Fault Tolerant Software Architectures

Titos Saridakis and Valérie Issarny

No 3350

THEME 1

ISSN 0249-6399

Fault Tolerant Software Architectures

Titos Saridakis and Valérie Issarny

Theme 1 — Réseaux et systemes
Projet Solidor

Rapport de recherche n ° 3350 — — 17 pages

Abstract: Coping explicitly with failures during the conception and the design of software development complicates significantly the designer's job. The design complexity leads to software descriptions difficult to understand, which have to undergo many simplifications until their first functioning version. To support the systematic development of complex, fault tolerant software, this paper proposes a layered framework for the analysis of the fault tolerance software properties, where the top-most layer provides the means for specifying the abstract failure semantics expressed in the initial conception stage, and each successive layer is a refinement towards an elaborated description of a fault tolerant software architecture. We present the logical vehicle that permits reasoning on the equivalence or the compatibility of the various expressions of fault tolerance properties at various abstraction levels. In addition, we propose a mapping schema, which permits the correct transformation of abstract entities into concrete ones, during a refinement process.

Key-words: Architecture Refinement, Fault Tolerance, Formal Specifications, Software Architectures.

(Résumé : tsvp)

Architectures logicielles tolérantes aux fautes

Résumé : La prise en compte de défaillances materielles et logicielles complique de manière significative les phases d'analyse et de conception du logiciel. Cette complexité se traduit par des descriptions du logiciel difficiles à comprendre, et devant être considérablement simplifiées pour réaliser la première version du logiciel. Afin de supporter le développement systématique d'un logiciel tolérant aux fautes, nous proposons une approche hiérarchique pour l'analyse des propriétés liées à la tolérance aux fautes. Dans ce cadre hiérarchique, le plus haut niveau fournit le moyen de spécifier formellement les sémantiques de défaillance requises lors de la phase initiale de la conception, et les niveaux suivants permettent un raffinement graduel vers une description élaborée d'une architecture logicielle tolérante aux fautes. Dans ce rapport, nous présentons notre solution pour raisonner sur l'équivalence ou bien la compatibilité des spécifications de propriétés de tolérance aux fautes des différents niveaux. En outre, nous définissons une relation qui permet de garantir, lors du raffinement, la transformation correcte des entités abstraites en entités concrètes.

Mots-clé : Raffinement d'architectures, Tolérance aux fautes, Spécifications formelles, Architectures logicielles.

1 Introduction

Software development becomes extremely difficult as different *nonfunctional* aspects (e.g. availability, reliability, security, etc) interfere with the *algorithmic* aspects of the software (i.e. the specific purpose that the software serves). The software designer is confronted with numerous problems when trying to refine some abstract specifications into a well organized software structure, or when trying to incorporate existing solutions into a new application framework. Employing the principle of *"separation of concerns"* in order to independently analyze and understand the mutual interferences of the various software aspects, has been doubted since the independence assumption does not always hold. Particularly, in the domain of fault tolerant software, a lot of effort has been made to identify a number of abstractions that capture the important software properties and suppress the irrelevant details with respect to the fault tolerance software properties (e.g. see [5] and [14]).

The focus of this paper is the analysis of fault tolerance software properties, and their incorporation in the description of a software architecture. Existing work in the field of fault tolerant software testifies that the system behavior in the presence of failures forms by its own a separate domain of software conception and design. A number of failure semantics have been defined to describe the possible failure modes of a software, and a wide range of generic fault tolerance techniques have been developed independently from some specific application domain. The consequent plurality of mechanisms providing fault tolerance, renders hopeless an exhaustive search performed by humans for the fault tolerant mechanism that best fits a given application. In addition, without a common basis to underpin the integration of the fault tolerance properties with other software aspects, verifying the correctness of the resulting software becomes impossible.

Fostered by the aforementioned facts, our research aims at providing means to support the systematic analysis of fault tolerance software properties, and the reasoning on the correctness of their integration within a software architecture. We do not seek to provide some linguistic support for the declarative description of the fault tolerance software aspects at different development stages, nor to promote some lexicon of graphic terms that would facilitate the graphical representation of relations among software entities of each stage. Our primary motivation is to relieve the software designer from the burden of refining by hand the abstract software specifications in order to obtain a concrete system, and our objective is to provide a formal framework for the specification of fault tolerance software properties, which fulfills the following criteria:

1. It is precise enough so as to allow the accurate expression of fault tolerant properties, while at the same time it should be generic enough so as not to restrict the independence of the algorithmic software aspects.
2. It is rich enough so as to allow the deduction of properties equivalence or compatibility.
3. It supports means that guarantee the refinement correctness which represents a transition from a software development stage to the next one. It should also be possible to

use these means in the reverse sense, to identify the abstract specifications from which a given property is derived.

The remainder of this paper is structured as follows: in the next section we present a layered decomposition of the software development process that provides an insight from different viewpoints on the analysis of the fault tolerance software properties. In Section 3 we define the formal framework that supports the specification of fault tolerance software properties, the reasoning on the correctness of their refinements, and the deduction of the properties of their compositions. In Section 4 we propose a classification schema based on the previously introduced formal framework, which organizes the fault tolerance properties in a way that allows their systematic and efficient retrieval during a specification refinement process. The practical use of the classification schema is discussed in Section 5. A comparison with related work is presented in Section 6, and the paper concludes with a brief summary, a discussion on the originality of our contribution, and a presentation of current work and open issues.

