# A UML-Based Framework for Assessing the Reliability of Software Systems

**Apostolos Zarras**
INRIA
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay - France
+33 1 39 63 5270
Apostolos.Zarras@inria.fr

**Valerie Issarny**
INRIA
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay - France
+33 1 39 63 5717
Valerie.Issarny@inria.fr

**ABSTRACT**
Software reliability can be defined as the probability that a software system successfully performs its designed functions for the duration of a specific mission profile. Modeling and assessing reliability is not a new challenge. As a matter of fact, there exists a variety of techniques for calculating reliability, which are, typically, supported by an underlying modeling formalism like reliability block diagrams, fault trees, reliability graphs, Markov chains, petri nets.

Nowadays, however, UML, has emerged as the software industry's dominant modeling language. Given this fact, in this paper we elaborate on the contribution of UML in modeling quality aspects of software, and in particular aspects that characterize software reliability. In consequence, we propose possible extensions of the UML meta-model. Finally, we investigate how to use a UML model to systematically generate models that serve as input to traditional techniques for assessing reliability. Hence, the main artifact of this paper is a UML-based framework that can be coupled with existing tools, implementing traditional techniques for assessing the reliability of software systems.

**Keywords**
Reliability, Software Architecture, Systems Design, UML.

## 1 INTRODUCTION

In principle, system reliability is a measure of the continuous service accomplishment from a reference initial instant [10]. The previous is often refined into more concrete definitions stating that reliability is the probability that a system successfully performs its designed functions for the duration of a specific mission profile [11]. Assessing reliability is not a new challenge and several techniques for calculating it have been proposed in the past [3]. Those techniques were used in the first place to assess the reliability of hardware and they range from purely analytical ones to simulation techniques. Later, reliability assessment techniques were proved to be also useful towards the assessment of software reliability [17]. Most of the times, the differences and the similarities between different techniques are well-defined and they mainly originate from the kind of systems that they support, their ability to scale well with respect to the size of the inspected system, the precision of the estimations resulting when applying them, etc. Techniques for reliability assessment are, typically, supported by an underlying modeling formalism, which allows specifying structural and behavioral aspects of the inspected system that affect the system's reliability. Well known and widely used examples of such formalisms are reliability block diagrams, fault trees, reliability graphs, Markov chains, and Petri-nets.

Nowadays, UML [1] has emerged as the software industry's dominant modeling language. In a relatively short period of time, UML has become a standard for modeling business, structural and behavioral aspects of software systems [13]. However, it is still not clear how to model quality aspects of the system and how the standard UML formalism contributes towards this purpose. In this paper, we concentrate on software reliability. Given that the basic aspects of software that affect the reliability of an inspected system are more or less known, our first objective is to investigate how UML supports modeling those aspects and how UML can be extended, when needed, towards modeling those aspects. Our second objective is, then, to investigate how a UML model that incorporates information regarding properties that affect the reliability of the inspected system can be exploited in a systematic way towards generating models that serve as input to traditional techniques for reliability assessment. The main artifact of this paper is, hence, a UML framework that can be coupled with tools implementing traditional techniques for reliability assessment.

The rest of this paper is structured as follows. Section 2 briefly presents work related to the use of UML for modeling software quality, and the systematic exploitation of UML models towards the generation of input to tools that implement techniques for assessing software quality. Section 3 gives an overview of traditional techniques for assessing software reliability. This section gives us a starting point with an ensemble of properties that need to be modeled for assessing the reliability of software. Section 4 is our main contribution, the proposed UML framework for the assessment of software reliability. More specifically, we define extensions to the standard UML meta-model, needed so as to model properties identified in Section 3, and we exploit the use of extended UML models for the assessment of software reliability using existing techniques. Section 5, that follows, briefly presents technical details related to the prototype implementation of our framework. Finally, Section 6 gives some concluding remarks.

## 2 RELATED WORK

Work related to modeling software qualities using UML, is so far in a rather primitive stage. To our knowledge, only UML-based performance modeling was addressed in the near past. This knowledge was proved, however, useful towards our approach for modeling and assessing issues of reliability. More specifically, the road-map lecture that was given by Rob Pooley at the ICSE 2000 conference [15], highlighted the need for methodologies towards UML-based performance modeling. Other interesting work co-authored by Pooley includes a paper that gives hints on how to derive queuing network models out of UML design models [16]. Moreover, in [7], the authors investigate the relation between UML collaborations, UML state machines, and Petri-nets, with the latter being widely used for assessing the performance of software. In [5], the authors propose the direct simulation of UML sequence diagrams. Finally, [14] proposes an approach for generating stochastic process algebra models, out of UML collaboration diagrams and state machines.

The work presented in all of the aforementioned papers mostly exploits ways to derive models that conform to formalisms traditionally used for performance modeling. What seems to be missing is at least a discussion that addresses, how to model in UML properties of the system that affect performance, and highlights the implications for the UML meta-model. For instance, modeling properties that affect performance, may impose the extension of the standard UML semantics. The approach presented in [6], is more mature regarding the previous remarks. The authors propose a complete framework for UML-based performance modeling. Following similar steps, in this paper we elaborate on our proposal presented in [18] towards a framework for modeling and assessing software reliability. Finally, let us point here that our approach is further inspired by the work presented in [8], on attribute-based architectural styles.

## 3 TECHNIQUES AND FORMALISMS FOR MODELING AND ASSESSING RELIABILITY

In general, existing techniques for assessing reliability can be divided into three basic categories: combinatorial, Markov-based and simulation techniques.

