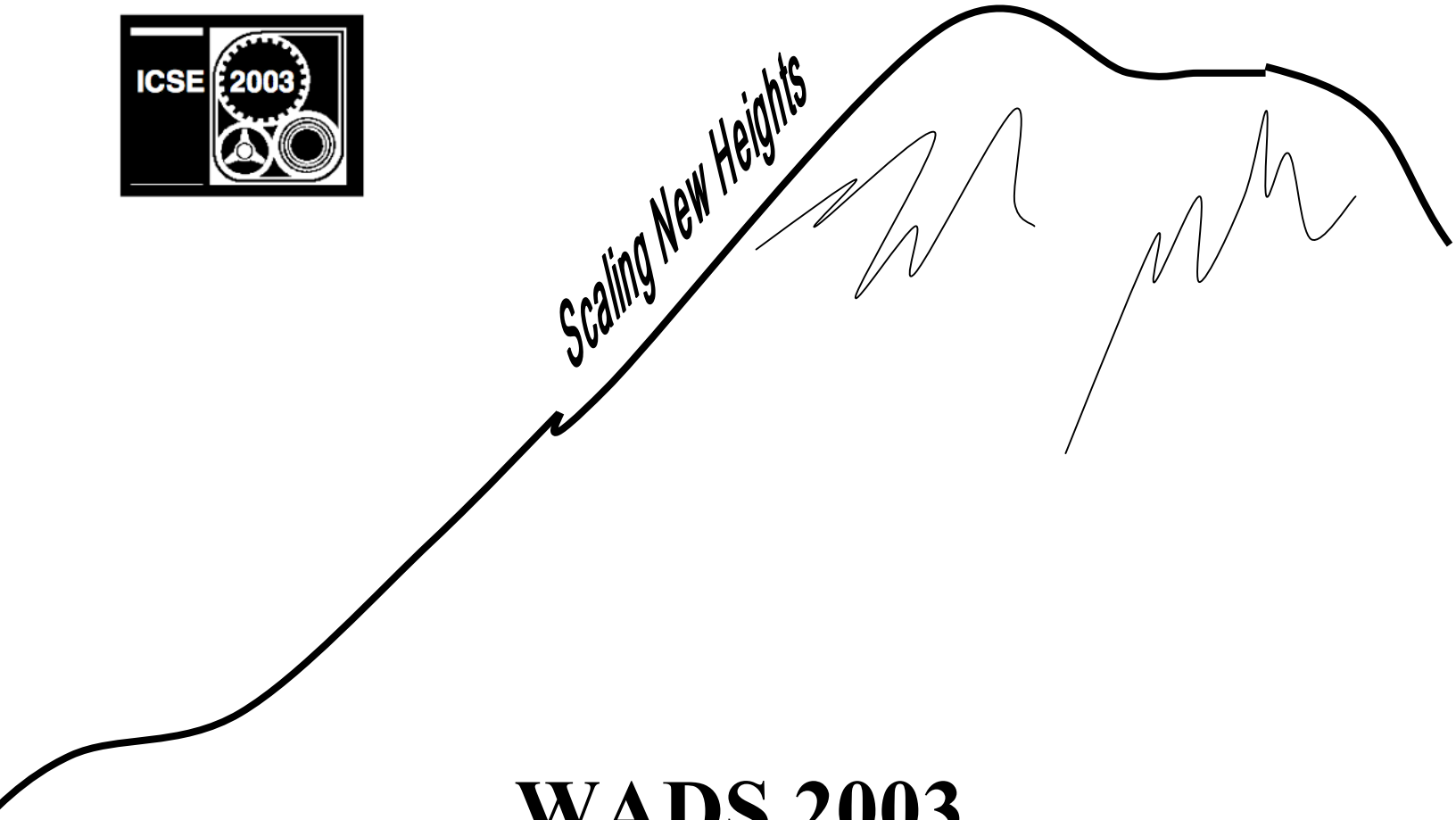

WADS 2003
Workshop on
Software Architectures
for Dependable Systems

ICSE'03
International Conference on Software Engineering
Portland, Oregon
May 3-11, 2003

Preface

Architectural representations of systems have shown to be effective in assisting the understanding of broader system concerns by abstracting away from details of the system. The dependability of systems is defined as the reliance that can justifiably be placed on the service the system delivers. Dependability has become an important aspect of computer systems since everyday life increasingly depends on software. Although there is a large body of research in dependability, architectural level reasoning about dependability is only just emerging as an important theme in software engineering. This is due to the fact that dependability concerns are usually left until too late in the process of development. In addition, the complexity of emerging applications and the trend of building trustworthy systems from existing, untrustworthy components are urging dependability concerns be considered at the architectural level. Hence the questions that the software architecture and dependability communities are currently facing: what are the architectural principles involved in building dependable systems? How should these architectures be evaluated?

By bringing together researchers from both the software architectures and the dependability communities, this workshop makes contributions from dependability more visible within the software engineering community and vice-versa, thus helping to build strong collaboration possibilities among the participants. The workshop provides software engineers with systematic and disciplined approaches for building dependable systems, as well as allows further dissemination of the state of the art methods and techniques.

During ICSE 2002 we organized the first workshop, which was a success (http://www.cs.ukc.ac.uk/wads/), and a LNCS volume has been edited that combines the state-of-the-art articles in the area. The aim of this Second Workshop on Software Architectures Dependable Systems is once again to bring together the communities of software architectures and dependability to discuss the state of research and practice when dealing with dependability issues at the architecture level.

We have received 20 submissions mainly from academic contributors. Each paper was reviewed by 3 members of the Program Committee, and a total of 14 papers have been accepted. We are thankful for the support and dedication of the Program Committee towards making this workshop a success. The Program Committee consisted of:

Jean Arlat, France
Andrea Bondavalli, Italy
Jan Bosch, The Netherlands
David Garlan, USA
Paola Inverardi, Italy
Valérie Issarny, France
Philip Koopman, USA
Nicole Levy, France
Nenad Medvidovic, USA
Dewayne E. Perry, USA
Debra Richardson, USA
Cecília Rubira, Brazil
William Scherlis, USA
Francis Tam, Finland
Kishor S. Trivedi, USA
Frank van der Linden, The Netherlands
Paulo Veríssimo, Portugal

We look forward to an interesting and stimulating workshop.

Rogério de Lemos, Cristina Gacek, and Alexander Romanovsky

Table of Contents

Perspective-based Architectural Approach for Dependable Systems

Sheldon X. Liang, J. Puett, Luqi
Software Engineering Automation Center
US Naval Postgraduate School
{xliang, jfpuett, luqi}@nps.navy.mil

Abstract

Explicitly architecting dependable systems inevitably involves establishing consensus among different stakeholders' concerns and then anchoring the design on architectural components that provide robustness. The goal is to architect evolvable systems upon which users can reasonably rely on receiving anticipated services. Unfortunately, there are few established approaches for rapidly prototyping architecture to identify dependable architectural components during the early stakeholder requirements resolution phases of software design. This paper presents a perspective-based architectural (PBA) approach process using rapid prototyping to build dependable architectures using compositional patterns. The approach is achieved through explicit architecting and system composition to provide a set of rules governing the system composition from coarser-grained dependable components. The approach provides a rationale for treating dependability as a set of semantic constraints localized on compositional patterns.

1. Introduction

Building dependability into the architectural design aims at attaining the benefits of reduced cost and increased quality. The central idea is that dependable architectures in large, complex, evolving systems will provide their users with a reasonable assurance that the system will deliver the services promised. Explicitly architecting such systems requires identifying and resolving different stakeholders' concerns. For instance, the architect may have to resolve the inherit conflicts between a user stakeholder that is concerned with achieving a particular computational requirement and an implementer stakeholder that may be concerned with achieving systematic long-term evolution of the system. Perspective-based architectural design [1-4] allows some resolution between these perspectives.

The difficulties in engineering software-intensive systems are further exacerbated by requirements uncertainty, dynamic organizational structures (and concerns), and the requirement for rapid application development. Engineering dependable systems involves three crucial aspects: 1) accurately identifying all customer requirements, 2) resolving customer requirement conflicts within the context of different customer perspectives, and 3) verifying that the resulting system satisfies customer intent (and if not, correcting the requirements and the system).

A number of techniques, frameworks, and approaches have emerged to address the problems in engineering software-intensive systems. Widely embraced efforts include rapid system prototypes [5-6], software architectures [7-11], and component techniques [12-14]; all of which focus on composing software systems from coarser-grained components. Rapid system prototyping is useful in effectively capturing and resolving uncertainty about requirements and providing computational visibility [6]. Component techniques assume a homogeneous architectural environment in which all components adhere to certain implementation constraints (e.g., design, packaging, and runtime constraints). They are unalterably associated with derivational implementation with little concern of the perspectives of the customer or architect [8-10]. Software architecture approaches typically separate computation (components) from interaction (connectors) in a system. However, the current level of understanding and support for connectors has been insufficient, so that connectors are often considered to be explicit at the level of architecture, but intangible in the system implementation [9-10]. Several sources have recommended the use of architecture views [1, 2]. Yet, while they provide guidance on how architecture should be represented, they generally do not provide a prototyping process for the early development of the artifacts that are used in that representation [3-4].