2 Software Development Stages

Traditionally, the software development procedure has been decomposed into the following stages: requirements, specifications, design, coding, testing, and maintenance. In this section we introduce a different decomposition that is better suited for the analysis of the fault tolerance software properties. The proposed decomposition reflects the successive refinements of the abstract and macroscopic view on the fault tolerance software properties, into a microscopic and elaborated description of the software elements that accomplish a given fault tolerant functionality. Notice that the proposed development decomposition is complementary to the traditional one rather than an alternative, and it permits the designer to focus on the fault tolerance aspects of software development.

Macroscopically, the development of fault tolerant software starts by specifying the semantics of the states that should be reached after the occurrence of a failure. These semantics refer in general to the relation of the state after the failure, with some correct state that can be reached by the software. For example, the ***Safety*** semantics can be defined as the constraint that the state after the failure should be a subset of some state preceding the failure (i.e. the functionality of the system might degrade but all failure effects are removed). In a similar way, the semantics of ***Availability*** and ***Reliability*** can be respectively defined as reaching a state that can be reached by some failure-free execution of the software (i.e. failure effects are repaired), and reaching a state that includes a correct state (i.e. after the failure there still exists a part of the system that correctly provides the complete software functionality). The formalization of these semantics is given in the next section, in the upper part of Table 1.

The macroscopic specifications of failure semantics should be refined to provide more information about the properties of the reached state. For example, the designer has to distinguish among the re-initialization of the system, the removal of the part of the system affected by the failure, and the replacement of the affected part by a correct one. This refinement of failure semantics at the second development stage, leads to constraints that give a strict definition of what state should be reached. Based on this definition, the designer has to specify *how* to reach that state. A new refinement takes place, that leads to the third development stage. In this stage, the system is viewed as a set of objects, and the set of object states forms a *partition* of the system state, i.e. there is no state sharing between any two objects, and the union of all object states gives the system state. Objects interact by performing I/O actions as defined by the CSP computation model. The goal of the refinement leading to this state is to decompose system state into sub-states that can be mapped onto objects and to identify the object actions that lead to a system state satisfying the predefined fault tolerance constraints.

The next step in the development of fault tolerant software is to outline its abstract architecture. Borrowing the concepts from the field of software architectures, the designer needs to specify the system as a set of *components* interconnected by *connectors*. The object states and actions that have been previously identified, must be associated to components and connectors of the software architecture in order to express their fault tolerance properties. The result is an abstract description of a fault tolerant software architecture. Elaborating on the fault tolerance properties assigned to components and connectors will result in another refinement leading to the decomposition of the architectural entities into more concrete ones that can be directly mapped onto the prevalent objects of a fault tolerant mechanism (e.g. the reliable broadcast protocol, the replication manager, the voting mechanism, etc).

The elaborated description of a fault tolerant software architecture reveals the software properties related to fault tolerance aspects of a system, but leaves undefined a number of parameters indispensable for the configuration, deployment and correct functioning of the corresponding fault tolerant mechanism. Such parameters include the degree of replication for a given failure probability of the software and hardware constituents of a system, and the tuning of the timeout periods for specific network characteristics and specific load-patterns. These parameters have to be determined during the probabilistic analysis of the software that takes place in a separate development stage. The final software development stage consists of writing the code that corresponds to the specification resulting from the previous stages (for software created from the scratch), or assembling the existing pieces of code that satisfy the specification constraints (for software reuse). In the remainder of this paper, we do not address issues related to the last two software development aspects.

3 Formalizing Fault Tolerance Software Properties

To be beneficial for the designer, the decomposition of the software development process presented in the previous section must be underpinned by a formal framework, which should satisfy the following conditions: it should be easy to use, and it should be expressive enough to capture a big majority of the properties related to failure semantics of the software. We have chosen to use predicate logic extended with the *precedence* binary operator $\prec$ (originally introduced in [10]), which defines a partial order in which predicates are verified. The extended predicate logic provides the designer with the means to combine the constraints on the system states that should be reached after a failure with the partial order of actions that should be performed to reach the given state. Hence, the designer obtains a global description of system's fault tolerance properties, both in terms of state semantics after a failure and of actions undertaken to reach that state. This description forms a blueprint of the fault tolerance mechanism that should be used and can be integrated in the software architecture of the system in a way similar to the one described in [8].