Combinatorial techniques, are fast and easy to apply. The overall system reliability is obtained through simple combinatorial calculations involving the reliabilities of primitive

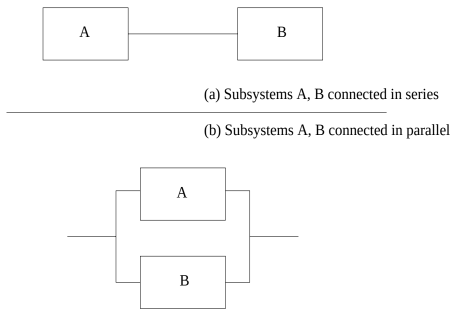

A
B
(a) Subsystems A, B connected in series
(b) Subsystems A, B connected in parallel
A
B

Figure 1: Example of an RBD.

subsystems that constitute the overall system. However, the calculation of the overall system reliability gets more complicated when the primitive parts of the system do not fail independently. In addition to the previous drawback, combinatorial techniques are not quite suited for assessing the reliability of repairable systems. Combinatorial techniques are supported by visual modeling formalisms like Reliability Block Diagrams (RBDs), reliability graphs, and Fault Trees (FTs). For illustration purpose, we give more details regarding RBDs. An RBD is a diagram that represents graphically a constraint for completing a mission. Hereafter, we call such a constraint, *constraint-to-succeed*. More specifically, an RBD depicts a structure of subsystems that need to be operational towards mission completion. Subsystems can be connected using serial, parallel, and possibly more complex connections. For instance, if the completion of a mission requires using a subsystem A and a subsystem B, then both must be operational for the duration of the mission profile. The aforementioned constraint can be specified as a logical formula, $A\ and\ B$, consisting of the conjunction of two predicates. Predicate $A$ (resp. $B$), stands for subsystem A (resp. B) and is true if A (resp. B) has to be operational and false otherwise. Then the resulting RBD, shown in Figure 1(a), depicts subsystem A connected in serial with subsystem B. The reliability in the case where A and B fail independently is the probability that the $A\ and\ B$ formula holds. This probability is calculated in terms of the reliabilities of subsystems A and B as follows: $R_{A\ and\ B} = R_A * R_B$, where $R_A$ (resp. $R_B$) denotes the reliability of subsystem A (resp. B). Similarly, if the completion of a mission requires using a subsystem A, or a subsystem B, then either of them must be operational for the duration of the mission profile. The resulting RBD, shown in Figure 1(b), depicts subsystem A connected in parallel with subsystem B. Reliability graphs are, in principle, similar to RBDs. In contrast to the previous, an FT specifies graphically a constraint, which describes undesired events that lead to mission failure. Undesired events are connected with AND and OR gates.

The second category of techniques for assessing reliability is more suitable for both non-repairable and repairable systems. Moreover techniques belonging to the second category facilitate the modeling of dependent failures. Briefly, given

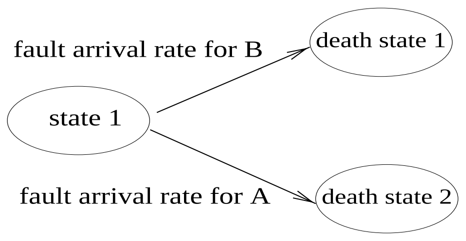


Figure 2: Example of a Markov chain.

a system, and in particular, a part of its configuration used for completing a mission, it is necessary to model how this configuration changes at runtime in the presence of failures and repair actions. Furthermore, in the case of dependent failures, there is a need to model how they spread across the system configuration. A Markov chain of states can then be used to model the previous issues. A state in a Markov chain represents a possible state of the configuration. In general, the configuration can be either in an operational state, or in a failed state (also called a death state). In the former case, the corresponding constraint-to-succeed the mission holds, while in the latter case it does not. A transition between Markov states that models a subsystem failure is characterized either by the arrival rate of the fault that causes this failure, or by the Mean Time Between Failures (MTBF). A transition between Markov states that models a subsystem repair action is characterized either by the repair rate, or by the Mean Time to Repair (MTTR) the subsystem. Given a Markov model that describes all possible transitions in cases of failures and repair actions, a mathematical model can be employed towards calculating the reliability of the configuration. This model involves solving a system of first order differential equations. Taking an example, if the completion of a mission requires using a subsystem A and a subsystem B, then the corresponding constraint-to-succeed is $A \; and \; B$. The resulting Markov model is shown in Figure 2. Three Markov states are defined in the model: `state 1`, which is the initial state where both A and B are operational and, hence, the constraint-to-succeed holds; `death state 1` where A has failed and hence the constraint-to-succeed is false; `death state 2` where B has failed and, hence, the constraint-to-succeed is false. The transition from the initial state to `death state 1` (resp. `death state 2`) is characterized by the arrival rate of the faults that cause the failure of subsystem A (resp. B). Although Markov-based techniques seem more powerful than combinatorial techniques, it is a fact that the manual specification of a Markov model is a laborious and error-prone task that requires spending time and great effort.

Finally, in the last category we have simulation techniques. Simulation techniques can be used for both repairable and non-repairable systems but they are typically slow. In principle, it is possible to simulate the system behavior starting from descriptions of Markov models, RBDs and FTs.

## 4 A UML-BASED FRAMEWORK FOR MODELING AND ASSESSING RELIABILITY

Assessing reliability of a target software system using techniques belonging to the basic categories presented in Section 3 mainly involves deriving a UML design model that delineates the architecture of the target software system and further includes the specification of a mission profile. Moreover, properties that characterize the reliability of the primitive elements making up the system must be incorporated in the design model. Furthermore, the design model must include the specification of constraints-to-succeed a mission, and the potential runtime behavior of the target system configuration, used to realize a mission, in the presence of failures and repair actions.

Hence, our main goal is to provide a UML framework that facilitates the construction of such a design model for assessing the target system reliability. At this point let us assume, without loss of generality, that a typical UML design model comprises 4 distinct architectural views of the system it represents [9], namely: *(a)* The *logical view*, which addresses the functional properties of the target software system. *(b)* The *process view*, which addresses concurrency issues that characterize the runtime behavior of the target software system. *(c)* The *implementation view*, which addresses the organization of an implementation model that complies with the design model described in the process and the logical views. *(d)* The *deployment view*, which addresses the deployment and installation of the implementation model onto a physical execution platform.