The rapid prototyping of architectural components shows promise in acquiring accurate and timely requirements and in establishing appropriate compartmentalization of functionality [2-4]. To reduce the amount of re-certification effort required after each requirement change, the approach presented in this paper helps to maintain the assurance of dependability as the system evolves by combining rapid prototyping with explicit architecting so that the system's architecture is based on properties that are invariant with respect to system requirements changes. This research integrates requirements validation techniques and stakeholder perspective resolution into a single model of explicit architectural composition.

2. Overview of the Approach

Fig. 1 depicts the PBA approach embodied in three perspectives: computational activity, compositional architecture and derivational implementation. Starting by rapid prototyping the user's informal needs, an initial prototyping model is created that represents the computational activity needed to implement the operational concept. Continued analysis and refinement of the prototype then derives the explicit architecture from which it is possible to extract valuable architectural properties. Compositional architecture is then explicitly built under the support of compositional patterns, and the generation of application framework is driven by both prototyping and architecting documentations. Next, PBA composers are applied to derive PBA components.

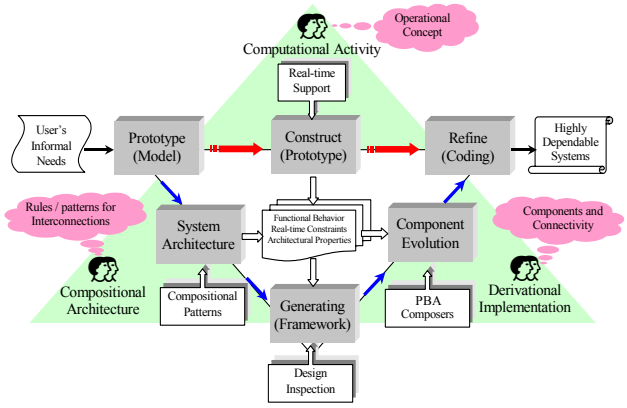

Fig 1 Synthesizing Approach

For each perspective design artifact, a computer-aided foundation is provided with significant formulated attributes enabling automated analysis, reasoning and code/framework generation. For instance, the computational activity captures the activities and information flows that will accomplish the operational concept (e.g., real-time support is the foundation for hard real-time systems [5]); the compositional architecture details what kinds of rules (patterns) are used to govern the interactions among components (e.g., compositional patterns [8,11] and design inspection [15-17] support semi-automated architecture generation); and the derivational implementation identifies physical components and connectivity that will be instantiated to carry out the computational activities (e.g., based on PBA composers [18]). Thus, compositional architecture bridges gaps between the computational and derivational artifacts (user and implementer perspectives).

3. Perspective-based Architecting

Central to the PBA approach, compositional patterns provide principles for guiding the design and evolution of system architecture and can be treated as architectural elements governing system composition from coarser-grained components. The transitional process is embodied in three perspective designs.

3.1 Computational activity

Computational activity accounts for the customer perspective concerns of computation and interconnection. This perspective addresses system requirements by capturing three kinds of formal arguments: components from which the system is built, interconnections enforcing interactions among them, and constraints on both components and interconnections:

$$\boldsymbol{P}_{\text{computation}} = [\boldsymbol{C}_c, \boldsymbol{I}, \boldsymbol{Ct}\,(\boldsymbol{C}_c, \boldsymbol{I})]$$

Where $\boldsymbol{C}_c$ is the set of conceptual components hierarchically decomposed, $\boldsymbol{I}$ is the set of interconnections among components, $\boldsymbol{Ct}\,(\boldsymbol{C}_c, \boldsymbol{I})$ is the set of constraints localized on components and their interconnections, respectively.

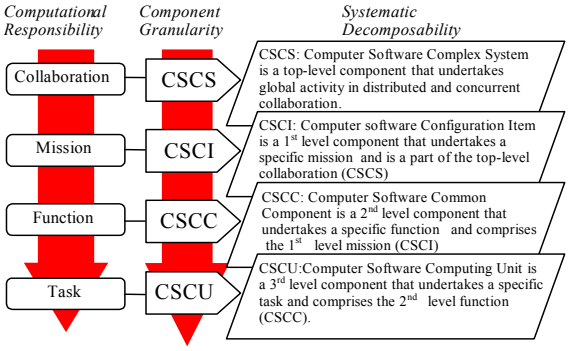

Fig. 2 Computational Responsibility and Properties

The constraints on components have properties of decomposability (representing the hierarchical level at which the constraint is implemented) and granularity (representing the logical packaged complexity of the component). Granularity is an important factor for constructing complex systems because well-grained components are helpful not only to increase productivity but also to improve understandability, reliability and maintainability. As illustrated in Fig. 2, a schema is introduced for PBA components, which identifies the granularity and decomposability of each level of computational responsibility.

3.2 Compositional architecture

Compositional architecture accounts for the architect's perspectives of explicit treatment of system composition and architecture with constraints localized on compositional patterns. Detailing what kinds of rules (patterns) are used to govern interactions among components and how quantitative constraints are associated with the patterns, this perspective addresses what kinds of interactions are applied among components and how to associate constraints with compositional patterns. This perspective is represented as follows:

$$\boldsymbol{P}_{\text{composition}} = [\boldsymbol{C}_c \Rightarrow \boldsymbol{R},\ \boldsymbol{R}_o \overset{S}{\underset{P}{\longrightarrow}} \boldsymbol{R}_i,\ \boldsymbol{Ct}\,(\boldsymbol{R}, \boldsymbol{S}, \boldsymbol{P})]$$

Where $C_c \Rightarrow R$ is the set of roles extracted from conceptual components. $R_o \text{—}^{S}/_{P}\text{→} R_i$ is the set of compositional patterns: R_o (output/producer) interacts with R_i (input/consumer) via architectural styles S while complying with communicatory protocols P. $Ct(R, S, P)$ is the set of constraints localized on roles, styles, and protocols, respectively.

Constraints on interactions further localized on architectural styles are embodied in such properties as composability and heterogeneity. Composability represents the hierarchical composition of architecture (i.e., an entire architecture becomes a single component in another larger architecture). Heterogeneity represents the diverse ways components interact with each other. Heterogeneity is inevitable in complex systems because diverse components or systems will have to work and interact together. In Fig. 3, a compositional coupling schema is introduced for PBA approach.

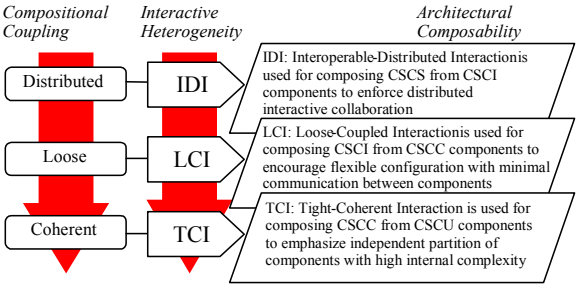

Fig. 3 Compositional Coupling and Properties

3.3 Derivational implementation

Derivational implementation accounts for the implementer's perspectives of component derivation and connectivity. This perspective addresses what kinds of components are needed to carry out computational activity, what connectivity is needed between the components and how to glue the components to specific roles. This perspective is represented as follows:

$$P_{\text{derivation}} = [R \supset C_p,\ (C_p \rightarrowtail R_o) \text{—}^{S}/_{P}\text{→} (R_i \mapsto C_p),\ Ct\,(C_p\, S,\ P)]$$

Where $R \supset C_p$ is the set of physical components derived from the associated role. $C_p \rightarrowtail R$ (its peer $R_i \mapsto C_p$) is the set of instantiated components that are glued to associated roles. $Ct(C_p, S, P)$ is the set of constraints localized on physical components, styles, and protocols, respectively.

Constraints on components are embodied in such properties as connectivity (representing the way components are derived from the related role) and evolvability (representing the evolution from roles to components). Interactive roles are represented as generalized role wrappers (GRWs) (an abstract class) to support component evolution through sub-typing and refinement. As illustrated in Fig. 4, the GRWs defined in PBA composers introduce derivational gluing to refer to connectivity and evolvability.

