detailed ones when the dependability-related behavior of these components is fully described by statecharts (e.g. in the case of the fault notifier). In this case the automatic model transformation can be used (like above).
- The connection among these sub-models – as defined in the UML object diagram – is provided by the event processing mechanism, i.e. event queues and dispatchers belonging to the objects.

In the following, we outline the model transformation necessary to construct the core part of the dependability model.

4.2 From UML statecharts to stochastic Petri nets

Our analysis of the redundancy management is based on a transformation from UML statechart models to Petri nets with timing and stochastic extensions. Petri nets (PN) are a widely accepted formalism for modeling and analysis of distributed systems. For performance and dependability evaluation extensions of PNs like Generalized Stochastic Petri Nets [1], Stochastic Reward Nets [12] offer not only precise mathematical background but also sophisticated analysis tools. Our choice was the class of Stochastic Reward Nets (SRN). SRNs generalize classical PNs by rewards (various measures) and by assigning guards and distributions of the firing time to transitions. Three SRN tools, SPNP [6], PANDA [2] and DEEM [5] were used in our analysis environment. Dependability measures can be specified by reward functions. In certain cases (e.g. in the case of exponential transition firing times) analytic solution is possible, otherwise simulation has to be performed. If a steady state exists then steady state measures can be computed, otherwise transient analysis can be executed. The analysis of the probability of states identified as representing erroneous behavior leads to reliability (if no repair is modeled) and availability characteristics (if repair is modeled).

Correct SRN representation of the statechart with event processing and state hierarchy needs a thorough analysis of the semantics of both models. Our transformation was defined in a modular way, by introducing a set of SRN *patterns*. These patterns are assigned to peculiar constructs (like event dispatcher) or concepts (like state hierarchy, synchronization) of the UML statechart formalism, this way they help in decomposing the problem and also in proving the correctness of the proposed solutions (according to the informal requirements of the UML semantics as defined in the standard [14]). The source models of the transformation described in this paper are restricted to UML statecharts without history states. Actions are restricted to generation of new events, while events do not have parameters.

According to the semantics of UML statecharts (presented informally in [14] and formalized in [9]), several peculiar concepts have to be taken into account. We discuss them in the following.

4.2.1 Event queue and dispatcher

The events arriving from the environment or from the state machine specified by the statechart itself are collected in the queue and dispatched by the dispatcher one at a time. Event queues provide the interfaces among state machines belonging to different objects. Since UML defines precisely neither the policy of the dispatcher nor the number and distribution of event queues, we have defined patterns for several policies and leave it to the designer to specify the details in the UML model (by using constraints). If the events are selected non-deterministically, then the queue can be implemented with SRNs quite easily. However, FIFO (First In, First Out) is a costly policy in terms of the size and state space of the SRN. Figure 2 presents a FIFO queue for two events ”up” and ”down”.