To express the fault tolerance software properties, it suffices to define the base predicates that formally describe system states and actions with respect to failures. We bring to the reader's attention that this set of base predicates is not unique; the designer is free to choose the base predicates that facilitate his reasoning. In the remainder of the paper we use the following computation model: a system state is a mapping of variables to values according to the software specifications. The value of some variables can be undefined in the specifications, but when the values of one or more variables lay outside the range given in the specifications, a failure is said to occur. The system state is *partitioned* by the states of the objects consisting the system, and state transitions are the result of the actions performed by the objects. To denote predicates on states, actions and their interplay, the following notations are used: ST, primed or followed by a subscript value, denotes the system state; lower-case greek letters denote objects (e.g. α, β, etc), and object states are denoted by the object name (which can be neglected when it is obvious in a given context) followed by ST, primed or followed by a subscript value (e.g. $\alpha.ST$, $\beta.ST_i$, etc). System and object actions are written in italics (e.g. $import(\alpha, \beta, data)$), and the corresponding predicate verified when the action is realized is written in small-capitals (e.g. $\textsc{import}(\alpha, \beta, data)$). Moreover, when not otherwise stated, we write ST to denote the object state before executing a given action, and ST' to denote the state reached after the execution of the given action. Based on these notations, we define the following predicates:

- $[ST]$, which is true when the system is in state ST. Similarly we define $[\alpha.ST]$ for object states.
- $faulty(F)$, $F \subseteq ST$, which is true when some of the variables of F have been assigned values not defined by system's specifications. Similarly we define $faulty(\alpha.F)$ for object failures.
- $\textsc{init}(\alpha, A)$, which is true when the object α is appears for the first time in the system, with initial state A (i.e. $\nexists\alpha.ST|[\alpha.ST] \prec \textsc{init}(\alpha, A)$).
- $\textsc{exit}(\alpha)$, which is true when the object α is removed from the system, (i.e. $\nexists\alpha.ST|\ \textsc{exit}(\alpha) \prec [\alpha.ST]$).
- $\textsc{export}(\alpha, \beta, data)$, which is true when the object α exports to object β the information $data$, where $data$ is a function of the object state before executing the *export* action, i.e. $data = f(ST)$ for some function f defined in the system specifications.

- IMPORT($\alpha, \beta, data$), which is true when the object β receives the information *data* sent by object α, where the state ST' of the object after executing the *import* action is a function of the previous state and of the parameter *data*, i.e. $ST' = g(ST, data)$ for some function g defined in the system specifications.

- FAIL(α, *action*), which is true when the state ST' reached after the execution of *action* by α verifies the predicate $faulty(ST')$.

Using the above predicates, we can give the abstract specifications and their refinements of the *Safety*, *Availability*, and *Reliability* semantics given informally in the previous section, as depicted in the upper- and the middle-parts of Table 1 respectively. In these formulas, the notation $\mathcal{X}^C$ denotes a correct system execution, i.e. a partially ordered set of states where the predicate $faulty(ST)$ is never verified, F represents the sub-state of faulty mappings (i.e. $F = \{m \in ST_i : faulty(\{m\})\}$), $dom(ST)$ denotes the variables of state ST (i.e. the *domain* of the mapping), and symbol $\setminus$ denotes the operation of set subtraction. The lower-part of Table 1 introduces specification of fault tolerance properties based on action predicates, which are the result of the analysis done in the third software development stage. In these formulas, we assume a failure occurring during the transition from state ST to state ST', and we use the notation ST_0 to denote the initialization state of the object (i.e. $\nexists \alpha.ST : [\alpha.ST] \prec [\alpha.ST_0]$), and ST'' to denote some object state occurring after state ST' (i.e. $[ST'] \prec [ST'']$).

Safety	$\equiv$	$([ST_i] \wedge \ faulty(ST_i)) \Rightarrow$
		$(\exists ST_j, ST_c : ([ST_i] \prec [ST_j]) \wedge ([ST_c] \prec [ST_i]) \wedge (ST_j \subseteq ST_c))$
Availability	$\equiv$	$([ST_i] \wedge \ faulty(ST_i)) \Rightarrow (\exists ST_j : ([ST_i] \prec [ST_j]) \wedge (ST_j \in \mathcal{X}^C))$
Reliability	$\equiv$	$([ST_i] \wedge \ faulty(ST_i)) \Rightarrow (\exists ST_j : ([ST_i] \prec [ST_j]) \wedge$
		$(\exists ST_c \in \mathcal{X}^C : (\forall ST : [ST] \prec [ST_i] \Rightarrow [ST] \prec [ST_c])) \wedge (ST_c \subseteq ST_j))$
Clean-up	$\equiv$	*Safety* $\wedge (ST_j = ST_i \setminus F)$
Crash	$\equiv$	*Safety* $\wedge (ST_j = \emptyset)$
Re-initialize	$\equiv$	*Availability* $\wedge (ST_j = ST_0, where \ \nexists ST : [ST] \prec [ST_0])$
Rollback	$\equiv$	*Availability* $\wedge (\exists ST_c : ([ST_c] \prec [ST_i]) \wedge (ST_c = ST_j))$
Roll-Forward	$\equiv$	*Reliability* $\wedge (ST_c = ST_j)$
Replacement	$\equiv$	*Reliability* $\wedge (dom(F) \subseteq ST_j \setminus ST_c)$
DOCLEAN-UP	$\equiv$	*Clean-up* $\wedge \forall \alpha \mid faulty(\alpha) \Rightarrow$ EXIT(α)
ATOMIC	$\equiv$	*Rollback* $\wedge \forall \alpha, action \mid$ FAIL($\alpha, action$) $\Rightarrow (ST'' = ST)$
DOREPLACE	$\equiv$	*Replacement* $\wedge \forall \alpha \mid (faulty(\alpha) \prec$ INIT($\alpha', \alpha.ST_{c'}$)$) \wedge (\alpha' \equiv \alpha) \wedge$
		$([\alpha.ST_{c'}] \prec [\alpha.ST])$

Table 1: Failure semantics related to Safety, Availability, and Reliability.