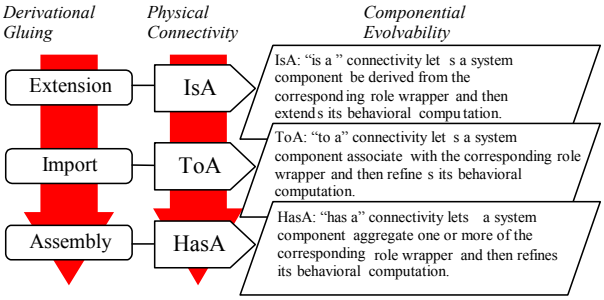

Fig. 4 Derivational Gluing and Properties

3.4 Automated transitional process

Starting with a prototyping model in the computational activity perspective, a transitional process is formed from computational activity, through compositional architecture, to derivational implementation. Two kinds of architectural elements evolve: PBA composers and PBA components. Under the support of automated software tools, two key mappings are used to bridge the gaps between perspectives: explicit architecting via compositional patterns and physical evolution via PBA composers. PBA approach is associated with support tools such as Prototyping Analyzer, Pattern Selector, Framework Generator, and Component Evolver [5,8]. Fig. 5 illustrates this transitional process.

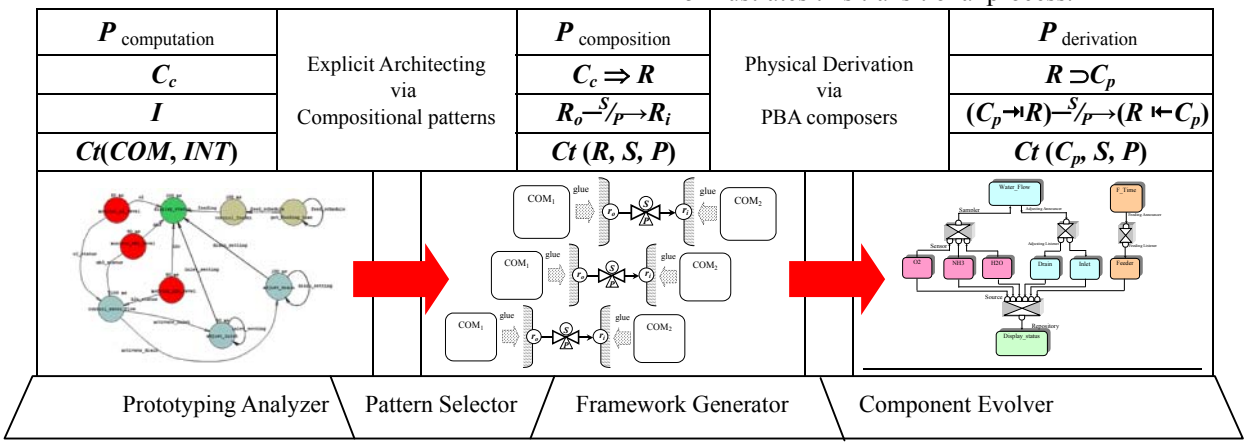

Fig. 5 Transitional Process between Architectural Perspectives

Explicit architecting of the computational activity starts with assigning components with specific roles. According to the architectural styles, related interactive roles and communicatory protocols can be determined so that suitable compositional patterns can be selected and applied to govern the interconnections among the roles. According to the assignment of which components play which specific roles, the components will be derived from the associated role facility. After being derived, the components will be instantiated and then glued to the associated roles by the PBA configuration.

A PBA composer is designed as a generic package-like architectural entity that includes two kinds of GRWs: one is for the "interactive producer" and the other is for the "interactive consumer." GRWs provide adherence to restricted, plug-compatible interfaces for interaction and provide the template of behavior that components are expected to refine. The physical connectivity between a component and a role is implemented by refining or overriding the restricted, plug-compatible interfaces defined by the GRW [11, 18].

4. Dependable Compositional patterns

Compositional patterns provide a set of rules that govern the interactions among components with localized constraints. They are characterized by three kinds of formulated arguments: interactive roles, architectural styles, and communicatory protocols.

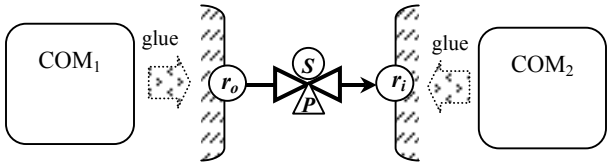

Fig. 6 Compositional pattern for interconnections

Fig. 6 depicts a compositional pattern. For a given interaction between two components (COM_1, COM_2), both are assigned to play specific roles $\mathbf{r}_o$ and $\mathbf{r}_i$ in the specific compositional pattern. An architectural style $\boldsymbol{s}$ specifies how $\mathbf{r}_o$ (output / producer) interacts with $\mathbf{r}_i$ (input / consumer), while communicatory protocol $\boldsymbol{p}$ builds a specific channel for message transportation during the interaction. More specifically, in order to construct the components as autonomous entities, roles in the compositional pattern are deputized for the components in dealing with interaction while the associated components are mainly concerned with their functionality (computation separated from interaction). The pattern also provides a means for gluing a specific component to a role.

By mathematically defining the compositional patterns, it is possible to translate, localize, and analyze them using automated CASE tools. Compositional patterns involve three sets: $\boldsymbol{R}$ representing interactive roles, $\boldsymbol{S}$ representing architectural styles, and $\boldsymbol{P}$ representing communicatory protocols. Examples include:

$\boldsymbol{R}$ = {	$\boldsymbol{S}$ = {	$\boldsymbol{P}$ = {
Caller, Definer,	Explicit-invocation,	Message-passing,
Announcer, Listener,	Implicit-invocation,	Event-broadcast
Outflow, Inflow,	Pipe-filter (Pipeline),	Data-stream,
Source, Repository,	Repository-centric,	Sampled-stream
Read, Writer, …	Blackboard, …	Shared-data, …
}	}	}

Regardless of any constraint, a composition is defined as an interaction between two roles (e.g., Caller and Definer) via an architectural style (e.g., explicit-invocation), while complying with a communicatory protocol (e.g., message-passing). So, the Cartesian product R x S x P x R enumerates all possible compositions C, represented as follows:

$$\boldsymbol{C}\,(\boldsymbol{R}, \boldsymbol{S}, \boldsymbol{P}) = \{\, \boldsymbol{r}_o \text{—}^{s}/_{p}\!\rightarrow \boldsymbol{r}_i \mid \mathbf{r}_o, \mathbf{r}_i \in \boldsymbol{R}, \boldsymbol{s} \in \boldsymbol{S}, \boldsymbol{p} \in \boldsymbol{P} \,\}$$

Where $r_o \text{—}^{s}/_{p}\!\rightarrow r_i$ represents interaction between $\boldsymbol{r}_o$ and $\boldsymbol{r}_i$ via a style $\boldsymbol{s}$ while complying with a protocol $\boldsymbol{p}$.

Applying specific constraints on compositions develops sophisticated compositional patterns. While GRWs provide adherence to restricted, plug-compatible interfaces for interaction and template of behavior for computation, the components derived from GRWs will specify, refine or override the template. In this way, interactions are separated from computations.

Compositional patterns CP are the relation on the Cartesian product of compositions with the constraints reasonably localized on roles, styles and protocol:

$$\boldsymbol{CP}(\boldsymbol{R}, \boldsymbol{S}, \boldsymbol{P}) = \{\mathbf{GRW}(\boldsymbol{r}_o) \text{—}^{s}/_{p}\!\rightarrow \mathbf{GRW}(\boldsymbol{r}_i) \mid \boldsymbol{r}_o, \boldsymbol{r}_i \in \boldsymbol{R}, \boldsymbol{s} \in \boldsymbol{S}, \boldsymbol{p} \in \boldsymbol{P}, \mathbf{Ct}(\boldsymbol{r}_o, \boldsymbol{s}, \boldsymbol{p}, \boldsymbol{r}_i)\,\}$$

Where $\boldsymbol{GRW}(\boldsymbol{r})$ abstracts the role $\boldsymbol{r}$ as a GRW that separates interaction from computation (the GRW "provides" while the component "performs"). $\text{—}^{s}/_{p}\!\rightarrow$ represents interaction between $\boldsymbol{r}_o$ and $\boldsymbol{r}_i$ via a specific style $\boldsymbol{s}$ while complying with a specific protocol $\boldsymbol{p}$. $\boldsymbol{Ct}(\boldsymbol{r}_o, \boldsymbol{s}, \boldsymbol{p}, \boldsymbol{r}_i)$ represents localized constraints.

4.1 Example of compositional patterns

Compositional patterns can be implemented as composers, an explicit architectural element. They can be organized in a reusable composer library that provides the evolutionary foundation for component derivation. Fig. 7 gives the typical composer *Pipeline* that exhibits dependable architectural properties (e.g., loose component coupling, asynchronous communication, and data buffering). The two sides interconnected by the composer are the *Outflow* and *Inflow* roles, respectively. *Outflow* deputizes the producer to output the data, while *Inflow* deputizes the consumer to input the data via *Pipeline*. The formal *Pipeline* composer provides two generic parameters for enhancing reusability: transported ***Data*** (a basic item for dataflow) and buffer ***Size*** (a data transportation buffer).

