Developing Dependable Systems Using Software Architecture

Titos Saridakis & Valérie Issarny

INRIA/IRISA, Campus de Beaulieu, 35042 Rennes Cédex, France
{saridaki, Valerie.Issarny}@irisa.fr

Abstract: The construction of dependable software systems is recognized as a complex task: the system developer has to address the usage of fault tolerance techniques in addition to the design of the functional aspects that are specific to the system. This paper proposes a framework aimed at easing the development of dependable systems by providing software designers with a repository of dependable software architectures. A dependable software architecture shows how to integrate a fault tolerance technique with a given system so as to make the system dependable. Furthermore, the dependability behaviors of architectures are formally specified, which allows to unambiguously interpreting the various fault tolerance techniques as well as to organize the repository of corresponding architectures into a refinement-based lattice structure.

Key words: Dependability, formal specification, software architecture, software reuse, specification refinement.

1. INTRODUCTION

Making a system dependable is recognized as a complex task. In addition to the treatment of functional aspects that are system-specific, the system's designer has to cope with the integration of the fault tolerant mechanisms that satisfy the system's dependability requirements. However, the field of dependability has reached a sufficient level of maturity to capture its various ramifications. In particular, there exist a significant number of fault tolerant mechanisms to handle various dependability needs over different system platforms. Thus, there is an *a priori* knowledge of the mechanisms that are eligible to make a system dependable with respect to the system's

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35563-4_35](https://doi.org/10.1007/978-0-387-35563-4_35)

dependability requirements and underlying platform. Furthermore, the understanding of fault-tolerance mechanisms and associated abstractions enables a separation of concerns in system design by addressing independently the design regarding functional and dependability aspects. In that context, we propose a framework for making a system dependable through the reuse of appropriate fault tolerance abstractions.

Our work builds on results of the software architecture field (Perry and Wolf 1992, Shaw and Garlan 1996). A system's software architecture abstractly describes the system's gross organization in terms of components (i.e., units of computation) and connectors (i.e., units of interaction). This allows the practical use of formal methods to define the behaviors of components and connectors, and to carry out complementary system analyses. Our framework for the construction of dependable systems consists of characterizing dependable software architectures that are generic with respect to the base functional architectural elements (i.e., functional components and connectors among them). The dependability behaviors of the architectures are further defined formally, which enables their unambiguous interpretation, as well as to organize the set of dependable architectures according to a refinement relation over their behavior. Practically, the developer is provided with a repository of dependable architectural patterns from which he may select the one that meets the dependability requirements of his system. Ultimately, the fault tolerance constituents of a dependable architecture may correspond to implemented mechanisms. Such mechanisms can be directly integrated with the system's functional structure according to the structure shown by the dependable architecture.

This paper is organized as follows. Section 2 details our approach to the formal specification of dependability behaviors. Section 3 introduces our framework for making systems dependable, presenting a repository of dependable software architectures. Finally, we conclude in Section 4, summarizing our contribution and comparing it with related work.

2. FORMAL SPECIFICATION OF DEPENDABILITY BEHAVIOR

To be practically beneficial for software development, a formal framework should satisfy two conditions:

1. It should be easy to understand and use.
2. It should be expressive enough to capture all (or at least a big majority) of the targeted properties (i.e., properties relating to dependability in our case).

Both these conditions are satisfied by an extension of predicate logic with the *precedence* relation (Lamport 1978) (binary operator « $<$ ») specifying a partial order in which predicates are verified. Based on the precedence relation, we define the relations *eventually* (unary operator « $\Diamond$ ») and *in the past* (unary operator « ∇ ») which denote that a predicate will be verified in the future or was verified in the past. The extended predicate logic provides comprehensible and easy to employ means for combining the constraints on system states that should be reached after a failure with the partial order of actions that should be performed to reach these states. Notice that the use of temporal logic relations is not indispensable for modeling the temporal precedence of the predicate. Indeed, means have been invented like *history lists*, which are employed by a number of approaches (e.g., see (Chrysanthis and Ramamritham 1994) and (Stoller and Schneider 1996)) in order to avoid temporal relations for ordering the occurrences of events in a system and to remain purely first order logic. However, we use them because we believe that they render the formulas more legible. The formal framework we use is presented hereafter, followed by our approach to the specification of system behaviors with respect to dependability, introducing the specification of dependability properties and a refinement relation over them.

2.1 Formal framework

A *system* is a set of variables, which can be assigned different values according to the system specifications. A *state* of the system is a mapping of variables to values, where the values of some variables can be undefined. When the values of one or more variables lay outside the range defined in system's specifications, a *failure* is said to occur. An *execution* of a system is a partially ordered set of system states, where one state in the set is distinguished as being the initial state (i.e., the state preceding all other states in the set). An *object*¹ of the system is an entity having some state. Hence, a system can be seen as a set of objects. An *action* is a state transition, which can be caused by some internal object computations, or by some I/O operation. Actions are associated with objects and we assume deterministic actions, i.e., given the specifications of an object, an object state and an action on that state, the resulting state after performing the action is uniquely defined. However, we do not constrain the choice of the next action to be performed, which can be a random choice among different alternatives. Hence, although actions are deterministic, the execution of an object and, consequently, the execution of the entire system are non-deterministic. In

¹ The term object signifies a logical entity and not entities specific to programming languages (e.g. C++ objects).

this context, an *event* is the execution of some actions or the reach of some state. In the remainder of this document, we use the following notations:

- Objects are denoted by the first five lower-case Greek letters, primed or followed by a subscript value (e.g., α , β_i , etc).
- σ , primed or followed by a subscript value, denotes a system state. For object states, we prefix the object name (e.g., $\alpha.\sigma$). We neglect the object name when it is obvious in a given context.
- Σ denotes system specifications. For object specifications, we prefix the object name (e.g., $\alpha.\Sigma$).
- X denotes a system execution which is a partially order set of system states. When followed by the superscript C , it denotes a failure-free execution.
- Actions are written in lower-case italics followed by a list of arguments in parentheses. To distinguish among actions of different objects, we prefix the object name to the action (e.g., $\alpha.\text{import}(\beta, \text{data})$). We neglect the object name when it is obvious in a given context.