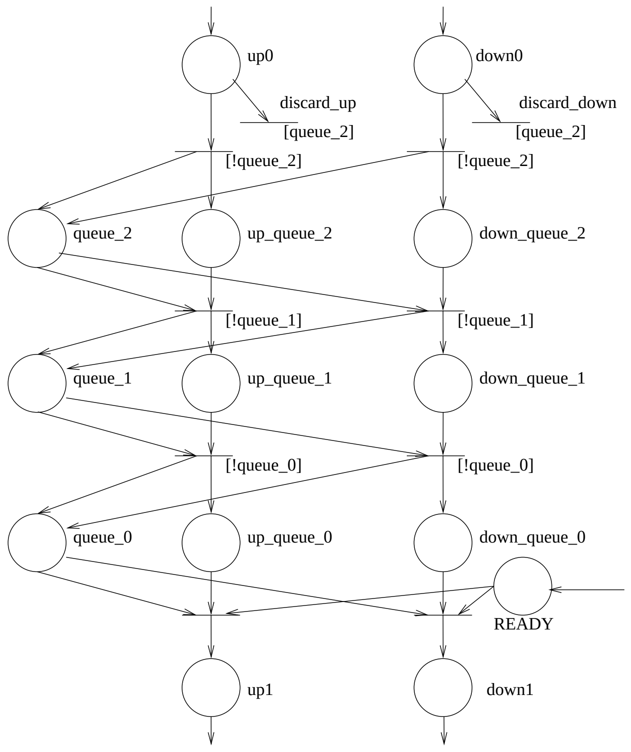

Figure 2. SRN pattern of a FIFO queue of two events

4.2.2 Hierarchy of states and transitions

One important feature of statecharts is the hierarchic structure of states. States can contain sub-states or concurrent sub-machines. Transitions of a statechart may have their source and target states at different levels of the state hierarchy. Due to the concurrency, multiple transitions (triggered by the same event) may be enabled at the same time. Enabled transitions which have common state(s) to exit are in conflict. Some conflicts can be resolved by the priority relation: if a transition has a source state that is substate of the source state of an other transition (being in conflict) then it has higher priority.

From the point of view of the priority, enabled transitions can be represented in the form of a tree according to their source states in the state hierarchy. Transitions on different branches of this tree can fire independently, while the conflicts of transitions being on the same path from the root to a leaf are resolved by the priority relation. Conflicts among transitions emanating from the same state or from different active states, over which the priority relation is not defined, are resolved non-deterministically.

Accordingly, the SRN representing the maximal selection of UML transitions triggered by the same event is a tree of interconnected sub-SRNs (each of them representing a single UML transition) with an additional control structure. This control structure consists of two chains of places. A token runs on one of the chains when the event is “not yet consumed” by the transitions on the given arc of the tree, and a token runs on the other chain when the event is “already consumed”. A joining node of the tree merges the chains of the subtrees. All of the UML transitions in the subtree have higher priority than any transitions along the common path of the tree (on the root-side of the joining node), therefore “event is unconsumed” applies to this common path if and only if the event was not consumed by any of the transitions of the subtree. The “event is consumed” applies to the common path when some of the transitions of the subtree have already fired (they had carried over the tokens on the “consumed” chain) and the other transitions could not fire (they passed on the tokens along the chain). This construction ensures that if the token representing the event reaches the root of the tree, no more sub-SRNs corresponding to transitions of the statechart will fire, the execution step is almost finished.

4.2.3 Semantics of timed transitions

The relationship of timing and guard evaluation is not specified in standard UML. In our approach, time delay is associated with UML transitions, assuming that this delay is the result of program code execution or communication delay. Accordingly, the guard expressions have to be evaluated before the firing of the (timed) transitions. Taking into account the needs of different applications, we implemented three possible semantics for timed and guarded UML transitions: (i) the selection of the transitions is irrespective of timing, (ii) the guard has to be true during the delay else the transition will be deselected and (iii) the “fastest” enabled transition wins.

The corresponding SRN patterns represent the timed transitions of the statechart. The types and parameters of the timed SRN transitions correspond to the ones of the transformed statechart transitions. The timing policy (resampling, race with age/enabling memory) is defined by the designer and must be supported by the SRN-tool used for the analysis.

4.2.4 Step semantics

The transitions of the UML statechart fire in steps, i.e. a stable state configuration is reached only if the maximal set of enabled transitions has already fired. In contrary, SRN reaches a stable state after each firing. The UML semantics requires the evaluation of the guards of the transitions at the beginning of a step, before firing of any transition. Thus the guards refer to the consistent state configuration before the actual step. In SRNs, the guard of a transition will be evaluated just before the given transition fires, the evaluation is not scheduled to the beginning of a “step” and the “results” are not stored. Accordingly, the last stable state configuration of the state machine must be recorded to ensure the correct evaluation of guards. To do that, the places representing the states of the statechart are duplicated. For a state A there is a place A containing a token if and only if the state A was active just before the actual step, and there is an other place A' containing a token if and only if the state A will be active after the actual step. In this way the guards and the transitions changing the state refer to different places. This concept necessitates a synchronization of the duplicated places at the end of each step.