This example provides a template for GRWs. With respect to behavioral computation of components, the CSP-based semantic description provides not only synchronous constraints but also asynchronous control transits. Both *Output* and *Input* are designed as exclusive procedures (*execution guards* are used to coordinate concurrent synchronization). Reference timing constraints [5-6], the role of *Outflow* is subjected to a maximum execution time (*met*) while *Inflow* is subject to a maximum response time (*mrt*). Both **met**(100) and **mrt**(100) are translated as asynchronous control transits for runtime monitoring of the real time constraints. "◊" represents an asynchronous operation. When outputting produced data onto the given pipeline, *Outflow* must be synchronized within a **met**(100) otherwise, an exception is triggered.

```
composer Pipeline is generalized
      type Data is private;
      Size : Integer : = 100;
style as   <#pipe-filter#>;
protocol as   <#dataflow-stream#>;
wrapper
   role Outflow is
    port
       procedure Output(d: Data);
       procedure Produce(d: Data) is abstract;
    computation
         Produce (d);
       *[ Output (d) → Produce (d) ◊ met(100) →exception; ]
    end Outflow;
    role Inflow is
    port
       procedure Input(d: Data);
       procedure Consume(d: Data) is abstract;
    computation
       *[ Input (d)→ Consume (d) ◊ mrt(100) →exception; ]
    end Inflow;
collaboration (P : Outflow; C : Inflow)
      P•Produce(d);
   *[ P•Output(d)→ P•Produce(d) □ C•Input(d) → C•Consume (d)]
end Pipeline;
```

Fig. 7 A Formal composer for Pipeline

The collaboration portion of the composer description will generate topological configurations that are connected graphs of components and composers. In concert with models of components and composers, configurations enable assessment of the autonomous and concurrent aspects of an architecture (such as the potential for deadlocks, starvation, reduced performance, reliability, security, etc.). Configurations also enable concurrent execution immediately after the roles are glued with the instances of corresponding components.

4.2 Substantiated interconnection

It used to be that interconnections in the architecture of a software system were annotated as a series of "box-line-box" diagrams [8-10]. Over time, this annotation has become much richer (for instance, the use of Architecture Description Languages (ADLs)) in order to more precisely capture and communicate more complex ideas related to interconnection. PBA continues in this vein by substantiating the interconnections among components so that large, complex architectures of systems can be built, dealing with following four aspects:

- *Dependable composers* by which interaction among components are promoted,
- *Heterogeneous forms* by which communication during interaction can be established.
- *Topological connectivity* that guides the connected configuration of components, and
- *Constraint localization* that governs interconnections by associating constraints on patterns

Dependable composers are used to implement compositional patterns by analyzing interactive roles of interconnected components in the prototyping model. Heterogeneous forms are associated with architectural styles and the way information is transported and refers to as communicatory protocols in compositional patterns. Constraint localization is presented next.

Topological connectivity simplifies the interconnection among components and comes in the following forms:

- Fork (1~N): single producer to multiple consumers
- Merge (N~1): multiple producers to single consumer
- Unique (1~1): single producer to single consumer
- Hierarchy: external[1] producer to interact with the internal[1] consumer, and vice versa.

Fig. 8 illustrates how to use a composer to implement *Fork* between one producer and more than one consumer.

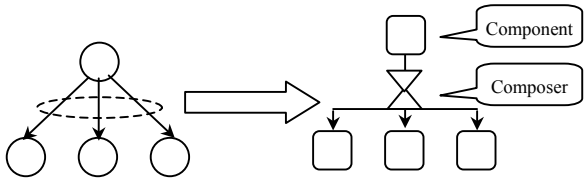

Fig. 8 Fork Connectivity with one PBA composer

4.3 Dependability as a set of Constraints

In this case "localization" represents the abstraction of dependability, its translation to quantitative constraints, and the handling of these constraints applied (localized) in the design, construction, and evolution of a software-intensive system. In order to achieve high confidence in the dependability of a system there must be a systematic method for expressing the dependability objectives via measurable constraints associated with the subsystems of the architecture. In a macro view, dependability can be abstracted as availability, reliability, safety, confidentiality, integrity and flexibility [15-17]. How these qualitative global requirements translate into quantitative constraints becomes crucial. Which dependable properties need translating and how they are localized on compositional patterns are the questions that have to be answered.

[1] External and internal refer to hierarchical decomposition. For a given hierarchical level of decomposition, a component in the current level is *external* to a component in a lower level, while the latter is the *internal* to the former.

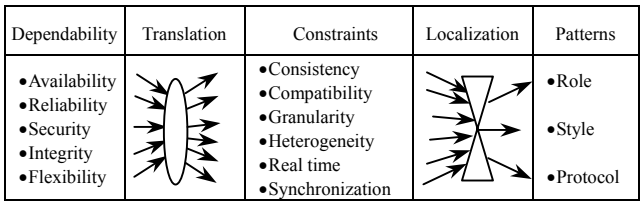

Fig. 9 Localization of Dependability

Fig. 9 shows a framework of localization applied to dependability. With respect to translating dependability and localizing the semantic constraints on the compositional patterns, the handling of real-time constraints provides a good example. Reliability of the time-critical system may be embodied as an immediate reply of a particular component, under a given request, within an *met*, or as a data stream between components performed within a specific *latency* [5]. First, this time-critical reliability should be translated into timing constraints *met* and *latency* (two quantitative constraints). Both are associated with the patterns referring to the role and protocol, respectively. *met* requires computation of the role (the component acts) and must be executed within a specific amount of time (a hard real-time constraint). The *latency* constrains the maximum delay during data transportation within the protocol. These timing constraints can be also verified by runtime monitoring and correctness assurance [15-17]. Dependability of the system would be translated into in the form of maximum execution time or latency of the data stream communication between components as shown in Fig. 10.

```
composer Pipeline is generalized
    …
  role Outflow is
  port
      procedure Output(d: Data);
      procedure Produce(d: Data) is abstract;
  computation
          Produce (d);
       *[ Output (d) ◊ latency(60) → Produce (d) ◊ met(100)
          □ latency-signaled          → LAT-EXCEPTION
          □ met-signaled              → MET-EXCEPTION
        ]
  end Outflow;
    … …
end Pipeline;
```

Fig. 10 A Formal composer for Pipeline

Procedure Output can be treated as execution guard that is tied to the communication protocol, so latency is associated with to the protocol by Output (d) ◊ **latency**(60). When executing Output is beyond the latency, the asynchronous control will set latency-signaled and abort current execution, and then raise LAT_EXECPTION. Similarly, *met* is directed to the procedure Produce by Produce (d) ◊ **met**(100). When executing Produce is beyond the *met* limitation, the asynchronous control will set met-signaled and abort current execution, and then raise MET_EXCEPTION.

5. Conclusion

Explicitly architecting software-intensive systems provides the promise of faster, better, cheaper systems. In order to consistently engineer dependable software-intensive systems, the PBA approach provides a process that uncovers perspective concerns of different stakeholders, and increases the effectiveness of requirements validation techniques. Because PBA approach can be used to localize and quantify invariant architectural constraints (such as "dependability" in the example above) it will also reduce the amount of re-certification effort required after each requirement change. The PBA approach illustrates that with automated tool support, the prototyping of software architecture can be used to identify and resolve conflicting stakeholder perspectives and develop reliable, dependable, consistent software-intensive systems.

References

[1] C. Hofmeister, R. Nord, D. Soni. Applied Software Architecture. Addison-Wesley, 2000.

[2] IEEE Standard Board, Recommended Practice for Architectural Description of Software-Intensive Systems (IEEE-std-1471 2000), September 2000.

[3] H. Alexander, *et el*, C4ISR Architectures: I. Developing a Process for C4ISR Architecture Design. Systems Engineering, John Wiley and Sons, Inc., Vol. 3 No. 4, 2000.

[4] W. Lee, et el, Synthesizing Executable Models of Object Oriented Architectures. Proc. Formal Methods in Software Engineering & Defence Systems. Adelaide, Australia, 2002.

[5] Luqi, M. Ketabchi, A computer-Aided Prototyping System, IEEE Software, March 1988.

[6] Luqi, Ying Qiao, Lin Zhang, Computational Model for High-Confidence Embedded System Development, Monterey workshop 2002, Venice, Italy, Oct 7-11, 2002.

[7] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Inc., 1996.

[8] Andrew P., Systems Integration and Architecting: An Overview of Principles, Practices, and Perspectives, System Engineering, John Wiley and Sons, Inc., 1998.

[9] N. R. Mehta, N. Medvidovic. Towards a Taxonomy of software Connectors. Proc. ICSE, Limerick Ireland, 2000.

[10] N. Medvidovic, Taylor, A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering, 2000, 26(1).

[11] X Liang, Event-based implicit invocation decentralized in Ada, ACM AdaLetters, March, 2002.

[12] Sessions N., COM and DCOM: Microsoft's Vision for Distributed Objects. John Wiley & Sons, Inc., NY, 1997.