In addition, all four views that were previously mentioned are constrained by the *use case view* of the system. The use case view is reflected to a use case model, which contains a set of key scenarios, or use cases, describing abstractly user functional, and non-functional requirements on the services that the system is supposed to provide. In other words, the use case model contains the specification of a set of mission profiles, each one of which describes the functionality expected from the system and measurable quality parameters and tolerances. The use case view of the system is typically constructed during the requirements elicitation. However, its construction is completed at a mature design stage, with specifications describing the exact realization of the missions that it includes.

In the remainder of this section, we detail our framework, which provides a number of basic UML constructs and automated procedures, supporting reliability assessment. Moreover, the use of the framework is described using a simple case study system. The target system consists of a Corba server providing some services. The system further comprises a legacy system (e.g. IBM CICS, PDM) used by the Corba server towards serving requests coming from external Corba clients. Since the legacy system does not belong to the Corba "world", communication between the Corba server and the legacy cannot be achieved directly. To deal with the previous problem, Corba facades are used. Each facade exports a Corba interface that matches the specification of the legacy system. Based on this structure, the Corba server diffuses a request to the facades which in turn call, at-most-once, the corresponding functionality provided by the legacy.

**Supporting the Construction of the Use Case View**
In UML, use case models are described with use case diagrams. A use case diagram comprises a number of use cases, describing parts of the system functional behavior as manifested to external users of the system. Users communicate with use cases and constraints, regarding reliability tolerances required by them, can be added to this association. Use cases may include (i.e. the functionality of a use case includes the functionality of another one), or extend (i.e. the functionality of a use case may be used by the extended use case), other use cases. Again constraints regarding reliability tolerances can be added to the aforementioned relationships.

To provide the ability to define reliability constraints required by external users, we provide a UML stereotype called `ReliabilityRequirements`. The base class of this stereotype is the binary association. The stereotype can only be used to associate a user with a use case. Moreover, `ReliabilityRequirements` extends the semantics of the standard UML association with a property called `reliability` that is defined for the associated use case, and whose value represents the reliability tolerances required by the associated user from this use case. Formally, the semantics of the above stereotype are given in the following OCL specification [1]:

```
ReliabilityRequirements:
  self.baseClass.oclIsKindOf(Association) and
  self.baseClass.connection->size = 2 and
  self.baseClass.connection->exists(
   c :  AssociationEnd |
   c.type.oclIsKindOf(Actor)) and
  self.baseClass.connection->exists(
   c :  AssociationEnd |
   c.type.oclIsKindOf(UseCase)) and
  self.requiredTag->exists(t :  TaggedValue |
   t.name = 'reliability' and
   t.value.oclIsKindOf(Real) and
   t.value <= 1 and t.value >= 0)
```

Similarly, to provide the ability to define reliability constraints required by use cases, from uses cases that extend them, or uses cases included by them, we define the `ReliablyExtends`, and the `ReliableInclusion` stereotypes, which specialize the `Extends`, and the `Includes` standard stereotypes of the `Generalization` relationship respectively. Instances of the `ReliablyExtends`, and the `ReliableInclusion` stereotypes define a property, whose value gives the reliability constraint that should be met by, either a use case included by another use case, or a use case that extends another use case. Formally:

```
ReliablyExtends:
  self.baseClass.oclIsKindOf(Generalization) and
  self.requiredTag->exists(t :  TaggedValue |
   t.name = 'reliability' and
   t.value.oclIsKindOf(Real) and
   t.value <= 1 and t.value >= 0)
ReliableInclusion:
  self.baseClass.oclIsKindOf(Generalization) and
  self.requiredTag->exists(t :  TaggedValue |
   t.name = 'reliability' and
   t.value.oclIsKindOf(Real) and
   t.value <= 1 and t.value >= 0)
```

[1]We do not give the precise meaning of the basic constructs used in the OCL formulae that follow, as they are quite direct to infer; the interested reader may refer to the UML semantics document [13] and to the OCL specification [12]

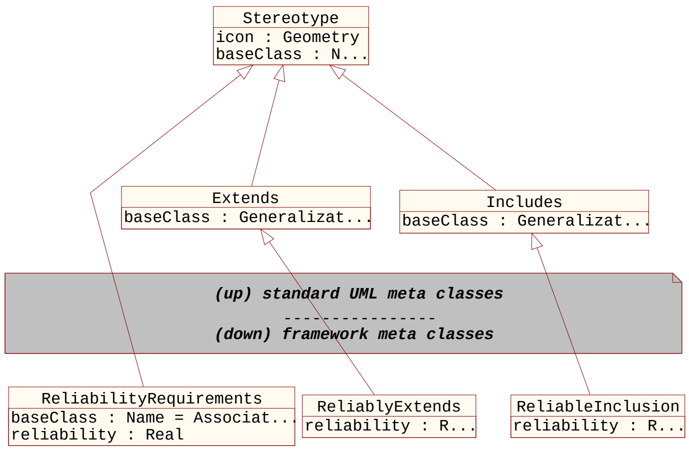


Figure 3: Framework: UML meta classes used in the specification of the use case view.

Figure 3 summarizes the basic stereotypes defined by our framework, supporting the construction of the use case view. Moreover, Figure 4 gives the use case view of our case study system, mentioned at the beginning of this section. A critical use case, called `Mission Profile`, is defined. This use case includes the functionalities of the `CorbaServer` and the `LegacySystem` use cases. Finally, a user is associated to `Mission Profile` through an association that is an instance of the `ReliabilityRequirements` stereotype. The constraint attached to this association is that the reliability of the `Mission Profile` should be at least 0.90.