The structural elements of the system model presented above do not suffice to describe the properties of a specific system (i.e., the relations among constituent objects, their interactions, their invariants, and their constraints). For this, a set of predicates is needed to capture the essential properties of system entities. This set of predicates should be minimal in order to be easy to use and understand. We present below a set of predicates that capture the fact that the system is in a given state, and the execution of I/O actions. Notice that this set of predicates is not unique; another set of predicates, richer or more frugal, can be chosen if it facilitates the system programmer's reasoning (e.g., additional predicates that can be defined are *init*, *exit*, *begin*, *commit*, and *abort*, to describe the actions related to object initialization and termination, or the actions related to transactional properties). In the remainder of this document, we use the following predicates:

- The predicate expressing that a system is in state σ is introduced by the unary operator $[]$, i.e., $[\sigma]$ is true when the system is in state σ . Similarly $[\alpha.\sigma]$ is verified when object α is in state σ .
- The predicate *export* expressing the I/O action performed by an object α when it sends to an object β the data d . The data d sent, are a function of some α 's state preceding the I/O action, and if some β 's state is a function of data d , the export action precedes this state. More formally, the predicate is defined as:

$$\text{export}(\alpha, \beta, d) \equiv (\exists \alpha.\sigma : \forall [\alpha.\sigma] \wedge d = f(\alpha.\sigma)) \wedge (\beta.\sigma = g(d) \Rightarrow \diamond [\beta.\sigma])$$
- The predicate *import* expressing the I/O action performed by an object β when it receives the data d sent by object α . The export of data d

from α to β preceded, and some state of β after receiving d is a function of the received data. More formally, the predicate is defined as:

$$\text{import}(\alpha, \beta, d) \equiv \nabla \text{export}(\alpha, \beta, d) \wedge (\exists \beta.\sigma : \beta.\sigma = g(d) \wedge \Diamond[\beta.\sigma])$$

Notice that the predicates *export* and *import* correspond to the $\alpha.\text{export}(\beta, d)$ and $\beta.\text{import}(\alpha, d)$ actions respectively.

2.2 Dependability properties

Given the above formal framework, we are able to define dependability properties, which serve to characterize dependability behaviors of software architectures. Let us point out here that the dependability properties introduced in the following, reflect more the authors' perspective on the issue rather than a widely acknowledged characterization. Alternative specifications of dependability behaviors can be envisioned. In the same way, there may exist alternative interpretations for the terms we use to qualify the dependability properties. Here, we base our work on the fault tolerance perspective introduced by Laprie (Laprie 1992).

The important point we want to make with respect to our approach to the specification of dependability properties is that it enables us to characterize the various behaviors of a system in the presence of failure, which are attainable using existing fault tolerance techniques. The set of these behaviors may further be expanded as new fault tolerance techniques emerge. In the remainder, we present the specification of some representative dependability properties so as to give the reader, the intuition of how dependability properties are characterized in general. Basically, dependability properties fall into two groups:

1. *Abstract properties* specified in terms of system states, which are defined independently of any fault tolerance technique. They serve to characterize the dependability behavior of an overall architecture, when this behavior is too abstract to associate a specific fault tolerance technique with it.
2. *Concrete properties* specified in terms of system actions, whose definition is closely related to some fault tolerance technique. They serve to characterize the dependability behaviors associated with architectural elements, with respect to a given fault tolerance technique.

Let us first give abstract properties defined at the state level. The most abstract dependability property, simply qualified as *Dependability*, ensures that a system makes progress despite the occurrence of failure. The *Safety* property defines that, after a failure, the system should enter an error-free state, which is some subset of a state, reached before the occurrence of the failure. The basic characteristic of this abstraction is the removal of the

failure products. The specification of the *Availability* property indicates that the state reached after a failure is a state contained in some failure-free system execution. The basic characteristic of this abstraction is the repair of failure effects. Another specification is the *Reliability* property, which defines that the state reached after a failure includes a state that should have been reached in the absence of failures. The basic characteristic of this property is the transition to the expected state despite the occurrence of failures. More specific dependability properties are the ones of *Detection* and *Fmask* where the former characterizes failure detection and the latter the system capability to mask the occurrence of failures. Let *faulty* be the predicate expressing that a system state contains an erroneous mapping of variables to values, i.e., *faulty*(σ) is true when some of the variables of σ have been assigned values not defined by system's specifications. Similarly *faulty*($\alpha.\sigma$) is verified when the object state $\alpha.\sigma$ contains an erroneous mapping from variables to values.

Table 1: The formal specifications of some dependability properties.