[13] OMG/ISO Standard, CORBA: Common Object Request Broker Architecture, http://www.corba.org/.

[14] Sun Microsystems, Inc. Java 2 Enterprise Edition Specification v1.2. http://java.sun.com/j2ee/.

[15] E. M. Clarke (CMU), R. P. Kurshan (Bell Lab). Computer-Aided Verification, Feb. 17, 1996.

[16] James C. Corbett, et el, Bandera: Extracting Finite-state Models from Java Source Code, Proc of the ICSE 2000.

[17] M. Kim, *et el*, Monitoring, Checking, and Steering of Real-Time Systems, 2nd Intl. Workshop on Run-time Verification. Copenhagen, Denmark, July 26, 2002.

[18] X Liang, Z. Wang. Omega: A Uniform Object Model Easy to Gain Ada's Ends, ACM AdaLetters, June, 2000.

Reliability Support for the Model Driven Architecture*

Genaína Nunes Rodrigues, Graham Roberts, Wolfgang Emmerich and James Skene
Dept. of Computer Science
University College London
Gower Street, London WC1E 6BT, UK
{G.Rodrigues|G.Roberts|W.Emmerich|J.Skene}@cs.ucl.ac.uk

Abstract

Reliability is an important concern for software dependability. Quantifying dependability in terms of reliability can be carried out by measuring the continuous delivery of a correct service or, equivalently, of the mean time to failure. The novel contribution of this paper is to provide a means to support reliability design following the principles of the Model Driven Architecture(MDA). By doing this, we hope to contribute to the task of consistently addressing dependability concerns from the early to late stages of software engineering. Additionally, we believe MDA can be a suitable framework to realize the assessment of those concerns and therefore, semantically integrate analysis and design models into one environment.

1. Introduction

Component-based development architectures (CBDA) are increasingly being adopted by software engineers. These architectures support distributed execution across machines running on different platforms (e.g. Unix, Windows). Examples of component models include Sun's Enterprise Java Beans (EJB), OMG's CORBA Component Model (CCM) and Microsoft's .NET. Additionally, CBDAs rely on the construction and deployment of software systems that have been assembled from components [3].

One of the advantages of applying a component-based approach is reusability. It is easier to integrate classes into coarse-grained units that provide one or more clearly defined interfaces. However, the lack of interoperability among diverse CBDAs may be one of the major problems that hinders the adoption of distributed component technologies. Once a platform has been chosen and the system has been developed, porting to a different platform becomes troublesome.

*This research is partially supported by CAPES and the European Union under grant IST-2001-34069

To fill the gap, the OMG has focused on paving the way to provide CBDAs interoperability standards through the Model Driven Architecture (MDA). Essentially, "the MDA defines an approach to IT system specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform" [9]. To accomplish this approach, MDA structures the system into key models: the Platform Independent Models (PIMs) and the Platform Specific Models (PSMs). While a PIM provides a formal specification of the structure and function of the system that abstracts away technical details, a PSM expresses that specification in terms of the model of the target platform. Basically, PIMs are mapped to PSMs when the desired level of refinement of PIMs is achieved.

The Unified Modeling Language (UML) is the core element to represent those models. According to the OMG, UML supports the formalization of an abstract, though precise, models of the state of an object, with functions and parameters provided through a predefined interface [9]. Furthermore, UML models facilitate the assessment of a design in the early stages of software development, when it is easier and cheaper to make changes.

The defined and standard structure of MDA would seem suitable to address software dependability, in that the MDA designates the system function as required by the stakeholders. Issues such as reliability, safety, security and availability comprise software dependability [8, 12]. However, there is no standard representation for dependability in MDA models. During the software execution this can lead to situations not foreseen in the platform models.

Reliability assurance is an important concern in software dependability. Quantifying dependability in terms of reliability can be carried out by measuring the continuous delivery of correct services or equivalently, of the mean time to failure [4]. A system can be considered reliable if it performs at a constant level, as the stresses on that system change. For example, if a0 request takes 10 ms to complete with one user, then it takes the same time to process

the same request with 1000 concurrent users.

To overcome the lack of dependability concern in the current MDA specification, we propose to explicitly tackle this problem in the levels of abstraction suggested by OMG's MDA. We believe it is feasible to accomplish this task using the standard meta-modeling approach of MDA and specifications, such as the work in [10], as sources to achieve this goal. Our focus on dependability at the moment is reliability. To guarantee and assess reliability properties of software systems using the MDA approach, we plan to achieve reliability in such a way that it would be specified in the early stages of software architecture design. In this way, we aim to provide reliability in a platform-independent way. In the context of MDA and current distributed component-based architectures, early reliability assessment is important as the software architecture design is specified in the context of software development.

The novel contribution of this paper is to provide a means to support reliability design following the principles of MDA. By doing this, we hope to contribute to the task of consistently carrying out dependability concerns from the early to the late stages of software engineering. Besides, MDA appears to be a suitable framework to realize the assessment of those concerns and therefore, semantically integrate analysis and design models into one environment.

In this position paper, we elaborate our approach on how the provision of reliability can be suitably realized through a standard model-driven architecture approach. In Section 2, we present the related work targeting reliability support and analysis in the CBDA scenario. In Section 3, we show how we plan to provide reliability modeling in MDA and the steps to be followed to accomplish this goal. In Section 4, we provide a sample scenario of how our approach addresses reliability support in MDA. Finally, Section 5 summarizes our contribution and discusses future work towards achieving standard reliability support from designing models to programatic interfaces.

2. Related Work

The work described in [1, 2, 7] looks at part of the problems we identify in our work, in terms of addressing dependability concerns in the early stages of software development. We can basically find in these works analysis techniques to validate design tools based on UML.

However, they differ from our approach in some important aspects. Primarily, they do not conform to the principles stated by MDA. MDA uses the straightforward approach through the concepts of mapping models among different platforms. Therefore, we believe that MDA offers a suitable environment to consistently integrate the analysis and design of dependability issues, and from design to implementation. [1] provides a useful transformation technique to automate dependability analysis of systems designed using UML. Nevertheless, to properly contemplate dependability in all stages of the software engineering process, we believe that one of the main concern is to provide a unified semantic between the analysis and the design models.

Another approach to address software dependability is to provide mechanisms to improve reliability of software after it has been implemented. Works such as [5] use testing techniques to identify faults in the software that are likely to cause failures. Although they carry out an important research agenda, we believe that is cheaper to design and evaluate dependability concerns in the early stages of software engineering processes . Besides, levels of abstraction like the one expressed by the PIM and PSM models of MDA seems to be necessary in a scenario where each of the existing CBDA holds distinct mechanisms to support dependability.

A meta-model is a model of a language expressed using a modeling technique. This feature in UML allow us to express the design and analysis domains naturally, using the concepts inherent to these domains. Moreover, this facility permits to map the behavior of distributed component architectures into a domain knowledge keeping the semantics of the modeling requirements of UML. Following this principle, our approach to meta-modeling using the UML lightweight extension mechanisms, i.e. profiles, is consistent with the official MDA white paper [9], which defines basic mechanisms to consistently structure the models and formally express the semantics of the model in a standardized way. Moreover, the profiles define standard UML extensions to describe platform-based artifacts in a design and implemented model. For example, the UML Profile for EJB [6] supports the capture of semantics expressible with EJB through the EJB Design Model and the EJB Implementation Model. Although currently cannot be found UML profiles to thoroughly address dependability, MDA seems to be a feasible environment to consistently assess and express dependability by means of profiles properly constructed.

Another benefit that arises from this consistent integration is the facility to realize reverse engineering. However, it is out of scope of our current work to cope with this topic.

3. A Profile for Reliability

To provide a reliability profile for MDA, we follow a bottom-up approach as MDA allows this flexibility (see Figure 2). Having J2EE as a first target to realize a reliability profile, we plan to extend the UML Profile for EJB [6] to express reliability primitives available in J2EE in a standard way. By standard way, we mean to specify a subset of UML meta-model that describes the semantics of mechanisms in J2EE to achieve reliability. This subset contains

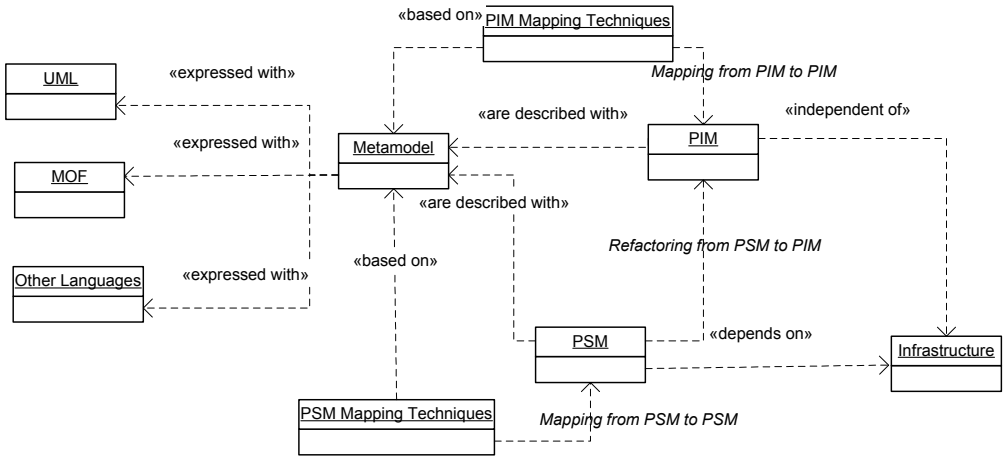

Figure 1. MDA Metamodel Description [9]

stereotypes, tagged values and OCL constraints.