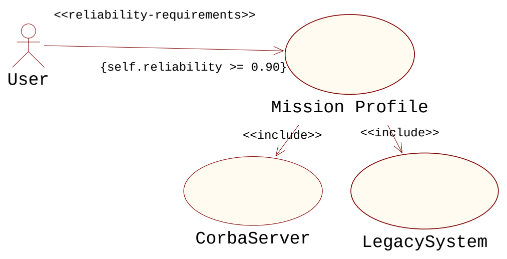


Figure 4: Case Study : Use case view of the target system.

**Supporting the Construction of the Logical View**
Typically, the logical view of a system comprises the definition of a number of classes that realize functional require-

ments specified in the use case view. Moreover, the logical view includes the specification of associations and dependencies among the defined classes. The logical view is usually organized into packages and subsystems. Subsystems are units of functionality, organizing different parts of the system, which can be designed, implemented and repaired independently. A subsystem provides interfaces realized by classes or subsystems contained by the subsystem. A subsystem is primitive if its specification contains the definition of classes.

In addition to the previous, and towards modeling and assessing the system reliability, the logical view must include for each primitive subsystem, the specification of its failure behavior. A *failure* denotes the subsystem's inability to perform its desired function because of errors in the subsystem, or its surrounding environment. An *error* is the effect of a *fault*, in the state of the subsystem [10]. Since the UML meta-model does not define elements whose semantics are at least close to the ones mentioned above for failures, errors and faults, we are obliged to extend it with the corresponding stereotypes, named `Failure`, `Fault`, `Error`. The base class of those stereotypes is the UML `Classifier`.

The `Failure` stereotype is further characterized by two properties called `domain` and `perception` [10]. The former property allows distinguishing between failures affecting the timing delivery of a service and failures affecting the value of the delivered service. The latter property allows distinguishing between failures for which both the system and the user have the same perception, and failures for which the previous does not hold. The `Failure` stereotype is specialized into `StoppingFailure`, which in turn is specialized into `OmissionFailure`, as shown in Figure 5. In OCL, we have:

```
Failure:
  self.baseClass.oclIsKindOf(Classifier) and
  self.requiredTag->exists(
   t1, t2 :  TaggedValue |
   t1.name = 'domain' and
   t1.value.oclIsKindOf(String) and
   t2.name = 'perception' and
   t2.value.oclIsKindOf(String)
)
```

The `Fault` stereotype is characterized by three properties, namely `nature`, `origin`, `persistence` [10]. The former allows, distinguishing among accidental and intentional faults. The `origin` property allows distinguishing between design faults, operational faults, physical faults, etc. Finally, the persistence property allows distinguishing among permanent, transient and intermittent faults, leading to the definition of the corresponding stereotypes, shown in Figure 5. Formally, in OCL we have the following semantics:

```
Fault:
 self.baseClass.oclIsKindOf(Classifier) and
 self.requiredTag->exists(t1,t2,t3:TaggedValue|
  t1.name = 'nature' and
  t1.value.oclIsKindOf(String) and
  t2.name = 'origin' and
  t2.value.oclIsKindOf(String)
  t3.name = 'persistence' and
  t3.value.oclIsKindOf(String))
```

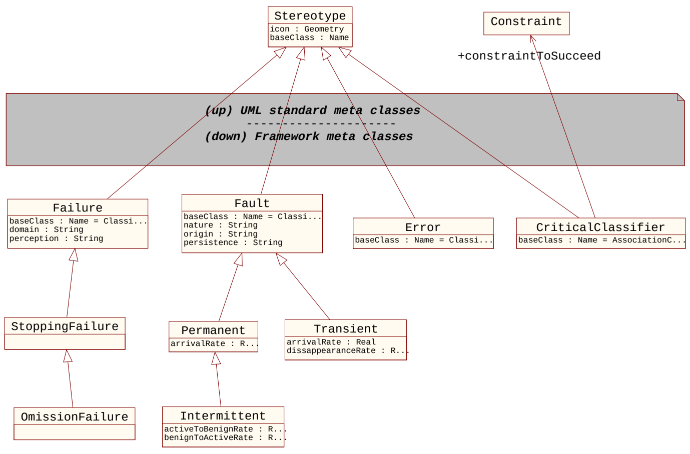


Figure 5: Framework: UML meta classes used for the specification of the failure behavior of primitive subsystems, in the logical view.

Still, however, we have not defined a meta-level element, that allows describing the semantic relationship between instances of faults, failures, errors and primitive subsystems. In UML, the semantic relationship among a number of model elements is defined using UML associations. Moreover, if the association itself has properties like attributes, operations, or methods it is said to be a class association. Given those remarks and in order to define the semantic relationship among the elements that describe the failure behavior of primitive subsystems, we extend the UML meta-model with a stereotype, named `CriticalClassifier`, whose base class is the `AssociationClass`. A `CriticalClassifier` association is restricted to associate a failure, manifesting a fault, with the fault, which in turn is associated with the error caused in the state of a failed subsystem. Finally, an associated failure qualifies an associated subsystem as being either operational, or failed. This qualification is achieved in UML through the definition of a qualifier, included in the definition of the `CriticalClassifier` association. A UML qualifier is a set of attributes whose values allow distinguishing between subsets of associated elements. Note that we do not impose restrictions on the range of values of the qualifier we define for the `CriticalClassifier` association, as it depends on the kind of faults that may be raised by an associated subsystem. A user of the framework must, hence, define the range of values for the qualifier, whenever he uses an instance of the `CriticalClassifier` stereotype. The qualifier can then be used to define a constraint-to-succeed for the instance. This constraint is associated with the `CriticalClassifier` stereotype through the `constraintToSucceed` association, as shown in Figure 5. In OCL we have the following specification of the `Criti-`

`calClassifier` stereotype:

```
 CriticalClassifier:
   self.baseClass.oclIsKindOf(AssociationClass) and
   self.connection->size = 4 and
   self.connection->exists(
    ae1, ae2, ae3, ae4 :  AssociationEnd |
    ae1.type.oclIsKindOf(Failure) and
    ae2.type.oclIsKindOf(Fault) and
    ae3.type.oclIsKindOf(Error) and
    ae4.type.oclIsKindOf(SubSystem) and
    ae1.qualifier->exists(a :  Attribute |
     a.type = Enumeration))
```