Dependability(S) $\equiv ([\sigma] \wedge \text{faulty}(\sigma)) \Rightarrow \exists \sigma' \in \Sigma : [\sigma] < [\sigma']$
Safety(S) $\equiv ([\sigma] \wedge \text{faulty}(\sigma)) \Rightarrow (\exists \sigma', \sigma'' \in \Sigma : ([\sigma] < [\sigma']) \wedge ([\sigma''] < [\sigma]) \wedge (\sigma' \subseteq \sigma''))$
Availability(S) $\equiv ([\sigma] \wedge \text{faulty}(\sigma)) \Rightarrow (\exists \sigma' \in \Sigma : ([\sigma] < [\sigma']) \wedge (\sigma' \in X^C))$
Reliability(S) $\equiv ([\sigma] \wedge \text{faulty}(\sigma)) \Rightarrow (\exists \sigma', \sigma'' \in \Sigma : ([\sigma] < [\sigma']) \wedge (\sigma'' \in X^C) \wedge (\forall \sigma_p : [\sigma_p] < [\sigma] \Rightarrow [\sigma_p] < [\sigma'']) \wedge (\sigma'' \subseteq \sigma'))$
Detection(S) $\equiv [\sigma] \wedge \text{faulty}(\sigma)$
Fmask(S) $\equiv \forall \sigma \in \Sigma : (\exists \sigma_f : \text{faulty}(\sigma_f) \wedge ([\sigma] < [\sigma_f]) \Rightarrow (\exists \sigma' \in \Sigma : ([\sigma_f] < [\sigma']) \wedge (\sigma \subseteq \sigma')))$
DetectionObj(α) $\equiv (\exists \beta : [\beta.\sigma] \wedge \text{faulty}(\beta.\sigma)) \Rightarrow [\beta.\sigma] < \text{export}(\alpha, \varepsilon, f(\beta\text{-failed}))$
FmaskObj(α) $\equiv ([\alpha.\sigma] \wedge \text{faulty}(\alpha.\sigma)) \Rightarrow (\exists \beta : \beta.\Sigma \approx \alpha.\Sigma \wedge [\alpha.\sigma] < [\beta.\sigma] \wedge \neg \text{faulty}(\beta.\sigma) \wedge \exists \alpha.\sigma' : ([\alpha.\sigma'] < [\alpha.\sigma]) \Rightarrow (\exists \beta.\sigma' : (\beta.\sigma' = \alpha.\sigma' \wedge [\beta.\sigma'] < [\beta.\sigma])))$

The upper part of Table 1 gives the specifications of the aforementioned dependability properties, for a system S . The properties in the upper part of Table 1 characterize only the system state that is reached after a failure

occurrence. They do not make explicit the system objects that are involved in fault treatment nor the needed interactions among them. This is captured by concrete properties, defined at the action level. For instance, the *Detection* and *Fmask* properties may be respectively revised into the specification of *DetectionObj* and *FmaskObj*. The specification of the former expresses the fact that a system object transmits a message to some other object in the system, after a failure occurred. This message contains the information of the occurred failure, which implies that the transmitting object captures this knowledge in its state. Similarly, the specification of the latter expresses the fact that for a failed object, there exists an equivalent object (not necessarily a different one) which reaches a correct state that follows all the failed object's states preceding the failure. In other words, this means that the state that would have been reached by a given object in the absence of failures, is eventually reached even if a failure occurs on the object in question.

The formal expressions that describe the aforementioned properties are given in the lower part of Table 1. Notice that the interaction events are expressed by the *export* and *import* predicates, and their parameters define the exact interaction pattern between the two objects indicated by the predicate parameters. Object ε is used to signify any object of the environment. In addition, the equivalence of object specifications, noted $\equiv_s$, is defined with respect to the observable behavior of objects, i.e., the specifications of two objects are equivalent if the sequences of *import()* and *export()* actions performed by the objects are equivalent.

As more concrete examples, let us consider the enforcement of dependability for an object, using a replication technique. Achieving replication consists of replicating an object into a group of objects and making the group behave as a single object from the perspective of the group's environment. The behavior of the objects group may differ depending on the replication technique (i.e., active, semi-active, passive) that is used. The formulas of Table 2 characterize the dependability properties for the active and passive replication techniques, where $id(d)$ uniquely identifies the data d among all the data exchanged in the system. The id function is defined so that $id(d) = id(d')$ if d and d' are exported by objects having equivalent specifications and the export actions correspond in the sequences of the I/O actions performed by the objects.

2.3 Refinement relation

Based on the proposed approach to the specification of dependability properties, we are able to define a refinement relation over these properties. This relation allows refining an initial dependability requirement into more

Table 2: Formal specification of active and passive replication

$\text{Active}(\alpha, N) \equiv \exists \alpha_1, \dots, \alpha_{N-1} : G = \{\alpha, \alpha_1, \dots, \alpha_{N-1}\} \wedge \text{Replication}(G) \wedge \text{Filter}(G) \wedge \text{AtomicDelivery}(G)$
$\text{Replication}(G = \{\alpha_i\}_{i=1..N}) \equiv \forall \alpha_i, \alpha_j \in G : ((\alpha_i.\Sigma \equiv_s \alpha_j.\Sigma) \wedge \neg(\text{faulty}(\alpha_i.\Sigma) \Rightarrow \text{faulty}(\alpha_j.\Sigma)))$
$\text{Filter}(G = \{\alpha_i\}_{i=1..N}) \equiv \exists \beta : (((\alpha_i, \alpha_j \in G) \wedge \text{import}(\alpha_i, \beta, d_i) \wedge \text{import}(\alpha_j, \beta, d_j) \wedge (\text{id}(d_i) = \text{id}(d_j))) \Rightarrow (\exists! \text{export}(\beta, \varepsilon, d_0) \wedge (\text{id}(d_0) = \text{id}(d_i) = \text{id}(d_j))))$
$\text{AtomicDelivery}(G = \{\alpha_i\}_{i=1..N}) \equiv (\exists \alpha \in G : \text{import}(\varepsilon, \alpha, d) \Rightarrow (\forall \alpha_i \in G : \text{import}(\varepsilon, \alpha_i, d))) \wedge (\exists \alpha \in G : (\text{import}(\varepsilon, \alpha, d_1) < \text{import}(\varepsilon, \alpha, d_2)) \Rightarrow (\forall \alpha_i \in G : (\text{import}(\varepsilon, \alpha_i, d_1) < \text{import}(\varepsilon, \alpha_i, d_2))))$
$\text{Passive}(\alpha) \equiv \exists \gamma, \beta : \text{Replication}(\{\alpha, \beta\}) \wedge \text{StableStorage}(\alpha, \gamma) \wedge \text{Restore}(\alpha, \beta, \gamma)$
$\text{StableStorage}(\alpha, \gamma) \equiv \text{import}(\alpha, \gamma, f) \wedge (f = \alpha.\sigma) \wedge ([\alpha.\sigma] < \text{export}(\alpha, \gamma, f)) \wedge \neg(\exists \gamma.\sigma : \text{faulty}(\gamma.\sigma))$
$\text{Restore}(\alpha, \beta, \gamma) \equiv (\exists \alpha.\sigma' : ([\alpha.\sigma'] < [\alpha.\sigma]) \wedge \text{import}(\gamma, \beta, f) \wedge (f = \alpha.\sigma')) \wedge (\forall \varepsilon, d : ([\alpha.\sigma] < \text{export}(\varepsilon, \alpha, d)) \Rightarrow \text{import}(\varepsilon, \beta, d))$