Given the aforementioned framework for the formal expression of the fault tolerance software properties, the designer can reason on their compatibility, and on their combinations. By combining different properties, the designer obtains a refinement of their initial constraints, since the combination can be seen as the superposition of new properties on the existing constraints related to fault tolerance. These refinements, once proved to be correct, can be used any number of times to indicate to the designer a set of correct refinements for a given fault tolerance software property. We elaborate on this utilization of the formal framework in the next section.

4 Refining Fault Tolerant Software Architectures

The field of existing fault tolerance techniques keeps expanding continuously, and their proliferation is most unlikely to be ceased, since it is instigated both by the advances in the areas of hardware and operating system support, and by the increasing application requirements in tolerating failures. Given the immense number of fault tolerant mechanisms and their complexity, it is indispensable, from the standpoint of software correctness and robustness, to reach the final software development stage with a clear and detailed description of the fault tolerance software properties. For this, we need to provide the means that will guide the designer in refining the abstract constraints related to fault tolerance software aspects into an elaborated description of properties of the fault tolerant mechanism that should be used.

4.1 Classifying Fault Tolerance Properties

One way to provide the desired guidance for the designer, is to couple a given fault tolerance property with a set of corresponding alternative refinements. For example, the lower-part of Table 1 shows some possible refinements of the failure semantics given in its middle-part, which, in turn, are some of the possible refinement of the failure semantics given in the upper-part. In general, we can associate to some fault tolerance property P, the set of fault tolerance properties $\{Q_i | Q_i \Rightarrow P\}$, i.e. the set of P's alternative refinements. This organization results in a classification of fault tolerance properties, which permits the designer to consider only correct refinements of some abstract software constraints regarding fault tolerance. Existing work (e.g. see [12]) suggests that once the refinement's correctness is verified, the refinement patterns represented by the classification schema can be used without needing to prove their correctness. This suggestion is valid in the case of fault tolerant software architectures, as long as the refinement process does not consider constraints related to other nonfunctional or algorithmic software aspects.

Given the above observation, the classification schema can serve as a structured representation of the knowledge about correct refinements of fault tolerance properties that has been obtained in the past. We have adopted the straight forward implementation of the classification schema as a disconnected directed acyclic graph (DAG), where a node represents a fault tolerance property, its parents represent the abstractions for which the property represented by the given node is a correct refinement, and its children represent the known correct refinements of the given fault tolerance property. Figure 1 illustrates graphically a part of the disconnected DAG that corresponds to the failure semantics given in Table 1.

There is a trivial (although not very efficient) construction of a software tool that manipulates such a classification schema. Based on the results of the process that verifies the correctness of a given refinement, the tool can insert a new fault tolerance property into the classification schema. Moreover, given a fault tolerance property, the tool may retrieve both its refinements or the abstractions for which the given property forms a correct refinement.

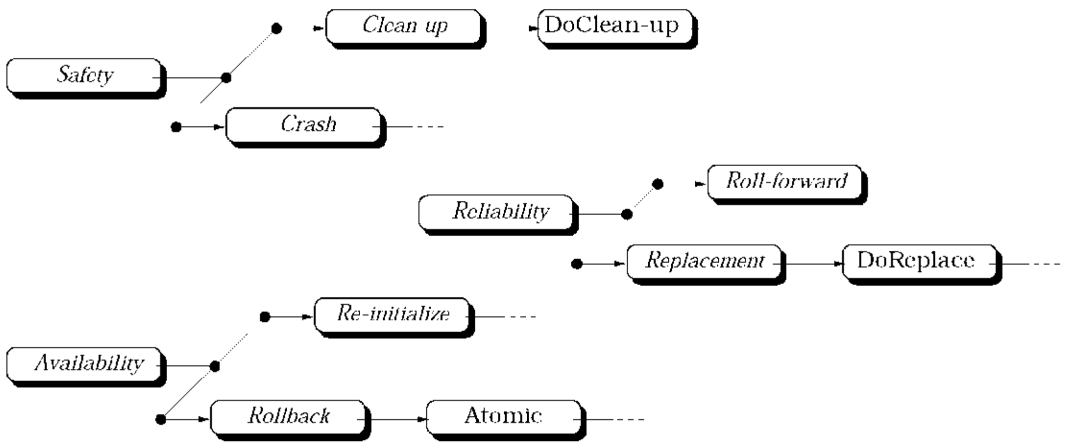

Figure 1: The DAG corresponding to the specification given in Table 1.

The refinements captured in the classification schema, represent a gradation from the abstract specifications of failure semantics to concrete descriptions of fault tolerance properties, which can be detailed enough so as to serve as precise specifications of a fault tolerant mechanisms. Hence, it is possible to associate pointers to DAG nodes that point into a repository of fault tolerant mechanism, and then use the classification schema as guidance means for the selection of a fault tolerant mechanism as the last step of a refinement process. We bring to the reader's attention that the same fault tolerant mechanism can be classified under more than one DAG node, since it might provide more than one fault tolerance property. Consider for example the case of an implementation of ISIS [3], which satisfies all three predicate ABCAST, CBCAST, and GBCAST given in Table 2. The corresponding mechanism should not be registered only as implementation of one of the three fault tolerance properties.