Going back to our case study example, Figure 6, gives the part of the target system logical view, that describes the failure behavior of the `CorbaServer` subsystem. The subsystem may fail, and its failures are caused by permanent faults. Failures, faults, and errors are associated with `CorbaServer` through the `CriticalCorbaServer` association, which is an instance of the `CriticalClassifier` stereotype. The qualifier defined for the `CriticalCorbaServer` association defines an attribute that may have two values, namely `OK` and `FAILED`, allowing to qualify instances of the `CorbaServer` subsystem as being either operational, or failed. To succeed to a mission, instances of `CorbaServer` should belong to the set of associated elements, qualified with the OK value. Hence the constraint-to-succeed is defined in OCL as:

```
 CorbaServer:
 self.constraintToSucceed.body =
   self.manifest.subsystem[OK]->includes(
    self.subsystem)
```

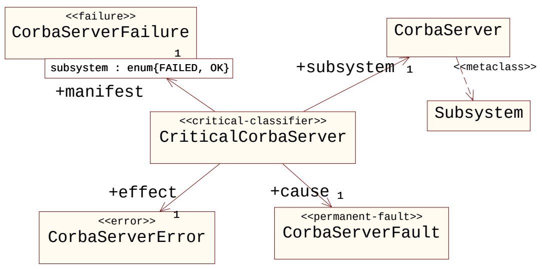


Figure 6: Case Study : Logical view of the CorbaServer primitive subsystem.

## Supporting the Construction of the Process View

An important issue of concurrency that affects the target system reliability is redundancy. More specifically, at a mature design stage, it is possible to identify critical points within the design model where some sort of redundancy is going to be used to increase the system reliability. The identification of those points can be done based on the values of the properties that characterize the failure behavior of subsystems. In UML, redundancy can be modeled in terms of class associations. The basic semantic relationships we need to express regarding redundancy are: *(a)* Independent instances of a subsystem are grouped together and they behave as a single fault tolerant instance. *(b)* Not all of the group members are absolutely necessary for the successful accomplishment of a mission. Hence, we define a stereotyped class association, called `NModularRedundant`, which has the aforementioned semantics. Instances of the `NModularRedundant` stereotype should define a constraint-to-succeed, which states the exact number of replicas that need to be operational for the group to be operational. In OCL we have the following well-formedness constraints:

```
 NModularRedundant:
  self.baseClass.oclIsKindOf(AssociationClass) and
  self.connection->size = 2 and
  self.connection->forall(ae :  AssociationEnd |
   self.connection->exists(ae' :  AssociationEnd |
     ae.type = ae'.type and ae <> ae'))
```

Moreover, it is often the case that instances belonging to a group can be repaired, i.e. removed, or replaced by spare replicas, in the presence of failures. In such a case, the repair behavior of a group must be specified. As with the case of the failure behavior of subsystems, the repair behavior of a group is characterized by a set of properties, which includes the repair-rate and the Mean Time to Repair (MTTR). Hence, to provide means for modeling the behavior of repairable groups, we define a stereotype, called `NModularStandbyRedundant`, which specializes the `NModularRedundant` stereotype. In addition, it defines properties whose values correspond to the repair-rate and the MTTR group instances. Finally, the `NModularStandbyRedundant` association qualifies instances of the associated classifier as being spares, or replicas. In OCL, we have the following well-formedness constraints:

```
 NModularStandbyRedundant:
  self.requiredTag->exists(t1, t2:  TaggedValue |
   t1.name = 'repairRate' and
   t1.value.oclIsKindOf(Real) and
   t2.name = 'MTTR' and t2.value.oclIsKindOf(Real))
  self.qualifier->exists(a :  Attribute |
   a.name = 'kind' and a.type = Enumeration and
   a.type.literal = {REPL, SPARE})
```

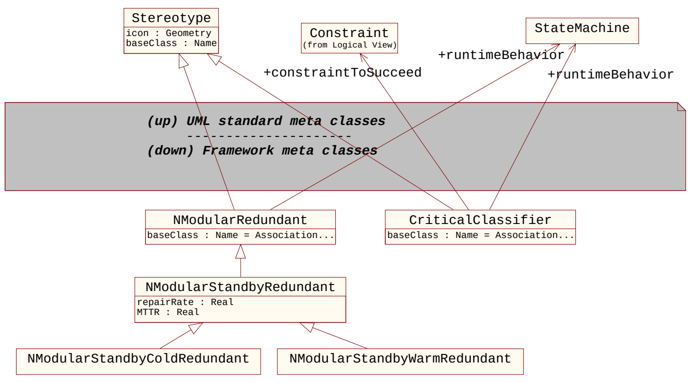


Figure 7: Framework: UML meta classes used in the specification of the process view.

Several specializations of the `NModularStandbyRedundant` association can be further defined but their semantics are not detailed here. In particular, different specializations can be defined regarding different kinds of spare instances (e.g. cold, or warm instances), and different repair policies confronting different kinds of faults. Figure 7 gives an overview of the meta-classes provided by our framework towards the construction of the process view.

Another issue of concurrency that must be specified in the process view towards the assessment of the target system reliability is the way critical subsystems and replicated groups of critical subsystems behave at runtime in the presence of failures and repair actions. In UML, runtime behavior is typically described in terms of State Machines. Hence, as shown in Figure 7, the `CriticalClassifier` stereotype and the `NModularRedundant` stereotype are associated with a `StateMachine`.