concrete dependability properties, which ultimately correspond to the behavior of fault tolerance mechanisms for which an implementation is available. Considering two dependability properties $P_1(S)$ and $P_2(S)$, the latter is a refinement of the former if $P_2(S) \Rightarrow P_1(S)$. For illustration, Figure 1 depicts the refinement relation that holds over the dependability properties introduced in the previous subsection. In the figure, each property P is represented by a box that contains a set of boxes to denote alternative correct refinements of P , and each of these sub-boxes points towards a set of properties whose conjunction is a correct refinement of P .

3. REPOSITORY OF DEPENDABLE SOFTWARE ARCHITECTURES

The proposed specification of dependability properties provides means to unambiguously describe the dependability behavior of an architecture, but it is of limited help from the standpoint of easing the development of dependable systems. To facilitate their use, we propose to attach to each

dependability property, the structure (i.e., the software architecture) of the corresponding system with respect to the fault tolerance technique that is used to enforce the given property.

Figure1: Some refinements of the dependability property

The refinement relation over dependability properties provides the adequate base ground to organize the repository of dependable software architectures. The repository is organized as a lattice structure defined according to the refinement relation, and each node stores the acquired knowledge about a given dependability property. For some property P , this knowledge includes: (i) the property name, (ii) the formal specification of

the dependability property, (iii) the set of dependability properties (through references to adequate nodes) into which P may be refined, and (iv) the dependable software architecture A_P , associated with P .

The repository may be depicted in a way similar to the graph given in Figure 1 except that each node now embeds the description of the *dependable software architecture* corresponding to the property defined by the node. The dependable architecture corresponding to an abstract property is a black-box component embedding the system since the property is too abstract to have a fault tolerance technique associated with it. On the other hand, the architecture defined for a concrete property exposes the system's structure with respect to some fault tolerance technique. The following subsection further elaborates on the description of dependable architectures, which, as shown in Subsection 3.2, may be derived from the specification of dependability properties. Subsections 3.3 and 3.4 then introduce the main functions used for the management of the architecture repository; they relate to the introduction and retrieval of a dependable architecture with respect to a given property.

Prior to detailing the description of dependable software architectures, let us note that we concentrate here on the definition of architectures with respect to the fault tolerance technique that is used to enforce a given dependability property. The proposed architectural description may be enriched when there is an available mechanism to implement the embedded fault tolerance technique. For instance, the architectural definition could then include the specification of the component's interaction protocol (e.g., using Wright (Allen and Garlan 1997)) and of the component's functional interface. In the same way, the definition of connectors could be introduced so as to detail the interaction protocol used by the mechanism. In general, the description of a dependable software architecture includes at least the specification of the dependability behavior of its components, and may be extended using the capabilities of existing ADLs (Architecture Description Languages). In particular, a dependable architecture may be defined using ACME (Garlan *et al.* 1997) so as to exploit different ADLs and thus allow various architecture analyses.

3.1 Dependable software architecture

To be helpful to system developers, the description of dependable architectures must make clear how to compose a dependable system from a base system. The components of a dependable architecture may be of either of the two following kinds: *Generic*, in which case the component corresponds to the initial system that is to be made dependable, or *Dependable*, in which case the component is specifically introduced for

enforcing some dependability behavior. Then, given a software architecture providing some concrete dependability property, a system can be integrated with the corresponding fault tolerance technique by mapping the system onto the generic components. We propose the following description for dependable architectures

Dependable Architecture: *Name* =

Dependability:

-- Architecture's dependability property --

Components:

{ *Component Name*: *TypeComp*: -- Dependability behavior -- } +

Configuration:

-- Description of a configuration through bindings among components --

where the specifications of dependability behaviors and properties are expressed according to our approach discussed in the previous section.

Table 3: Architectural descriptions associated with the Replication, Filter, and AtomicDelivery properties

<p>Dependable Architecture : Replication = Dependability : Replication(G); Components : G[i: 1..N] : Generic: Replication(G); Configuration : nil ;</p>
<p>Dependable Architecture : AtomicDelivery = Dependability : AtomicDelivery(G); Components : G[i: 1..N]: Generic : (i:1..N) : $(import(\epsilon, G(i), d) \Rightarrow \forall j \in [1, N] : import(\epsilon, G(j), d))$ $\wedge ((import(\epsilon, G(i), d) < import(\epsilon, G(i), d')) \Rightarrow$ $(\forall j \in [1, N] : import(\epsilon, G(j), d) < import(\epsilon, G(j), d')));$ Configuration : (i: 1..N) : AtomicDelivery.Import to G(i).Import;</p>
<p>Dependable Architecture : Filter = Dependability : Filter(G); Components : G[i: 1..N] : Generic : TRUE ; $F : Dependable : (i, j \in [1, N] \wedge import(G(i), F, d) \wedge$ $import(G(j), F, d') \wedge (id(d) = id(d')) \Rightarrow$ $(\exists ! export(F, \epsilon, d'') : (id(d'') = id(d))));$ Configuration : (i: 1..N) : G(i).Export to F.Import; F.Export to Filter.Export;</p>

A dependability behavior may simply be *TRUE* if there is no dependability requirement associated with the architectural element. The type of a component identifies whether the component is generic or dependable. We further assume that each architectural component (including the architecture itself) has an **Import** and an **Export** port. For illustration, Table 3 gives the descriptions of the architectures associated with the *Replication*, *Filter*, and *AtomicDelivery* properties. Let us remark that the proposed architectural descriptions expose only structural information regarding fault tolerance. In particular, only bindings dedicated to fault tolerance are characterized.