The classification schema provides a considerable aid in indicating possible choices for the appropriate fault tolerant mechanism with respect to some fault tolerance requirements. However, this issue in its globularity is far from considered as settled. The number of incarnations that may correspond to a single fault tolerance property can be more than one, and still the fault tolerant mechanisms may have radically different attributes. The differences can be due to factors influenced by other software aspects, irrelevant to failure semantics (e.g. synchronization characteristics). Hence, the reader should consider that the classification schema can be employed to help the designer reject those fault tolerant mechanisms that are not related in any way with the software failure requirements, rather than as a tool for selecting a set of fault tolerance mechanisms conforming with software requirements.

4.2 Iterative Architecture Refinement

Until this point, we have dealt with the analysis of fault tolerance properties as a domain specific software architecture, independently from the other nonfunctional or algorithmic software aspects. This is useful for the construction of a fault tolerant mechanism, in which case however, we can no longer consider the fault tolerance properties as nonfunctional software aspect, but rather as its algorithmic aspects. While the utility of studying the construction of fault tolerant mechanisms is incontestable, our research was fostered by the need to support the analysis of fault tolerance properties as a nonfunctional aspect of a distributed application. Hence, besides the capability to analyze and to express fault tolerance properties, we need to be sure that the proposed formal framework allows the designer to consider the conjunction of failure semantics with the constraints placed by other nonfunctional and algorithmic software aspects. The fact that the proposed framework is based on predicate logic, which has been successfully used for program specification, suggests an *a priori* compatibility with the expressions of general program specifications (e.g. see [11]). This implies that it is possible to express the constraints related to various software aspects in the same formal framework with little effort. Our experience so far has shown that software properties related to fault tolerance [13], and security [2] can be uniformly expressed. Moreover, we have not encountered any problems yet with transactional properties, although our experience in this domain is still preliminary (e.g. see [8]).

The advantage of expressing the algorithmic and the various nonfunctional software properties on a unified basis, is that we can reason on their combinations and detect possible incompatibilities among the constraints related to different software aspects. For example, let us consider the case of a software architecture that has defined all aspects but those related to fault tolerance, and that failure considerations should be introduced, without altering the constraints placed by the rest of the software aspects. The designer can start with some initial failure constraints and verify their compatibility with existing software properties. Once the compatibility is verified, the designer, using the proposed classification schema, may choose some refinement of the failure constraints. While the correctness of the refinement with respect to the fault tolerance properties is not necessary, the compatibility of the refinement results with the existing software properties must be checked again. Notice however, that deducing that a combination of nonfunctional properties is free from incompatibilities does not imply that we obtain a description of how to accomplish the combination. This is another issue that lies out of the topics discussed in this paper.

The procedure described above is iteratively employed until the point where the designer obtains an elaborated software architecture that describes globally the various software aspects. The invariant that should be verified in all iteration steps, is that the set of constraints should remain coherent, i.e. there are no conflicting constraints in the various software aspects. If an incompatibility is detected during some iteration, then the designer is called to choose another alternative of possible refinement. In the case where incompatibilities raise for all alternative refinements registered in the classification schema, the designer has to decide either to remove some of the constraints related to other software aspects, or to expand the classification schema by introducing new refinements of the problematic property.

In the general case, we expect that the refinement of software aspects will be done in parallel and hence, in the case of incompatibilities, the designer will have more than one choice on what refinement to change. Figure 2 illustrates graphically the iterative refinement process. When all software properties are refined in parallel, there must be some priority that will define which software aspects are most significant and hence should be the last to change. Otherwise, the designer will be confronted with a large variety of choices without any indication on which should be preferred.

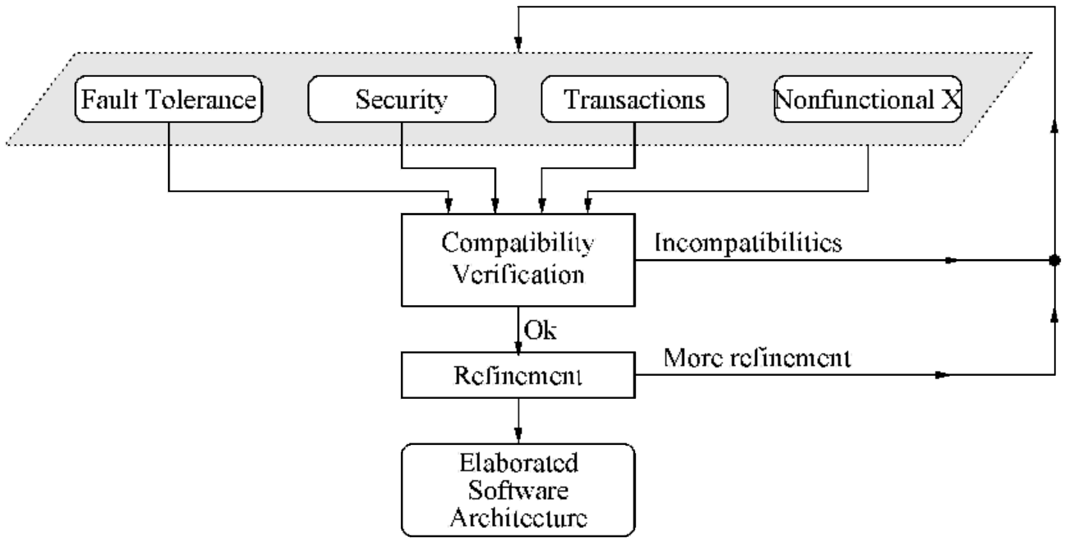

Figure 2: The iterative refinement process that leads to the elaborated software architecture description.