Going back to our example, Figure 8 gives the process view of the `CorbaFacade` subsystem. Instances of this subsystem represent facades that mediate the interaction among the `CorbaServer` and the `LegacySystem`. As in the case of the `CorbaServer` subsystem (see Figure 6), an instance of the `CriticalClassifier` stereotype, called `CriticalCorbaFacade`, is defined. This instance is used to associate failures, faults, and errors with the `CorbaFacade` subsystem. Multiple independent instances of the `CriticalCorbaFacade` type form a replicated group called `CorbaFacadeGroup`. The multiple independent instances are divided into a set of active replicas and a set of spares used to repair failed replicas. The `CorbaFacadeGroup` defines a qualifier, named `kind`, which is used to distinguish between the aforementioned two sets. The constraint-to-succeed associated to the group states that at least one replica, member of the group, must be operational for the group to be operational. In OCL we have:

```
CorbaFacadeGroup:
self.constraintToSucceed.body =
self.kind[REPL].manifest.subsystem[OK]->size>=1
```

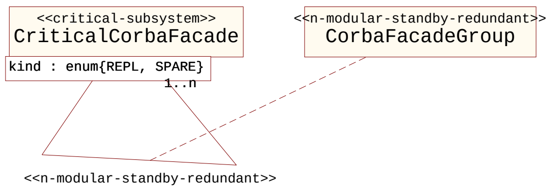
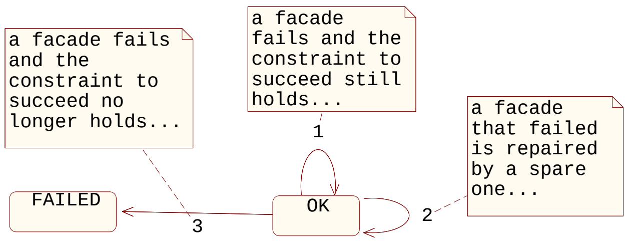


Figure 8: Case Study : Process view of the CorbaFacade primitive subsystem.

Figure 9 gives the state machine that describes the runtime behavior of the `CorbaFacadeGroup` in the presence of failures and repair actions. Two states, named `OK` and `Failed`, are defined. The former represents all runtime states of the group where the group is operational (i.e. the constraint-to-succeed defined above holds), while the latter represents all runtime states of the group where the group is failed (i.e. the constraint-to-succeed defined above does not



a facade fails and the constraint to succeed no longer holds...
a facade fails and the constraint to succeed still holds...
a facade that failed is repaired by a spare one...
FAILED
OK
1
2
3

Figure 9: Case Study : Runtime behavior of the CorbaFacadeGroup.

hold). Moreover, three state transitions are defined. Transition `1` states that if a failure takes place when the number of replicas is greater than two, the group remains operational and the number of working replicas is reduced by one, while the number of failed replicas is increased by one. The detailed specification is the following:

```
1:
-- Event
CorbaFacadeFailure
-- Guard condition
self.constraintToSucceed and
self.kind[REPL].manifest.subsystem[OK]->size > 1 /
-- Actions after transition
self.kind[REPL].manifest.subsystem[OK]->size-- and
self.kind[REPL].manifest.subsystem[FAILED]->size++
```

Transition `2` states that if there exist at least, one failed replica and one available spare, the failed replica is substituted by the spare. The detailed specification is the following:

```
2:
-- Event
Unspecified
-- Guard condition
self.constraintToSucceed
and
self.kind[REPL].manifest.subsystem[FAILED]->size >=1
and
self.kind[SPARE].manifest.subsystem[OK]->size >= 1 /
-- Actions after transition
self.kind[REPL].manifest.subsystem[OK]->size++
and
self.kind[REPL].manifest.subsystem[FAILED]->size--
and
self.kind[SPARE].manifest.subsystem[OK]->size--
```

Transition `3` states that the group gets into a death state if a failure takes place, while there is only one working replica left. The detailed specification of this transition is similar to the one of transition `1`.

**Supporting the Construction of the Implementation View**
Typically, building the implementation view of the target software system comprises organizing the target system implementation into implementation subsystems, conforming to the design subsystems organization provided in the rest of the architectural views that were discussed in the previous subsections. In principle, there is nothing additional to be

modeled regarding reliability and, hence, we do not further detail this view.

**Supporting Construction of the Deployment View**
The deployment view describes how the target software system implementation is deployed on top of a particular execution platform. Assessing the reliability of the target software system regarding both software and hardware, requires specifying the failure behavior of the nodes that make up this execution platform. Our framework, as detailed so far, provides the basic constructs allowing to accomplish the previous task.

**Completing the Use Case View**
The use case view of the system is completed when the design of the system has reached a mature stage, with specifications describing how use cases are realized using elements that constitute the target system architecture. Typically, each use case is associated with a set of collaborations, describing how model elements interact to accomplish a corresponding mission. Assessing the reliability of the system for a particular mission thus requires defining constraints to succeed for the aforementioned collaborations. Moreover, assessing reliability requires describing the behavior of each collaboration in the presence of failures and repair actions. Hence, as in the case of `CriticalClassifier` stereotype, which extends the definition of the `Classifier` element, we define a stereotype, called `CriticalCollaboration`, whose base class is the standard UML `Collaboration` meta-class. The `CriticalCollaboration` meta-class is associated with `Constraint` describing constraints-to-succeed and a state machine describing the runtime behavior of `CriticalCollaboration` instances in the presence of failures and repair actions. Figure 10 shows the relation of the `CriticalCollaboration` with the standard UML meta classes.

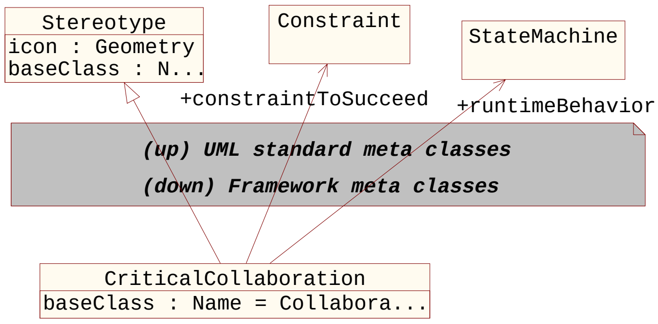


Figure 10: Framework: UML meta classes used in the specification of use case realizations.