Considering the proposed description of dependable architectures, a system *S* may be modified so as to enforce a given dependability property *P* by mapping *S* onto each generic component of the architecture associated with *P* while ensuring the declared dependability behavior, and providing an adequate implementation for the dependability-specific components. Alternatively, the repository of dependable architectures may further be exploited to find out more refined architectures, which possibly correspond to available fault tolerance mechanisms.

3.2 Deriving dependable architectures from properties specifications

Ideally, one would like to have a systematic way to derive the structure of a dependable architecture from its associated formal specification. Although not direct, the proposed specification of dependability properties embeds the needed information. Let us take a close look at dependability properties. From a property specification, we are able to infer:

1. the objects involved in the enforcement of the property, which are all the objects appearing in the specification
2. the objects' behaviors with respect to dependability, which are given by part of the specification that refers to the object
3. the needed interactions among objects, which are given by part of the specification expressed in terms of *import* and *export* predicates.

To systematically infer the above information and hence a dependable architecture, from a property specification, we propose to structure the specification of dependability properties accordingly. For *ObjectType* stating whether the object is generic or not, and parameters *VarName* being of type integer, Table 4 gives the form of the specifications of a property *P*, followed by an illustration of its employment using as an example the *Filter* property.

Table 4. The form of property specification and an example

$P(\text{objects } \{ \text{ObjectName} : \text{ObjectType} \}^*; \text{Ind } \{ \text{VarName} \}^+) \equiv$ objects : $\{ \text{ObjectName} : \text{ObjectType} \}^+$ behaviors : $\{ \text{ObjectName} : \text{-- formula --} \}^+$ configuration : -- formula -- ; Filter ($G : \text{Generic}[1, N]$) $\equiv$ objects : $F : \text{Dependable}$; behaviors : $(i : 1..N) : G(i) : \text{TRUE}$; $F : (\text{import}(G(i), F, d) \wedge \text{import}(G(j), F, d') \wedge \text{id}(d) = \text{id}(d')) \Rightarrow$ $(\exists ! \text{export}(F, \epsilon, d'') : (\text{id}(d'') = \text{id}(d)))$; configuration : $((i : 1..N) : \text{import}(G(i), F, d)) \wedge \text{export}(F, \epsilon, d')$;

Intuitively, we can infer from the specification of the *Filter* property that the corresponding dependable architecture is made of the set of generic components $G(i)$ and of the dependable component F . In addition, the formula given in the **configuration** part enables to deduce interaction among components based on the semantics of the *import* and *export* predicates: *import*(α, β, d) as well as *export*(α, β, d) implies that the **Export** port of α is bound to the **Import** port of β . We further recall that ϵ is used to signify any object of the environment. Thus, *import*(α, ϵ, d) (resp. *import*(ϵ, α, d)) signifies that the **Export** (resp. **Import**) port of α is bound to the architecture's **Import** (resp. **Export**) port. The same applies for the *export* predicate. Precisely, the inference of the logical formula and of the software architecture corresponding to a given dependability property is achieved as follows. Let P be defined as:

$$\begin{aligned}
 P(\text{objects } O_i, 1 \leq i \leq n; \text{var } \dots) \equiv \\
 \text{objects} : O'_i, 1 \leq i \leq n'; \\
 \text{behaviors} : O_i : B_i, 1 \leq i \leq m; \\
 \text{configuration} : B;
 \end{aligned}$$

The corresponding logical formula is equivalent to: $\exists O_1, \dots, O_n, \exists O'_1, \dots, O'_n : (B \wedge (\bigwedge_{i=1..m} B_i))$

Let us remark here that the proposed specification of properties may lead to extend the original specifications. This is exemplified by the new definition of *Filter*, which extends the original one with the formula stated in the configuration part. As another example, let us consider the *AtomicDelivery* property. The embedded formula $\exists \alpha \in G : \text{import}(\epsilon, \alpha, d) \Rightarrow (\forall \alpha_i \in G : \text{import}(\epsilon, \alpha_i, d))$ relates to the behavior of the α s. It also relates to the architecture's configuration: all the α s are accessible by objects of the environment. Thus, this formula must appear in two parts of the

property specification. However, the formula for configuration is simplified into $\forall \alpha_i \in G : \text{import}(\varepsilon, \alpha_i, d)$. In general, we do not see the required modification of property specification as a major drawback given the resulting benefit for the production of architectural descriptions.

Let us now examine the inference of the architecture associated with $P()$. It consists of defining the interpretation of each constituent of the property specification in terms of architectural description. The treatment of the **objects** and **behaviors** parts of the specification is direct: each object given in the **objects** lists translates into an architectural component whose type (i.e., dependable or generic) is the one declared in the embedding list; and each object behavior given in **behaviors** is attached to the corresponding architectural component. The interpretation of the **configuration** part is less direct, it requires to interpret each element of the corresponding logical formula. Precisely, a formula defining a configuration is of the form: $\bigwedge_i P_i$ where each P_i is expressed as either an *import* or an *export* predicate, whose parameters may possibly be universally quantified. Thus, each P_i is translated into bindings among components according to the parameters of the *import* or *export* predicates.