5 An Example

To demonstrate how the designer operates in the various software development stages, we consider the example of a file system, focusing primarily on the specification of the fault tolerance properties. The first two stages should define what are the state that should be reached after a failure occurrence and what are the permissible intermediate states, i.e. what should be done and what should be prevented. If erroneous data caused by the failure must be eliminated, the designer may specify that after the failure the system reaches a state that is a subset of some state preceding the failure, and become more precise by stating that this state should be reached by removing the erroneous data, or by re-initializing the file contents. The actions that lead the file system after a failure to a state where the files affected by a failure are inaccessible, would correspond to a rejection of importing requests referring to operations on these files.

To exemplify the utility of the formal documentation of failure semantics, let us consider the case of a file system providing only the operation *read(fn, buf)* that places in buffer *buf* the contents of the file *fn*. A first architectural description of the system's structure may consist of two components, representing the client and the server of the file system, interconnected by a connector employing an RPC communication protocol. To satisfy certain reliability constraints with respect to client accesses to files, we have to express those constraints in the description of the architectural entities that interact with the one representing the client (i.e. the component representing the server and the connector binding the client to the server). We assume that the designer wishes to build the system in such a way that a failure would neither prevent the client from reaching a correct state, nor degrade the system's functionality (e.g. by rendering inaccessible the *read* operation). Using the specifications given in Table 1, the aforementioned constraint corresponds to the *Replacement* refinement of the *Reliability* failure semantics.

Notice that the *Replacement* constraint does not have any impact on the graphical representation of the system's architecture, as given in the upper-part of Figure 3. Therefor, we need to refine the constraint in order for the fault tolerance properties to appear in the graphical architecture representation. Otherwise, it is not possible to assign the functionality related to fault tolerance properties, to concrete architectural entities, and hence to move towards a more elaborated and concrete descriptions of the file system. One possible refinement can be achieved by specifying that the server component described above is actually a set of replicas of the file server, and that the connector used for the communication between the client and the server diffuses a client request as a broadcast to the group of file servers. We can formally specify these fault tolerance properties, based on the abstractions used in the ISIS environment [3]. The formal expressions of the ISIS abstractions are given in Table 2, by the three predicates ABCAST, CBCAST, and GBCAST that represent respectively the *atomic* broadcast where messages are delivered in the the same order at all members of a group, the *causal* broadcast where causally related messages are delivered in their causal order, and *group* broadcast where ordinary messages are delivered atomically and messages indicating the failure of a group member are delivered after the delivery of all messages sent by the failed member[1].

The predicates defined in Table 2, can be used in the file system example to describe the broadcast of a message (predicate BCAST), the group of replicas (predicate REPLICA(M)),

[1]For simplicity, we do not consider the recovery of group members, which is treated by the GBCAST [3].

$\textsc{membership}(\mathrm{M})$	$\equiv$	$((\alpha \in M) \wedge \textsc{import}(\gamma, \alpha, D)) \Rightarrow \forall \beta \in M \mid \textsc{import}(\gamma, \beta, D)$
$\textsc{bcast}(\alpha, G, D)$	$\equiv$	$\forall \beta \in G \mid \textsc{export}(\alpha, \beta, D)$
$\textsc{replica}(\mathrm{M})$	$\equiv$	$\textsc{membership}(\mathrm{M}) \wedge (\forall \alpha, \beta \in M \mid \alpha \equiv \beta)$
$\textsc{fmask}(\alpha)$	$\equiv$	$(\exists M \mid (\alpha \in M) \wedge \textsc{replica}(\mathrm{M}) \wedge \textsc{membership}(\mathrm{M}) \wedge (\forall \beta \in M \mid Crash))$
$\textsc{abcast}(\mathrm{M})$	$\equiv$	$(\textsc{membership}(\mathrm{M}) \wedge (\alpha \in M) \wedge (\textsc{import}(\gamma, \alpha, D) \prec \textsc{import}(\gamma', \alpha, D'))) \Rightarrow$
		$\forall \beta \in M \mid (\textsc{import}(\gamma, \beta, D) \prec \textsc{import}(\gamma', \beta, D'))$
$\textsc{cbcast}(\mathrm{M})$	$\equiv$	$(\textsc{membership}(\mathrm{M}) \wedge (\textsc{bcast}(\alpha, M, D) \prec \textsc{bcast}(\beta, M, D'))) \Rightarrow$
		$\forall \gamma \in M \mid \textsc{import}(\alpha, \gamma, D) \prec \textsc{import}(\beta, \gamma, D')$
$\textsc{gbcast}(\mathrm{M})$	$\equiv$	$\textsc{abcast}(\mathrm{M}) \wedge ((\beta \in M) \wedge \textsc{bcast}(\alpha, M, f_\beta) \Rightarrow$
		$\forall \gamma \in M, \nexists D \mid \textsc{import}(\alpha, \gamma, f_\beta) \prec \textsc{import}(\beta, \gamma, D)),$
		$\mathit{where}\ \exists \alpha \neq \beta \mid \textsc{fail}(\beta, \mathit{action}) \prec \textsc{bcast}(\alpha, M, f_\beta)$