4.2.5 Composition of subnets

The SRN corresponding to a given UML statechart is composed of the subnets introduced in the previous subsections. The subnets are connected with each other using interface places.

The initial state of the SRN is defined as follows. If the event queue contains events in the initial state then these events are represented by the initial marking of the appropriate places of the event queue subnet. The initial state configuration is mapped to the SRN by inserting tokens into the corresponding place-pairs.

5 An example

Our illustrative example is a model of the architecture of a distributed object-oriented system. The application (e.g. an e-commerce application or a hospital patient monitoring system) cannot tolerate long unavailability of the service provided by the system. To achieve this goal active object-level replication is used. In the following presentation we focus on the model of the RM. The model is completed by simplified models for the infrastructure objects and the server replicas.

5.1 Model of the replication management

In our model the RM creates 2 replicas when the server object group is constructed. In case of a host failure the replica deployed on that host is removed from the group and a new one is created on a different host. We do not focus on the repair of hosts, thus we assume here that the number of hosts is not limited. In case of an object failure the RM initiates the recovery of the replica. If the recovery was not successful then the replica is removed. When failure of a just-recovered replica is reported, it is removed without trying to recover again. The replica is also removed when there are no other replicas working. The services of the replicated objects are available as long as there is at least one working replica.

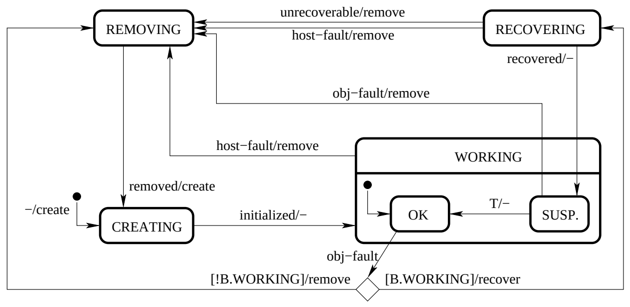

Figure 3. Statechart model of the redundancy manager

The statechart model of the RM consists of 2 concurrent, identical sub-machines (A and B) supervising the 2 replicas. Figure 3 depicts the statechart of one sub-machine (A). (We simplified this Figure by not depicting a ”time-out - retry on different host” mechanism.)

When starting, the RM sends event *create* to the factory of the chosen host. If it has received a message about the successful construction (*initialized*) then it considers the given replica working. In state *Working* two events will trigger transitions. An event *host-fault* moves the component to the state *Removing* sending an event *remove* to the factory of the given host. An *object-fault* moves the component either to the same state (*Removing*) sending *remove* or to the state *Recovering* sending *recover* to the object replica. The choice depends on the state of the other replica (B). In the state *Recovering* the local fault notifier of the host can report the successful recovery of the object, in which case the component moves back to a sub-state of its state *Working*.

The component leaves this sub-state when a timer has expired (the timer is a separate object). If another *object-fault* occurs before this time, the event *remove* is sent to the factory of the replica. The local fault notifier of the host may report the object being unrecoverable by sending an event *unrecoverable* to the RM. In this case *remove* is sent to the factory of the replica. In the state *Removing* the RM waits for an event *removed* from the factory, and after receiving it, it begins with an event *create* again.

The RM is considered to have a FIFO event queue of length 6, capable of accepting 12 different events.