To assure reliability, the J2EE platform has several mechanisms [15]:

- Fail-Over through clustering of containers[1]
- Asynchronous communication with persistent JMS and Message Beans;
- Persistent Entity Beans;
- Transactions through the two-phase commit protocol;
- Security.

For the sake of achieving abstraction, these mechanisms should be supported by the UML/EJB profile. In order to realize this task, UML meta-models must be designed to reflect each of those mechanisms, relying on the current UML Specification [11]. By doing this, it will be possible to express semantics and notations that adequately address those reliability mechanisms in a standard way.

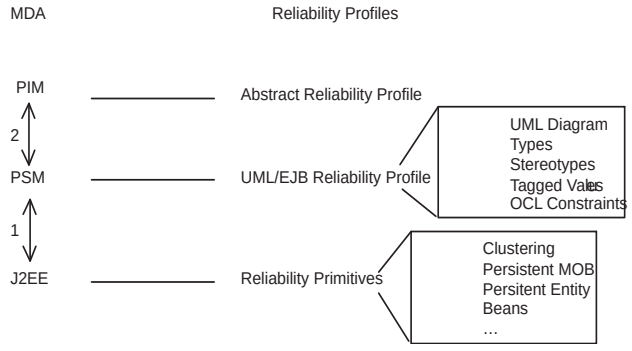

Figure 2. Reliability Profiles in MDA

[1]Interested readers may refer to [13], chapter 14

UML provides a wide set of modeling concepts and notations to meet current software modeling techiniques. The various aspects of software systems can be represented through UML, what makes it suitable for architecture modeling. Besides, it is widely used for software analysis and development. To model complex domains in UML, new semantics can be added through extension mechanisms packages that specify how specific model elements are customized and extended with those semantics. We adopt profile as our extension mechanism, which comprises model elements customized for a specific domain or purpose (e.g. designing reliability) by extending the meta-model using stereotypes, tagged definitions and constraints [11]. In order to design and formalize J2EE reliability mechanisms, we will first map them into a profile, the Refactoring mechanism in Figure 1.

In MDA, a mapping is a set of rules and techniques used to modify one model in order to get another model. The following step is to design reliability in the highest abstract level stated by the architecture of MDA, which is the PIM. Achieving a platform-independent reliability model, the task of designing dependability concerns can be carried out in the early stages of software engineering where the software architecture is designed. The steps to accomplish this goal are as follows:

1. Determine the reliability properties of interest.
2. Create a set of stereotypes, tagged values and constraints to build the UML/EJB Profile for Reliability.
3. Provide the design domain mappings between each reliability profile instances and the UML/EJB Profile.
4. Define a mapping between the design domain achieved in the previous step and a platform-independent design domain that correctly represents the semantics of each

reliability mechanism. A preliminary PIM version is expected at the end of this step.

5. Identify those qualities that are of interest but require formal analysis to determine. Choose an appropriate analysis technique for reliability analysis (e.g Bayesian Networks) and define a profile to represent the entities within the analysis domain.

6. Define a mapping between the design domain and the analysis domain that correctly represents the semantics of each.

7. Choose a commercial existing component model other than EJB for J2EE platform to make the PIM to PSM mapping, providing the PIM reliability profile.

8. Automate the mapping.

9. Provide scenarios to monitor and assess the models in a real-life case study.

It should be noticed that in Step 4, the mapping from PSM to PIM will be carried out in order to reach the highest abstract level of reliability mechanism. The MDA principles allows this bottom-up approach and we decided to follow this step in order to raise the kind of resources needed in a reliability UML profile. This mapping can be formalized using the Object Constraint Language (OCL), which is the formal language used to specify well-formedness rules of the meta-classes comprising the UML meta-model [11]. However, this formalization would require an assessment of the designed properties. That is the purpose of Step 6. Achieving this level of abstraction is not the whole plan however. In Step 8, mechanisms to automate the target reliability primitive may be desired by those who want to apply the reliability profile attained in our work. Finally in Step 9, we apply and evaluate our approach using a real-life case study.

4. A Scenario of Reliability Support in MDA

This section shows how we plan to achieve the previously stated goals through an example. We highlight how one of the reliability mechanisms can be mapped to a UML profile in a standard way and how it would reflect on the deployment of the components. To make it concrete, we plan to first build a reliability profile for a target platform, which is the J2EE platform. By doing this, it will be easier to identify the kind of resources for reliability that should be comprised in a platform-specific model and therefore those to be comprised in a platform-independent model through MDA mappings.

In this scenario, we exploit one of the mechanisms of EJB to provide reliability, the fail-over through clustering. The fail-over mechanism redirects a single request to another node in the cluster because the original node could not process the request. This implies another concept, which is clustering. The overall goal of employing a cluster is to increase the availability or reliability of the system by joining services into groups that provide services to their clients in a loosely coupled way. The number of nodes comprising a cluster will vary according to the degree of reliability we want to assure.

The first step towards achieving reliability in MDA principles, is to define the architecture of the application by means of the UML Profile for EJB [6]. To express how reliable the method invocations will be and the deployment relationships between the application components, a reliability profile is needed. Figure 3 shows what the overall scenario would look like. Basically, there are three main profiles: the design (where the reliability mechanism is modeled), the mapping (to map the desired reliability to the designed classes), and the deployment (to provide how the components will be distributed in the network according to the required reliability support).

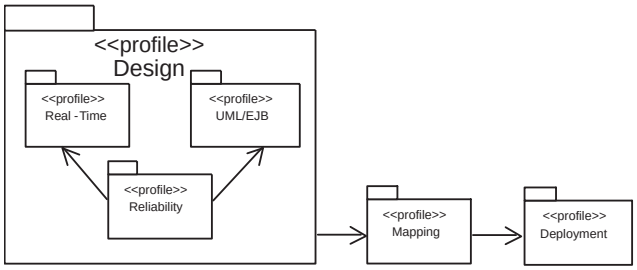

Figure 3. Profiles to model reliability in EJB applications

In the design profile, meta-modeling techniques will be used to map out reliability mechanisms in a profile. This profile is composed of three main specifications:

1. UML/EJB Profile - which expresses the basic semantics of EJB in the UML notation.

2. UML Profile for Schedulability, Performance and Real-Time Specification (briefly, Real-Time Profile) [10] - for the reason that it specifies how applications requiring a quantitative measure of time can be modelled in a MDA standard way.

3. UML Specification [11] - to realize what is lacking in the above specifications to carry out the reliability profile following standardized UML notations, definitions and semantics.

In the mapping domain, where the mapping profile will be realized, constraints that rule the desired reliability mechanism are mapped to a designed application. For

example, all the stateful session beans to be replicated throughout the clusters should be idempotent (i.e. they can be called repeatedly without worrying about altering the system so that it becomes unusable or provides errant results). Finally, the deployment profile will provide the configuration of how the components will communicate and be distributed throughout the network.

In order to map the clustering mechanism proposed in this scenario, we should know what is the desired reliability assurance of the system. By this means, it is possible to know how many replicated components there should be in each cluster to guarantee the desired reliability level.

The functional formula for this assurance is:

$$1 - (1 - c)^n > a \quad (1)$$

where c is the reliability of each component, a is the required reliability of the system and n is the number of components that should be comprised in each cluster. Suppose a is 95% and c is 75%. Then, according to Formula 1 the value of n is 3. Therefore, each cluster of the deployment diagram should have at least 3 copies of the component to be replicated.

To reflect this scenario, the classes of the design profile to be replicated should be mapped to the deployment profile through the mapping profile. A fragment of the mapping profile to assure the reliability property above is described in OCL as follows:

```
context mapping inv:
self.supplier.ownedElements->select(
  m : ModelElement | m.oclIsType(Class) and
  m.stereotype->exists(name =
  "replicatedComponent"))-> forAll(
     (1 - (1 - m.taggedValue->any(type =
     "componentReliability").dataValue)^
      self.consumer.ownedElements->
         select(n : ModelElement |
           n.oclIsType(Component) and
           n.name = m.name))->
           size())> m.taggedValue->any(type =
           "aggregateReliability").dataValue)
```

where `self.supplier` refers to the classes in the designed profile and `self.consumer` refers to the components in the deployment profile.