Going back to our example, Figure 11 gives the critical collaboration that realizes the `Mission Profile` use case shown in Figure 4. More specifically, the critical collaboration contains an instance of type `CriticalCorbaServer`, named `server`, an instance of type `CorbaFacadeGroup` which represents two replicas of type `CriticalCorbaFacade` and a spare one, and an instance of type `CriticalLegacySystem`. The interaction that takes place among those instances is a follows. The server instance diffuses a request to both replicas contained by the `CorbaFacadeGroup`. Then, this request is transmitted by the group instance at most once to the legacy system.

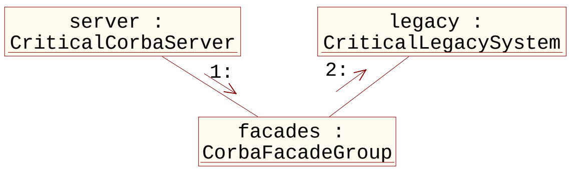


Figure 11: Case Study : Use case realization of the `Mission Profile`.

At this point, let us highlight the fact that in the architectural views detailed so far, the user of our framework is supposed to define the runtime behavior and constraints-to-succeed, for primitive subsystems making up a system. Moreover, as noted, a typical collaboration describes the interaction among instances of those primitive subsystems, or subsystems made out of those primitive subsystems. Consequently, the constraints-to-succeed and the runtime behavior, defined for the collaboration, are expressed in terms of the constraints-to-succeed and the state machines, defined for the primitive subsystems. Given the previous remarks, our framework provides systematic procedures that automate the construction of constraints-to-succeed and of state machines, for critical collaborations, using information spread all over the different views described so far.

*Deriving constraints-to-succeed*
In the UML meta-model, a collaboration is associated with a set of interactions taking place through the elements used within the collaboration. Hence, a constraint-to-succeed, defined for a collaboration, must state that each interaction needs to be successful for the collaboration to be successful. Consequently, the constraint-to-succeed is the conjunction of a set of constraints-to-succeed, each one of which corresponds to an interaction. The following OCL formula specifies how to navigate through the UML meta-model associations to construct the constraint-to-succeed:

```
CriticalCollaboration:
self.interaction->forall(i :  Interaction |
 self.constraintToSucceed.body.stm->includes(
  getConstraintInter(i))
```

Still, in the UML meta-model, an interaction is associated to a collection of messages sent through the interaction. A message is associated to an action that takes place upon the reception of the message. It is often the case that an action is guarded by a boolean condition, whose value determines whether an action is performed, or not, and consequently whether a message needs to be sent for the completion of a mission, or not. Hence, a constraint-to-succeed, for an interaction, is the conjunction of a set of conditional expressions, each one of which corresponds to a message. Each such expression should state that if a message needs to be sent, then it has to be successful, for the overall interaction to be successful. The OCL formula that follows specifies the way to navigate through the UML meta-model associations to systematically derive the constraint-to-succeed for an interaction:

```
CriticalCollaboration:
getConstraintInter(i : Interaction): Expression
post:
i.message->forall(m : Message |
  (if(m.action.recurrence.oclIsKindOf(
    BooleanExpression)) then
   Res.body.cond->includes(m.action.recurrence)
  endif) and
  Res.body.stm->includes(getConstraintMess(m))
```

A message in the UML meta-model is associated to a receiver and a sender, which in our case are instances of critical classifiers. Furthermore, a message is associated to a set of messages whose completion precedes the execution of the current message. Finally, a message is associated to the message that activated it. Hence, the constraint-to-succeed, for a message simply states that if preceding messages and the activating message were successful, both the sender and the receiver of the messages need to be operational, for the message to be successful. Moreover, the critical nodes on top of which the sender and the receiver execute need to be operational. In OCL we have the following formula, describing how to build systematically a constraint-to-succeed for a message:

```
CriticalCollaboration:
getConstraintMess(m :Message) : Expression
post:
  m.predecessor->forall(p : Message |
   Res.body.cond->includes(
    getConstraintMess(p))) and
  Res.body.cond->includes(
   getConstraintMess(m.activator)) and
  Res.body.stm->includes(
   m.sender.base.constraintToSucceed.body) and
  Res.body.stm->includes(
   m.receiver.base.constraintToSucceed.body) and
  Res.body.stm->includes(
   getConstraintNodes(m.sender)) and
  Res.body.stm->includes(
   getConstraintNodes(m.receiver))
-----------------------------------------
getConstraintNodes(cr : ClassifierRole)
     : Expression
post:
  cr.implementation->forall(c : Component |
   component.deployment.forall(n : Node |
    Res.body.stm->includes(
     n.constraintToSucceed.body)))
```

Applying the previous procedure in the collaboration shown in Figure 11, results in parsing message `1, 2`. Given that messages are not guarded, we have:

```
MissionProfileRealization:
self.constraintToSucceed.body =
 server.base.constraintToSucceed.body and
 facades.base.constraintToSucceed.body and
 legacy.base.constraintToSucceed.body)
```

After building a constraint-to-succeed for a particular collaboration, and given the values of reliability for the individual elements used within it, it is possible to assess reliability, based on a simple combinatorial technique. More specifically, the reliability would be the probability that the constraint-to-succeed holds.