3.3 Updating the repository

Updating the architecture repository requires providing functions for the addition and removal of dependability properties. However, since the treatment of the latter is quite straightforward, we address only the former in the following. The introduction of a dependability property P leads to insert the corresponding node N within the repository, according to the refinement relation over properties.

Inserting a property: Let us use the following notations:

- P denotes the set of dependability properties.
- N denotes the set of nodes of the repository.
- $\text{Prop}(N)$ is the function that returns the property defined by node N .
- $\text{Anc}_N(P)$ denotes the set of immediate ancestor nodes of N , with respect to the dependability property P .
- Dec_N denotes the set of immediate successor nodes of N .
- $\text{POW}(X)$ denotes the power-set of X .

Let us first consider the introduction of a property P refining a property of the repository (i.e., P needs not to be conjuncted with another property). For instance, if we consider Figure 1, P may be *Reliability* but not *Fmask*, which has to be conjuncted with *Detection* to be a refinement of an existing property. Given our assumption, the node N for property P must be

introduced within the repository in a way that guarantees the following two conditions:

$$\mathbf{C1}(P, \text{Anc}_N(P)) = \forall N' \in \text{Anc}_N(P) : (P \Rightarrow \text{Prop}(N')) \wedge \neg(\exists N'' \in N - \{N'\} : (P \Rightarrow \text{Prop}(N'') \Rightarrow \text{Prop}(N)))$$

$$\mathbf{C2}(P, \text{Dec}_N) = \forall N' \in \text{Dec}_N : ((\text{Prop}(N') \Rightarrow P) \wedge \neg(\exists N'' \in N - \{N'\} : (\text{Prop}(N') \Rightarrow \text{Prop}(N'') \Rightarrow P)))$$

Let us now consider the introduction of a property P that refines an existing one, when conjuncted with a set of complementary properties. We require all these properties to be inserted in the repository at once, using the following *Insert* function. Given a set of properties $\{P_i\}_{i=1..n}$ to insert and the current nodes of the repository, the function returns the ancestor nodes that are common to all the N_i s defining the P_i s, with respect to the property $\bigwedge_{i=1..n} P_i$, and the set of successor nodes for each N_i :

$$\text{Insert} : \text{POW}(P) \times \text{POW}(N) \rightarrow P \times \text{POW}(\text{POW}(P))$$

$$\begin{aligned} \text{Insert}(\{P_i\}_{i=1..n}, N) = & (\bigcap_{i=1..n} \text{Anc}_{N_i}(\bigwedge_{j=1..n} P_j), \{\text{Dec}_{N_i}\}_{i=1..n}) \text{ if} \\ & \mathbf{C1}(\bigwedge_{i=1..n} P_i, \bigcap_{i=1..n} \text{Anc}_{N_i}(\bigwedge_{j=1..n} P_j)) \text{ and} \\ & \forall i \in [1, n] : \mathbf{C2}(P_i, \text{Dec}_{N_i}) \end{aligned}$$

When a node defining a concrete property P is created within the repository, the node should be completed with its corresponding architecture description. This is realized by inferring the architecture description from the property specification as discussed in the previous subsection.

Correct architecture refinement: Up to this point, we have seen that the introduction of a property within the repository is achieved according to the refinement relation over dependability properties. Let us consider two properties P_1 and P_2 such that P_2 refines P_1 . From the developer's standpoint, this means that the architecture A_2 associated with P_2 , may be safely used to enforce property P_1 . Let us now assume that the architecture A_1 associated with P_1 was originally selected to make a system dependable, but was later replaced by A_2 (e.g., such a replacement may be due to the availability of the mechanisms embedded by A_2). The replacement of A_1 by A_2 is practical only if both architectures have compatible structures, i.e., A_2 exposes the structure of A_1 's architectural elements. In this way, the later replacement of a dependable architecture by an architecture enforcing a stronger property does not impact on the design decision made so far. Thus, when a property P_2 refines a property P_1 , we require the architecture A_2

associated with P_2 , to be compatible with the architecture A_1 associated with P_1 . We say that A_2 is a *correct refinement* of A_1 (with respect to their architectural structures). Let us notice that in the case of architectures corresponding to available mechanisms, the refinement relation over architectures could additionally be constrained according to the definition of (Moriconi *et al.* 1995).

Let us first consider the simplest case that is when P_2 corresponds to a single node: the corresponding architecture A_2 is a correct refinement of an architecture A_1 , if P_2 refines the dependable property associated with A_1 , and if A_2 defines a set of sub-architectures that maps onto the components of A_1 . Let us use the following notations:

- An architecture A is defined by the triplet (P_A, C_A, B_A) .
- P_A denotes the dependability property of A .
- C_A denotes the components of A .
- $B_A = \{(C_b, C_i')\}_{i=1..n}$, $C_b, C_i' \in C_A$, defines the architectural bindings among A 's components.
- $Comp : POW(B) \rightarrow POW(C)$ is the function that returns the set of components embedded in a given set of bindings.
- A denotes the set of dependable architectures.
- $Beh : P \times C \rightarrow P$ is the function that returns the dependable behavior of a given component belonging to the specification of a given dependability property.