Table 2: Formal expressions of the abstractions used in ISIS.

and the *Failure Masking* property (predicate $\textsc{fmask}(\alpha)$). Informally, the *Failure Masking* property based on the replication of an object α is verified when there exists a group of objects identical to α that verify the MEMBERSHIP property (i.e. when one group member imports some data, then all group members import the same data), and that follow the *Crash* failure semantics defined in Table 1. Given these fault tolerance properties of the file server and the connector, the system description can no longer be graphically represented by the simple structure of the two boxes interconnected by a single line. Figure 3 illustrates graphically the results of the refinement.

6 Related Work

The need to master the conceptual complexity stemming from the interference of the algorithmic software aspects with the fault tolerance properties, has fostered the research towards the identification of abstractions that simplify the analysis and design of fault tolerant software. Some of the most significant results are reflected in [5], where a small set of intelligible concepts is used to analyze the hierarchical software construction and to express the relevance of fault tolerance properties to the entities of different hierarchy levels. This approach can be supported by the formal specifications of failure semantics in a variation of Hoare's logic, introduced by the same author in [4]. The occurrence of failures considered in the software specifications are tolerated, in the sense that the resulting system behavior is specified. Our work builds on these results in two ways: first we adopt the consideration of failures as part of the software specifications, and second we use an extension of the predicate logic to formally describe the fault tolerance software properties at each stage of its development. However, we go one step further in allowing the description of fault tolerance properties at different abstraction levels and in supporting the refinement mappings of abstract levels to concrete ones.

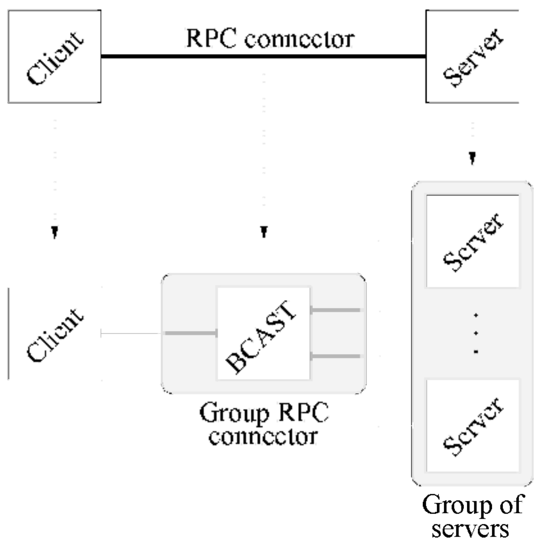

Figure 3: The results of the specifications refinement reflected in the graphical representation of the corresponding software architecture.

Another important research direction followed in defining abstractions for modeling fault tolerance software properties, is the one of specifying a fixed set of failure semantics which thereafter can be used as design patterns for the construction of a certain range of fault tolerant software. A representative example is the work presented in [14], where a set of constraints on object execution is given, concerning the *Agreement*, *Order*, *Failure Detection*, and *Stable storage* abstractions. These abstractions capture the important software properties and suppress the irrelevant details, and can be used to map the specifications of the failure behaviors onto some concrete fault tolerant mechanisms. This is very close to the mapping of an abstract software architecture on executable code of a computing system. However, the use of these abstractions presumes that the previous software development stages were accomplished successfully and that the designer knows exactly what failure semantics to employ in order to meet the initial constraints of the software. Compared to our work, similar approaches lack the flexibility to easily adapt in changes to their initial assumptions caused by the evolution of the fault tolerance requirements on the software.

Except from the explicit specification of fault tolerance software properties, the research results in areas coping with the more general issue of refining abstract specifications can be used in the fault tolerance domain. These research results (e.g. see [11]), assert that the correct specifications refinement must be underpinned by a formal basis which should allow the designer to reason on the composition and the equivalence of the software properties based on the rules of *composition* [6] and *specification matching* techniques (e.g. see [16]). The refinement process can be supported by some given refinement patterns that are proved to be correct (e.g. see [12]). However, the refinement patterns cannot be used for the fault tolerance software properties independently from the other software aspects, since the refinement correctness must be proved with respect to the constraints regarding the entire set of software aspects. As a result, refinement patterns specific to fault tolerance properties can be used only in cases where the software system conforms with some given failure semantics (e.g. the four-layer RCP architecture [15] can be used for developing fault tolerant software based on the semantics of the *State Machine Approach*).

Software architectures that express the relations of the architectural entities consisting a software can also be employed for analyzing the fault tolerance software properties. Existing work focuses on the analysis of the coordination and synchronization of software entities [1], and on the description of the allocation function that assigns functionality to the entities of a software architecture [9]. Our work is complementary to these research directions, since it permits to integrate the fault tolerance properties within the description of a software architecture. Our experience towards this direction has shown that it is possible to describe the nonfunctional software properties in general, without sacrificing the clarity of the architectural description [8].