*Deriving state machines*

A state machine that describes the runtime behavior of a collaboration, is simply a combination of independent state machines that specify the runtime behavior of the elements used within a collaboration. Hence, the process that generates such a state machine simply iterates through all the different types of elements used within a collaboration, and for each one of them, creates a new state and associates this state with the state machine that describes the runtime behavior of the element. The precise way to navigate through UML meta-model associations towards performing the previous task is described by the following OCL specification:

```
CriticalCollaboration:
self.interaction.message->iterate(
  m: Message; acc : Set(Classifier) |
   acc->includes(receiver.base) and
   acc->includes(sender.base)
) -> forall ( c: Classifier |
   self.runtimeBehavior.top.substate->includes(
   state : StateVertex |
   state.oclIsKindOf(SubMachineState) and
   state.submachine = c.runtimeBehavior))
```

After assembling the state machines, a complete Markov chain model can be generated using the algorithm described in [4]. Briefly, the algorithm takes as input an initial Markov state. In our case, the initial state consists of a set of elements used within a collaboration, which are modeled as being either operational, or failed. Then the algorithm applies recursively a set of rules describing how this state changes in the presence of failures and repair actions. In our case, those rules are given by the state machine generated according to the process that was previously described. During a recursive step, the algorithm produces a transition to a state derived from the initial one. Depending on the rule that is applied, in the resulting state, one or more elements are modeled as being failed, or repaired, while in the initial state they were modeled as being operational or failed, respectively. If the resulting state is a death state the recursion ends.

## 5 Framework Prototype

The basic concepts of the framework detailed in the previous section are realized in a prototype implementation. Our prototype makes use of the Rational Rose tool [2]. Rose is a commercial tool that provides means of specifying and

[2]http://www.rational.com

organizing UML design models. Moreover, Rose supports a script language, which enables users to extend the basic GUI provided by the tool. A more important feature of the script language is that it allows navigating within the contents of a design model. Navigation is realized through standard meta-model associations/dependencies, or through associations/dependencies defined within a particular user-defined design model. In our particular case, we used to script language to extend the Rose GUI with, per architectural view, widgets that facilitate the definition of the stereotypes detailed in the previous section. Moreover, the script language proved extremely helpful for implementing the procedures that automate the derivation of state machines and the construction of constraints-to-succeed, for collaborations that realize mission critical use cases. Finally, we used the script language to integrate the Rose tool with the SURE/ASSIST[2, 4] tool. The SURE tool provides Markov and semi-Markov techniques for reliability assessment. ASSIST facilitates the generation of a Markov model, starting from abstract specifications of rules describing the failure and repair behavior of systems. Using the script language we managed to generate such descriptions from UML state machines describing the failure and repair behavior of collaborations that realize mission critical use cases.

## 6 Conclusion

This paper proposed a UML-based framework for assessing the reliability of software systems. The main contributions are summarized in the following two points:

- The identification and definition of basic constructs that need to be incorporated within the standard UML meta-model, towards the specification of aspects that affect the reliability of software systems.
- The specification and realization of systematic procedures that facilitate the reliability assessment of software systems, whose UML design models incorporate the specification of aspects that affect their reliability.

Still, however, we believe that certain refinements are needed both at the design and the implementation of the proposed framework. Currently, we are mostly working on enhancing the implementation of the prototype and we seek other tools implementing traditional methods for reliability assessment, which can be integrated with our prototype.

## REFERENCES

[1] UML in Action. *Communications of the ACM*, 42(10):26–70, 1999.

[2] R. Butler and W. Ricky. The SURE Approach to Reliability Analysis. *IEEE Transactions on Reliability*, 41(2):210–218, June 1992.

[3] R. Geist and K. Trivedi. Reliability Estimation of Fault Tolerant Systems : Tools and Techniques. *IEEE Computer*, 23(7):52–61, July 1990.

[4] S. C. Johnson. Reliability Analysis of Large Complex Systems Using ASSIST. In *Proceedings of the 8th Digital Avionics Systems Conference*, pages 227–234. AIAA/IEEE, 1988.

[5] C. Kabajunga and R. Pooley. Simulating UML Sequence Diagrams. In *Proceedings of the 14th UK Performance Engineering Workshop*, July 1998.

[6] P. Kahkipuro. A UML Based Performance Modeling Framework for Object Oriented Distributed Systems. In *Proceedings of UML'99*, Oct. 1999.

[7] P. King and R. Pooley. Using UML to Derive Stochastic Petri-net Models. In *Proceedings of the 15th UK Performance Engineering Workshop*, pages 45–56, 1999.

[8] M. H. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, and H. Lipson. Attribute-Based Architecture Styles. In *Proceedings of the 1st Working Conference on Software Architecture (WICSA 99)*, pages 225–243. IFIP, February 1999.

[9] P. B. Kruchten. The 4+1 view model of software architecture. *IEEE Software*, 12(6):42–50, 1995.

[10] J.-C. Laprie. Dependable Computing and Fault Tolerance : Concepts and Terminology. In *Proceedings of the 15th International Symposium on Fault-Tolerant Computing (FTCS-15)*, pages 2–11, 1985.

[11] V. P. Nelson. Fault-Tolerant Computing Fundamental Concepts. *IEEE Computer*, 23(7):19–25, July 1990.

[12] OMG. *Object Constraint Language Specification*, 1.1 edition, Sept 1997.

[13] OMG. *UML Semantics*, 1.1 edition, Sept 1997.

[14] R. Pooley. Using UML to Derive Stochastic Process Algebra Models. In *Proceedings of the 15th UK Performance Engineering Workshop*, 1999.

[15] R. Pooley. Software Engineering and Performance : A Road-map. In *Proceedings of the 22 International Conference on Software Engineering (ICSE 2000)- The Future of Software Engineering*, pages 189–200. ACM - IEEE - SIGSOFT, June 2000.

[16] R. Pooley and P. King. The Unified Modeling Language and Performance Engineering. *IEE Software*, 146(1):2–10, 1999.

[17] M. Shooman. *Software Engineering - Design/Reliability/Management*. McGraw-Hill, 1983.

[18] A. Zarras and V. Issarny. Assessing Software Reliability at the Architectural Level. In *Proceedings of the 4th International Software Architecture Workshop (ISAW-4)*, pages 11–16. ACM - IEEE - SIGSOFT, June 2000.