We introduce the following function to identify whether an architecture A_R is a correct refinement of an architecture A , with respect to the architectures' structures:

Refine : $A \times A \rightarrow BOOL$

Refine(A, A_R) = $\exists$ total function $M : C_A \rightarrow POW(B_{A_R})$ such that M is 1-to-1 and onto, **and**

$\forall C, C' \in C_A : (C \neq C' \wedge Comp(M(C)) \cap Comp(M(C')) = \emptyset)$, **and**
 $\forall C \in C_A : Dependability(P_{A_R}, M(C)) \Rightarrow Beh(P_A, C)$

Dependability gives the dependability behavior of the sub-architecture given by a set of bindings among components:

Dependability : $P \times POW(B) \rightarrow P$

Dependability(P, B) = $(\bigwedge_{C_i \in Comp(B)} Beh(P, C_i)) \wedge \bigwedge_{(C, C') \in B} (import(C, C', d) \Rightarrow export(C, C', d))$

Let us now consider the case where a conjunction of dependability properties P_i , $1 \leq i \leq N$, is introduced as a refinement of an existing property P . We must define the software architecture A that results from the

combination of the set of architectures A_i , $1 \leq i \leq N$, associated with each property P_i , and then verify that A is a correct refinement of the architecture associated with P , according to the definition of *Refine*. We have seen that the components of an architecture subdivide into *generic* and *dependable* components. Let us further recall that generic components correspond to the same functional component that is the software system to be made dependable. Henceforth, the generic components of the A_i s correspond to the same components. Thus, generic components are mapped onto the same components in the architecture A , and their dependable behavior is the conjunction of the behavior declared in each of the A_i s for generic components.

On the other hand, the dependable components of an architecture are in general specific to this architecture. Thus, the dependable components of A are the union of the dependable components of the A_i . However, there are two cases where dependable components of distinct architectures may have to be merged into a single component. One of these cases is exemplified by the architectures used to enforce *Passive* replication: the γ object is shared by the architectures enforcing *StableStorage* and *Restore*. In general, this case is detected through the definition of the conjunction of properties, which may explicitly share objects. The other situation where dependable components of distinct architectures may be merged is when there is a relation of logical implication between each pair of associated dependable behaviors. Here, we can keep only the dependable component that enforce the strongest dependability behavior among the set of components. So far, we have stated how to infer the set of generic and dependable components of an architecture resulting from the composition of some architectures. The set of bindings among these components are further the ones that are specified for the corresponding components within the A_i s.

3.4 Using the repository

Using the architecture repository for the construction of a dependable system consists of retrieving the software architecture associated with the dependability property that is targeted for the system. Let $\perp$ be the undefined node. The retrieval function is defined as $Retrieve : P \rightarrow N \cup \perp$ with:

$$\begin{aligned} Retrieve(P) = N \quad & \text{if } (N \in N) \wedge (Prop(N) \Rightarrow P) \wedge \neg(\exists N' \in N : \\ & (Prop(N) \Rightarrow Prop(N') \Rightarrow P)), \text{ or} \\ \perp \quad & \text{if } \neg(\exists N \in N : (Prop(N) \Rightarrow P)) \end{aligned}$$

The node N returned by the *Retrieve* function allows us to identify all the dependable architectures that are eligible to make a system dependable with

respect to the given dependability property. These architectures are all the architectures defined by the nodes of the sub-lattice whose root is N . Some of the eligible architectures may possibly be combinations of architectures when properties of the sub-lattice are refined into a conjunction of properties. Architecture combination is achieved according to the approach discussed in the previous subsection. Given eligible architectures, it is up to the system developer to select the one that is the most appropriate for the system. Several factors may influence the selection process. Among the most prominent factors, we foresee the existence of implementation for all or part of the dependable components embedded in the architectures. At this time, the selection of the most appropriate dependable architectures among the set of eligible ones is left upon the system developer.

We are currently examining solutions to help the developer in the selection process by coupling the architecture repository with an implementation repository. The benefit of our proposal for the construction of dependable systems lies in providing a repository of dependable architectures whose behaviors are precisely characterized using temporal first order logic. This characterization allows: (i) to infer an architectural description from a property specification, (ii) to retrieve an architecture providing the dependability property targeted for a given system, and (iii) to use an architecture selected from the repository to know how to extend a base non-dependable system with appropriate fault tolerance mechanisms. However, we cannot expect system developers to carry out the proofs appertained to the management of the repository of dependable architectures. Tools must be provided to assist this management. These tools include:

- A tool for the inference of a dependable architecture from the specification of a dependability property.
- A tool for updating the repository and retrieving architectures. This tool subdivides into a tool for classical database management, and a theorem prover for implementing the database functions that are defined over dependability properties.

We are currently implementing the first tool as well as the one relating to database management, their features are direct from the presentation we made in this paper. From the standpoint of providing a theorem prover, we are currently examining existing provers (e.g., (Manna *et al.* 1994)) so as to reuse an existing one for our framework.

4. CONCLUSIONS

This paper has presented a framework aimed at easing the construction of dependable systems. The framework relies on the formal specification of

dependability properties, using temporal first-order logic. The proposed specification of dependability properties allows to infer the dependable software architecture corresponding to a property, which characterizes the structure of a dependable system with respect to the fault tolerance technique enforcing the given property. The structure of a dependable architecture further makes clear how to compose a dependable system from a base system. Formal specification of dependability properties enables us to provide a repository of dependable architectures, which is organized according to the refinement relation holding over dependability properties. Our proposal relates to a number of research efforts of the software engineering domain. In particular, it builds on results in the area of architecture description languages, and of software reuse.

From the standpoint of existing ADLs, there have been many proposals based on formal techniques. However, these proposals aim at complementary goals to ours. For instance, objectives for ADLs based on formal techniques include comparison of architectural styles using the Z notation (Abowd *et al.* 1995), reasoning about interaction patterns of architectural styles using a CSP-based calculus (Allen 1997), comparison of architecture designs and proving properties with regard to a specific architecture using the chemical abstract machine model (Inveradi and Wolf 1995), verification of reconfiguration correctness of architectures using graph grammars (LeMetayer 1996), definition of executable prototypes for architectures using partially ordered set of events (Luckham *et al.* 1995), and correct stepwise refinement of architectures using first-order logic (Moriconi *et al.* 1995). The last reference appears to be the most closely related to our proposal. However, in this reference, the architectural refinement relates to preserving topological constraints of the architectural elements. On the other hand, we are concerned with characterizing the semantics of an architecture from the standpoint of provided dependability properties. This characterization further serves to provide developers with a repository of dependable architectures that show how to make a base system dependable, using a fault tolerance technique enforcing the targeted dependability.