There is, however, one important step that is not described here but must be accomplished, which is the support in MDA for formal analysis. In this regard, it is required a formal analysis profile to separately express dependability in an analysis model. This accomplishment might follow the approach in [14], which is under development here at UCL. That work aims at providing a MDA performance analysis to enterprise applications and has shown that worthwhile benefits arise, such as:

- Flexible application of analysis techniques to design domains by defining new mappings;
- Use of model checking tools to check the semantic validity of the analytic model against the design model.

5. Conclusions And Future Work

In this paper we have presented the idea on how we plan to tackle the problem of reliability assurance in MDA. The motivation to achieve this purpose is the identified importance and benefits arising from addressing dependability concern in the stage of software engineering where the architecture is designed.

There are many steps, however, to accomplish that task. First of all, a reliability profile should be carried out. In order to achieve a consistent and integrated environment, all the steps should be expressed within the available mechanisms of MDA. Exploiting the standard UML and the profiles already created constitutes the basis of our work. However, there are complementary mechanisms that the current MDA does not provide. For example, ways to assess the designed dependability issues. Therefore, a profile to carry out this assessment should be created by means of metamodeling techniques, following the same approach of [14].

Immediate future challenges include determining precisely a profile to translate reliability in terms of a valid MDA profile. To achieve this goal within a more concrete approach, we plan to map into that profile the reliability mechanisms available in the J2EE platform. This bottom-up approach is expected to aid in identifying the required resources that should be mapped in a reliability-aware PSM. Following all the steps presented in Section 3, we finally wish to enhance the level of automation for mappings and evaluate the practicality of our approach using real-life case studies with realistic complexity.

Acknowledgments

We would like to thank Licia Capra, Rami Bahsoon and Philip Cook for their assistance with this document.

References

[1] A. Bondavalli, I. Majzik, and I. Mura. Automatic Dependability Analysis for Supporting Design Decisions in UML. In R. Paul and C. Meadows, editors, *Proc. of the 4^{th} IEEE International Symposium on High Assurance Systems Engineering*. IEEE, 1999.

[2] V. Cortellessa, H. Singh, and B. Cukic. Early reliability assessment of uml based software models. In *Proceedings of the third international workshop on Software and performance*, pages 302–309. ACM Press, 2002.

[3] W. Emmerich. Distributed Component Technologies and Their Software Engineering Implications. In *Proc. of the* 24^{th} *Int. Conference on Software Engineering, Orlando, Florida*, pages 537–546. ACM Press, May 2002.

[4] J. C. L. et.al. *Dependability: Basic Concepts and Terminology*. Springer–Verlag, 1992.

[5] P. Frankl, R. Hamlet, B. Littlewood, and L. Strigini. Choosing a Testing Method to Deliver Reliability. In *International Conference on Software Engineering*, pages 68–78, 1997.

[6] J. Greenfield. UML Profile for EJB. Technical report, http://www.jcp.org/jsr/detail/26.jsp, May 2001.

[7] G. Huszerl and I. Majzik. Modeling and analysis of redundancy management in distributed object–oriented systems by using UML statecharts. In *Proc. of the* 27^{th} *EuroMicro Conference, Workshop on Software Process and Product Improvement, Poland*, pages 200–207, 2001.

[8] B. Littlewood and L. Strigini. Software Reliability and Dependability: A Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 177–188. ACM Press, Apr. 2000.

[9] Object Management Group. Model Driven Architecture. Technical report, http://cgi.omg.org/docs/ormsc/01-07-01.pdf, July 2001.

[10] Object Management Group. UML Profile for Schedulability, Performance and Real-Time Specification. Technical report, http://www.omg.org/cgi-bin/doc?ptc/02-03-02.pdf, March 2002.

[11] Object Management Group. Unified Modeling Language (UML), version 1.4. Technical report, http://www.omg.org/cgi-bin/doc?formal/01-09-67.pdf, January 2002.

[12] B. Randell. Software Dependability: A Personal View (Invited Paper). In *Proc. 25th Int. Symp. Fault-Tolerant Computing (FTCS-25, Pasadena)*, pages 35–41. IEEE Computer Society Press, 1995.

[13] E. Roman. *Mastering Enterprise Java Beans*. John Wiley & Sons, Inc, 2002.

[14] J. Skene and W. Emmerich. Model Driven Performance Analysis of Enterprise Information Systems. *Electronical Notes in Theoretical Computer Science*, March 2003. To appear.

[15] Sun MicroSystems. Enterprise JavaBeans Specification, version 2.1. Technical report, http://java.sun.com/j2ee/docs.html, August 2002.

FaTC2: An Object-Oriented Framework for Developing Fault-Tolerant Component-Based Systems

Fernando J. Castor de Lima Filho Paulo Asterio de C. Guerra
Cecília Mary F. Rubira
Instituto de Computação
Universidade Estadual de Campinas, Brazil
{fernando.lima, asterio, cmrubira}@ic.unicamp.br

Abstract

Component-based systems built out of reusable software components are being used in a wide range of applications that have high dependability requirements. In order to achieve the required levels of reliability and availability, it is necessary to incorporate into these complex systems means for coping with software faults. In this paper we present FaTC2, an object-oriented framework which facilitates the construction of fault-tolerant component-based systems by giving support to fault tolerance techniques. FaTC2 is an extension of C2.FW, an OO framework which provides an infrastructure for building applications using the C2 architectural style. More specifically, FaTC2 extends C2.FW in order to introduce a forward error recovery mechanism by means of an exception handling system. Our main contribution is to provide a framework which gives support to a software architectural level exception handling system. We also present a case study showing how our framework can be employed for building a fault-tolerant component-based application.

1. Introduction

Modern computing systems require evolving software that is built from existing software components, developed by independent sources[2]. Hence, the construction of systems with high dependability requirements out of software components represents a major challenge, since few assumptions can generally be made about the level of confidence of third party components. In this context, an architectural approach for fault tolerance is necessary in order to build dependable software systems assembled from untrustworthy components[8].

Fault tolerance at the architectural level is a young research area that has recently gained considerable attention[7]. Most of existing works in this area emphasize the creation of fault tolerance mechanisms[9, 11] and description of software architectures with respect to their dependability properties [12, 14].

The work of Guerra et al[6] presents a structuring concept for the incorporation of an exception handling mechanism in component-based systems, at the architectural level. This notion is based on the concept of the Idealised Fault-Tolerant Component(IFTC)[1]. The IFTC separates the abnormal (fault tolerance measures) activities of a system from its normal activity. Upon the receipt of a service request, an IFTC produces three types of responses: *normal responses* in case the request is successfully processed, *interface exceptions* in case the request is not valid, and failure exceptions, which are produced when a valid request is received but cannot be correctly processed.

In this paper we present an object-oriented framework, called FaTC2, for building fault-tolerant component-based systems based on the IFTC. Our framework is an extension of C2.FW[10], an OO framework which provides an infrastructure for building applications using the C2 architectural style[15]. FaTC2 introduces forward error recovery in the original framework by means of an exception handling system (EHS). An EHS offers control structures which allow developers to define actions that should be executed when an error is detected. This materializes by the capability to signal exceptions and, in the code of the handler, to put the system back in a coherent state. A forward error recovery mechanism manipulates the state of a system in order to remove errors and enable it to resume execution without failing. Forward error recovery is usually implemented by means of exception handling.

The C2 architectural style[10, 15] is a component-based architectural style which supports large grain reuse and flexible system composition, emphasizing weak bindings between components. The C2 style has been chosen due to its ability to compose heterogeneous off-the-shelf

components[10]. The work of Rakic and Medvidovic[11] is the only one we know of which describes means for supporting the construction of fault-tolerant C2 applications. It presents the concept of *Multi-Version Connector*, a mechanism created to permit the reliable upgrade of software components in a configuration, by means of design diversity[1].

Our main contribution is the construction of a framework which supports an architectural level EHS. In component-based development, source code for the components which make up a system might not be available, specially if third party components are employed. Hence, it is not possible to introduce exception handling directly in the component. An architectural level EHS deals with this kind of problem by providing an infrastructure for defining exceptions and attaching the corresponding handlers to components without the need to modify them.

The rest of this paper is organized as follows. Section 2 provides some background information. Section 3 presents the proposed framework, FaTC2, describing its most important elements. An example application is presented in Section 4. Final conclusions are given in Section 5.

2. Background

2.1. The C2 Architectural Style

In the C2 architectural style components communicate by exchanging asynchronous messages sent through connectors, which are responsible for the routing, filtering, and broadcast of messages. Figure 1 shows a Software Architecture using the C2 style where the elements A, B, and D are components, and C is a connector.

Components and connectors have a *top interface* and a *bottom interface*(Figure 1). Systems are composed in a layered style, where the top interface of a component may be connected to the bottom interface of a connector and its bottom interface may be connected to the top interface of another connector. Each side of a connector may be connected to any number of components or connectors. Two types of messages are defined by the C2 style: requests, which are sent upwards through an architecture, and notifications, which are sent downwards. Requests ask components in upper layers of the architecture for some service to be provided, while notifications signal a change in the internal state of a component.