We bring to the reader's attention that the related work cited in this section is only a representative sample of the research results published in numerous conferences and journals. The goal was to show that existing work either does not separate the algorithmic from the nonfunctional software aspects, or when it does, it does not provide means combine the constraints stemming from the various software aspects and to reason on their correctness. By this we are not advocating that directions followed in existing work are wrong or incomplete. On the contrary, we believe that there is a tremendous evolution on the analysis and the comprehension of different stages of software development and from different viewpoints. Thus, our goal is to benefit from the research results in the aforementioned areas in order to provide a framework that allows the designer to survey easily and safely the development procedure of fault tolerant software.

7 Conclusions

In this paper we presented a formal framework that supports the development and the refinement of fault tolerant software architectures. We have used this framework for the formal documentation of software properties related to fault tolerance, and based on the specifications refinements and their correctness derived from its use, we have proposed a classification schema for organizing fault tolerance software properties. The classification schema is used to indicate to the designer a number of possible correct refinements for a given fault tolerance software property. We did not aim at providing some linguistic or graphical support for the expression of the fault tolerance software properties. Rather, we focused on the formal basis underpinning the refinement process of some abstract specifications into an elaborated description of a fault tolerant software architecture.

The originality of our contribution is that we provide the designer with means to reason on the composition, refinement and equivalence of fault tolerance software properties at different stage of software development. We build on our previous work [7], where, based on predicate logic and specification matching techniques, we reason on the equivalence and the compatibility of software properties in general. Our current research departs from the point in which we realized that the incorporation of fault tolerance requirements in a predefined application structure leads to several inconsistencies among the constraints placed by various software aspects. Based on this observation, we reached the conclusion that the selection of a fault tolerant mechanism with appropriate semantics should be guided by a classification schema in order to be both correct and efficient [13]. The mathematical basis proposed in this paper provides the means to construct such a classification schema, and to keep enriching it with newly specified fault tolerance properties. In addition, the formal basis permits the integration of our work with existing approaches on reasoning on the algorithmic software constraints, that are based on predicate logic.

In parallel with specifying fault tolerance properties and introducing them into the classification schema, we also focus on the security software aspects [2], where we are facing the problem of systematically combining security constraints with fault tolerance properties. In some specific cases, the priority between those two nonfunctional aspects can be deduced by the correctness of their combination in a given order (e.g. confidentiality constraints should be satisfied before prior to the satisfaction of constraints related to communication reliability). However, we are not aware of any methods that provide systematic rules for combining these two nonfunctional aspects in a wide spectrum of possible scenarios. It is in our intentions for the near future, to look more closely into the combinations of fault tolerance software properties with other nonfunctional and algorithmic software aspects. We believe that this research direction is of great interest not only for the Aster project, but more generally for the construction of fault tolerant software architectures.

References

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[2] C. Bidan and V. Issarny. Security Benefits from Software Architecture. In *Proceedings of the 2nd International Conference COORDINATION'97*, pages 64–80, September 1997. Also available at http://www.irisa.fr/solidor/work/aster.html.

[3] K. P. Birman and T. A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[4] F. Cristian. A Rigorous Approach to Fault-Tolerant Programming. *IEEE Transactions on Software Engineering*, 11(1):23–31, January 1985.

[5] F. Cristian. Understanding Fault-Tolerant Distributed Systems. *Communication of the ACM*, 34(2):56–78, February 1991.

[6] C. A. R. Hoare. An Axiomatic Approach to Computer Programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[7] V. Issarny and C. Bidan. Aster: A Framework for Sound Customization of Distributed Runtime Systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 586–593, May 1996. Also available at http://www.irisa.fr/solidor/work/aster.html.

[8] V. Issarny, C. Bidan, and T. Saridakis. Characterizing Coordination Architectures According to Their Non-Functional Execution Properties. In *Proceedings of the 31st Hawaii International Conference on System Science*, pages 275–283, January 1998. Also available at http://www.irisa.fr/solidor/work/aster.html.

[9] R. Kazman, L. Bass, G. Abowd, and M. Webb. SAAM: A Method for Analyzing the Properties of Software Architectures. In *Proceedings of the 16th International Conference on Software Engineering*, pages 81–90, 1994.

[10] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[11] C. Morgan. *Programming from Specifications.* Series in Computer Science. Prentice Hall International, 1990.

[12] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.

[13] T. Saridakis and V. Issarny. Towards Formal Reasoning on Failure Behaviors. In *Proceedings of the 2nd European Research Seminar on Advances in Distributed Systems*, March 1997. Also available at http://www.irisa.fr/solidor/work/aster.html.

[14] F. B. Schneider. Abstractions for Fault Tolerance in Distributed Systems. Technical Report TR 86-745, Department of Computer Science – Cornell University, Ithaca, New York 14853, April 1986.

[15] B. L. Di Vito and R. W. Butler. Provable Transient Recovery for Frame-Based, Fault-Tolerant Computing Systems. In *Proceedings of the 13th IEEE Symposium on Real Time Systems*, pages 275–278, 1992.

[16] A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 6–17, October 1995.

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399