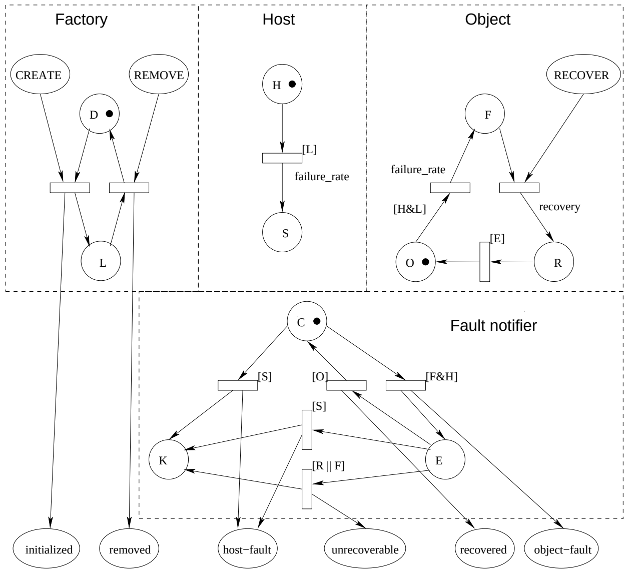

Figure 4. Petri net models of the objects

To analyze the model we transformed it to stochastic reward nets (SRN) by using the patterns introduced above. The other infrastructure objects and the replicas were represented by simplified SRNs assigned to them from a predefined library, on the basis of their role in the object model. Some of them are presented in Figure 4. The 3 ellipses on the top depict the places, where the SRN of the RM can put tokens representing the corresponding events. On the bottom of the figure the 6 ellipses depict the input places belonging to the event queue of the RM. Guards in square brackets refer to the marking of places of concurrent objects.

- The model of an object replica is on the right of the top row in the figure. Failure of the replica is represented by a timed transition (its parameter is the failure rate estimated by the designer). When the object has failed, it can start a recovery phase, when an event *recover* is sent to it by the RM. After a successful recovery it can resume work again.
- The factory constructs and destructs the object replica accepting events *create* and *remove* and sending events *initialized* and *removed*.
- The failure of a host is effectual only if an object replica is deployed on it.
- The fault notifier collects information about the state of the object replica and the host with some delay, and forwards these events to the RM. Its SRN model was constructed from the UML statechart model by using the transformation.

5.2 Measurement results

Size of the model. The SRN model of the system consists of 109 places and 147 transitions. The state space of the underlying Markov-chain is 7,386 tangible states (i.e. states in which the system spends non-zero time), and there are 24,406 transitions among them.

Transient analysis. The analysis answers the question what is the probability of having at least one (or two) working object replicas. In the early phase of the design usually timed SRN transitions with exponential distribution are used in the model and the designer estimates the parameters of the distributions. This assumption enables an analytical solution of the model. Here we assumed the following parameters:

Modeled occurrence	Average time units
Host failure	10,000
Object failure	1,000
Recovery	10
Local fault detection	10
Global fault detection	100
Step of the RM	1

Figure 5 presents the probability of one (two) working replicas. The probability of having (on the long run, i.e. in steady-state) at least one working replica is 99.92%, of having a selected replica working is 97,09% and of having both replicas working is 94,26%.

Comparison of RM strategies. A central question of the early design is the comparison of different architectural solutions. The designer can reduce the design cycle by comparing the solutions and elaborating only the best fitting one. In our example one parameter of the design is the number of object replicas required to achieve the required availability. Other interesting parameter is the time delay considering a recovered object as suspicious. It is also questionable whether the policy of considering a recovered object suspicious (and handling a subsequent failure in this interval in a different way) is meaningful.

The comparison of systems with different number of object replicas is quite easy, the required modification of the dependability model is straightforward.

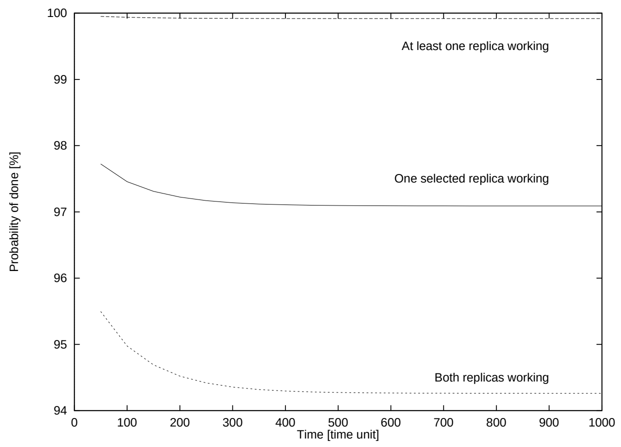

Figure 5. Probability of having working replicas

However, the analysis of the time delay of considering an object being suspicious is more tricky. Assuming exponential distribution of all timing activities, the analysis will show that the availability of the system is not sensitive to the parameter of this time delay (as the length of the interval and the number of failures in this interval are not bound). Naturally, this result points out the ambiguity of the assumption of exponential timing, and not the inappropriateness of the fault handling policy. The analysis can be performed correctly by using SRN models with deterministic timed transitions.