There is a significant amount of work in the area of software reuse (Krueger 1992). In this subsection, we concentrate on two research efforts on this topic: systematic component retrieval, and software reuse for customizing execution environment. To our knowledge, systematic component retrieval based on a specification of components using first-order logic has firstly been experimented in the Inscape environment (Perry 1989). This environment belongs to the family of development environments that can be seen as ancestors of the ones based on ADL, i.e., applications are described using a module interconnection language which is roughly an

ADL without the connector notion. The Inscape environment demonstrated that it was feasible to use the specification of components in terms of pre- and post-conditions to guide complex system design but also to retrieve component implementations in a systematic way. Successors of this proposal then enhanced the practicality of systematic software retrieval. A software retrieval tool that may be used in any development environment is presented in (Rollins and Wing 1991). This capacity is further enhanced in (Zaremski and Wing 1997), which provides a framework to support the definition of various refinement relations. Efficiency of software retrieval is addressed in (Mili *et al.* 1997), which proposes to organize the software database according to a refinement relation over software specifications. This work and its more recent version (Jilani *et al.* 1997) supply, moreover, a retrieval function that returns a software component approaching a specification if there is no available component matching the requested specification. The proposal presented in (Schumann and Fischer 1997) also addresses efficiency of the software retrieval process; it consists of using rejection filters based on signature matching and model checking technology to rule out non-matching components as early as possible. Our proposal builds on the above results and applies them to the domain of retrieving a software architecture with respect to a requested dependability property instead of a functional one.

Customizing execution platforms so as to adapt to application needs is now a growing concern in the software engineering domain. This has led to the definition of notations to ease the development of customized systems using existing software. Examples of environments offering such a facility can be found in (Batory and O'Malley 1992, Hiltunen and Schlichting 1995, Struman and Agha 1994). These proposals differ from ours in that we are addressing customization of execution platforms based on the refinement of requested dependability properties, while they provide a way to construct such platforms based on its adequate structuring. Thus, these environments could be conveniently exploited in our framework to take over the construction of the dependable system after the selection of the adequate dependable architecture.

REFERENCES

- Abowd G. *et al.* (1995) Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319-364.
- Allen R. (1997) A Formal Approach to Software Architecture. PhD Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, USA.

- Allen R. and Garland D. (1997) A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213-249.
- Batory D. and O'Malley S. (1992) The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355-398.
- Chrysanthos P. and Ramamritham K. (1994) Synthesis of Extended Transaction Models using Acta. *ACM Transactions on Database Systems*, 19(3):450-491.
- Garlan D. et al. (1997) ACME: An Architecture Interchange Language. Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, USA.
- Hiltunen M. A. and Schlichting R.D. (1995) Constructing a Configurable Group RPC Service. *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*, pages 288-295.
- Inverardi P. and Wolf A. L. (1995) Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):373-386.
- Jilani L. L. et al. (1997) Retrieving Software Components that Minimize Adaptation Effort. *Proceedings of the IEEE International Conference on Automated Software Engineering*, pages 255-262.
- Krueger C. W. (1992) Software Reuse. *ACM Computing Surveys*, 24(2):131-183.
- Lamport L. (1978) Time, Clocks, and the Orderings of Events in a Distributed System. *Communications of the ACM*, 21(7):558-565.
- Laprie J. C. (1992) Dependability: Basic Concepts and Terminology. *Dependable Computing and Fault-Tolerant Systems*, Springer-Verlag.
- LeMetayer D. (1996) Software Architecture Styles as Graph Grammars. *Proceedings of the ACM SIGSOFT'96 Symposium on Foundations of Software Engineering*, pages 15-23.
- Luckham D. C. et al. (1995) Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336-355.
- Manna Z. et al. (1994) STeP: The Stanford Temporal Prover. Technical Report No.94-1518, Computer Science Department, Stanford University, Stanford, CA, USA.
- Mili R. et al. (1997) Storing and Retrieving Software Components: A Refinement Based System. *IEEE Transactions on Software Engineering*, 23(7):445-460.
- Moriconi M. et al. (1995) Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356-372.
- Perry D. E. (1989) The Inscape Environment. *Proceedings of the 11th International Conference on Software Engineering*, pages 2-12.
- Perry D. E. and Wolf A. L. (1992) Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40-52.
- Rollins E. J. and Wing J. M. (1991) Specifications as Search Keys for Software Libraries. *Proceedings of the 8th International Conference on Logic Programming*, pages 173-187.
- Schumann J. and Fischer B. (1997) NORA/HAMMR: Making Deduction-based Software Component Retrieval Practical. *Proceedings of the 12th IEEE International Conference on Automated Software Engineering*, pages 246-254.
- Shaw M. and Garland D. (1996) Software Architecture: Perspectives on an Emerging Disciplines. Prentice Hall.
- Stoller S. D. and Schneider F. B. (1996) Automated Analysis of Fault-Tolerance in Distributed Systems. Technical Report No.14853, Department of Computer Science, Cornell University, Ithaca, NY, USA.
- Sturman D. C. and Agha G. A. (1994) A Protocol Description Language for Customizing Failure Semantics. *Proceedings of the Thirteenth IEEE Symposium on Reliable Distributed Systems*, pages 148-157.

Zaremski A. M. and Wing J. M. (1997) Specification Matching of Software Components.
ACM Transactions on Software Engineering and Methodology, 6(4):333-369.