6 Conclusion

We showed in this paper that complex, application-dependent replication strategies of distributed object-oriented systems can be analyzed automatically. The analysis can be performed in an early design phase when the structure of the system and the behavior of the replication manager is defined. On the one hand, the hosts, infrastructure objects and server object replicas can be represented by simplified dependability sub-models (their detailed behavior should not be specified). On the other hand, the designer can use the full power of UML statecharts to describe the core part of the redundancy management, i.e. the behavior of the RM. The statechart of the RM is transformed to an SRN dependability model which is completed by the other sub-models and analyzed by off-the-shelf tools. The optimal replication management can be selected by modeling alternative behaviors of the RM, executing the automatic model transformation and the subsequent dependability analysis.

References

[1] M. Ajmone Marsan. Stochastic Petri nets: An elementary introduction. In G. Rozenberg, editor, *Advances in Petri Nets*, LNCS 424, pages 1–29. Springer Verlag, 1991.

[2] S. Allmaier and S. Dalibor. Panda - Petri net ANalysis and Design Assistant. In *Tools Descriptions, 9th Int. Conf. on Modeling Techniques and Tools for Computer Performance Evaluation (Tools'97)*, St. Malo, France, 1997.

[3] A. Bondavalli, M. Dal Cin, D. Latella, and A. Pataricza. High-level Integrated Design Environment for Dependability (HIDE). In *Proc. Fifth Int. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS-99)*, Monterey, California, USA, November 18-20. 1999.

[4] A. Bondavalli, I. Majzik, and I. Mura. Automatic dependability analysis for supporting design decisions in UML. In *Proc. HASE'99, Fourth IEEE Int. Symposium on High Assurance Systems Engineering*, 1999.

[5] A. Bondavalli, I. Mura, S. Chiaradonna, R. Filippini, S. Poli, and F. Sandrini. DEEM: A tool for the dependability modeling and evaluation of multiple phased systems. In *Proc. IEEE Int. Conf. on Dependable Systems and Networks (DSN)*, New York, June 26-28, 2000.

[6] G. Ciardo, J. Muppala, and K. S. Trivedi. SPNP - stochastic Petri net package. In *Proc. IEEE 3rd Int. Workshop on Petri Nets and Performance Models (PNPM'89)*, pages 142–151., Kyoto, Japan, 1989.

[7] G. Huszerl and I. Majzik. Quantitative analysis of dependability critical systems based on UML statechart models. In *Proc. HASE 2000, Fifth IEEE Int. Symposium on High Assurance Systems Engineering*, 2000.

[8] Isis Distributed Systems Inc. and Iona Technologies Ltd. *Orbix+Isis Programmers's Guide*, 1995.

[9] D. Latella, I. Majzik, and M. Massink. Automatic verification of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[10] S. Maffeis and D. C. Schmidt. Constructing reliable distributed communication systems with CORBA. *IEEE Communications Magazine*, 14(2), 1997.

[11] L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, L. Tewksbury, and V. Kalogeraki. The Eternal system: An architecture for enterprise applications. In *Proc. Int. Enterprise Distributed Object Computing Conf.*, pages 214–222, 1999.

[12] J. K. Muppala, G. Ciardo, and K. S. Trivedi. Stochastic reward nets for reliability prediction. *Commun. in Reliability, Maintainability and Serviceability*, 1(2):9–20, July 1994.

[13] Object Management Group. Fault tolerant CORBA specification v1.0, ptc/2000-04-04. *http://www.omg.org/*, 2000.

[14] OMG. *UML Semantics, version 1.1*. Object Management Group, September 1997.