2.2. C2.FW Framework

The C2.FW framework[10] provides an infrastructure for building C2 applications. It is part of the ArchStudio[16] environment, which is an architecture-oriented integrated development environment which comprises a collection of tools to help in the development of applications based on

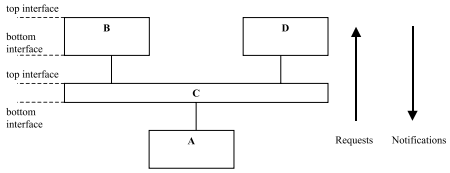

Figure 1. An example architecture using the C2 style.

the C2 style. C2.FW has been implemented in C++, Java, Python and Ada.

The C2.FW Java[5] framework comprises a set of classes and interfaces which implement the abstractions of the C2 style, such as components, connectors, messages, and interconnections. The framework provides various features, such as support to different threading models and queuing policies, and sophisticated message processing and event propagation mechanisms. It does not, however, implement any mechanisms for the provision of error recovery.

2.3. Idealised C2 Component

The work of Guerra et al[6] uses the concept of Idealised Fault-Tolerant Component (IFTC) to structure the architecture of component-based software systems compliant with the C2 architectural style. It introduces the Idealized C2 Component(iC2C), which is equivalent, in structure and behavior, to the IFTC. Service requests and normal responses of an IFTC are mapped as requests and notifications in the C2 architectural style. Interface and failure exceptions of an IFTC are considered subtypes of notifications.

The iC2C is composed of five elements: NormalActivity and AbnormalActivity components, and iC2C_top, iC2C_internal, and iC2C_bottom connectors. Its internal structure is presented in Figure 2.

The NormalActivity component processes service requests and answers them through notifications. It also implements the error detection mechanisms of the iC2C. The AbnormalActivity component encapsulates the exception handlers of the iC2C. While a system is in a normal state, the AbnormalActivity component remains inactive. When an exceptional condition is detected, it is activated to handle the exception. In case the exception is successfully handled, the system enters a normal state and the NormalActivity component resumes processing. Otherwise, a failure exception is sent and components in lower layers of the architecture become responsible for handling it.

The iC2C_bottom connector is responsible for filtering and serializing requests received by the iC2C. This conservative policy aims at guaranteeing that requests are always received by the NormalActivity component in its ini-

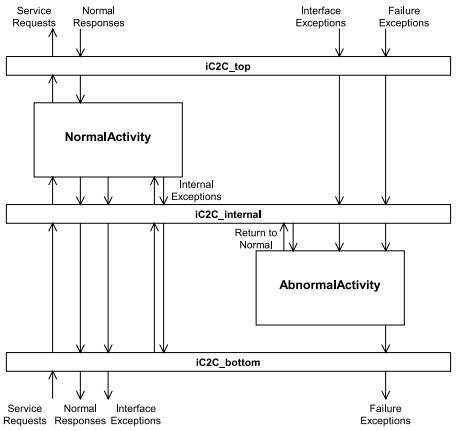

Figure 2. Internal structure of an iC2C.

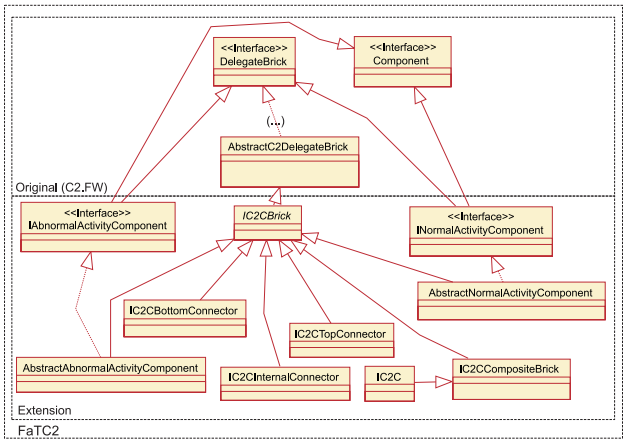

Figure 3. A summarized class hierarchy for C2.FW and FaTC2.

tial state, to avoid possible side-effects of an exceptional condition caused by a concurrent service request. The iC2C_internal connector is responsible for the routing of messages inside the iC2C. The destination of the messages sent by the internal elements of the iC2C depends on the message type and whether the iC2C is in a normal or abnormal state.

The iC2C_top connector encapsulates the interaction between the iC2C and components located in upper levels of the architecture. It is responsible for guaranteeing that service requests sent by the NormalActivity and AbnormalActivity components to other components located in upper levels of the architecture are processed synchronously. And that response notifications reach the intended destinations. The iC2C_top connector also performs domain translation, converting incoming notifications to a format which the iC2C understands and outgoing requests to a format which the application understands.

The structure of the iC2C makes it compatible with the constraints imposed by the C2 architectural style. Hence, an iC2C may be incorporated into an existing C2 configuration. Previous experiments[6, 8] with the IC2C model have shown its adequacy for the construction of component-based systems, including systems built from off-the-shelf components[7].

3. Description of the Framework

In order to facilitate the development of fault-tolerant applications using the C2 style, we have extended the Java[5] version of C2.FW with the concept of iC2C. The original C2.FW framework does not provide adequate support for the construction of fault-tolerant systems. Our aim is to provide the support for error recovery, more specifically, forward error recovery, by means of an EHS.

The extended C2.FW framework has been baptized *FaTC2*, which is an abbreviation for Fault-Tolerant C2. FaTC2 allows fault-tolerant systems to be built in a well-organized manner, using iC2Cs as structural units. The main advantage of this approach is the fact that framework users do not need to implement an EHS in order to create fault-tolerant applications. Only the normal activity(functional requirements) and abnormal activity(exception handling) of the component should be defined. Connections between normal and abnormal parts are managed by FaTC2.

Figure 3 presents a summarized class hierarchy for FaTC2, and its intersection with C2.FW. In the following sections we describe FaTC2, based on the elements which compose an iC2C(Figure 2).

3.1. IC2C

The creation of an iC2C is encapsulated by the **IC2C** class. Instances of **IC2C** are created by a factory method[3] which takes as arguments the name of the iC2C to be created and objects representing the NormalActivity and AbnormalActivity components(Figure 2). Optionally, it may also receive objects representing the iC2C_top and iC2C_bottom connectors as arguments, in case filtering or domain-translation are required. If these arguments are omitted, default implementations are employed.

Although the **IC2C** class may be used directly in an application, it is recommended that developers create subclasses of it, specifying the NormalActivity and AbnormalActivity components, and iC2C_top and iC2C_bottom connectors which are to be used.

3.2. NormalActivity Component

The NormalActivity component is one of the elements of the iC2C which must be implemented by developers employing FaTC2. In order to define a NormalActivity component, a developer must provide a class that implements the **INormalActivity** interface. This interface declares three operations which define the application-dependent behavior of the component: `handleRequest()`, `returnToNormal()`, and `reset()`. These operations must be implemented by the developer. The **AbstractNormalActivityComponent** abstract class should also be extended. This class implements the internal protocol of the iC2C, which is application-independent.

The `handleRequest()` method is responsible for processing service requests. It takes as argument a message corresponding to the request to be executed, and returns a response notification to be delivered to the client component. It is important to note that the framework provides the reusable code which actually sends the response notification and receives the service request. The code responsible for these tasks is implemented by **AbstractNormalActivityComponent**.

If an error occurs during the processing of a service request, an exception is thrown, which may be a failure exception (class **IC2CFailureException**) or an interface exception (class **IC2CInterfaceException**). These are caught by the framework and packaged as exception messages, which are sent to the AbnormalActivity component. It is important to note that the application code only throws language-specific exceptions. Architecture-level exceptions are managed by the framework itself.

In case the handling of a request demands the NormalActivity component to request services from components located in upper levels of the architecture, the **AbstractNormalActivityComponent** class provides a utility method, `requestService()`, which may be used to send synchronous(request/response) requests transparently, upwards the architecture.

The `returnToNormal()` and `reset()` methods are related to the abnormal activity of the iC2C. The former is called when the iC2C has successfully handled an exception, and should return to normal activity. The latter is called when the iC2C is unable handle an exception, and should return to its initial state so that the erroneous state does not affect subsequent requests.

3.3. AbnormalActivity Component

In order to implement an AbormalActivity component, a developer must provide a class that implements the **IAbnormalActivityComponent** interface and extends the **AbstractAbnormalActivityComponent** abstract class. Similarly to the NormalActivity component, the AbnormalActivity component has both application-dependent and application-independent behaviors. **AbstractAbnormalActivityComponent** implements the application-independent behavior of the AbnormalActivity component, while **IAbnormalActivityComponent** specifies the application-dependent behavior.

A single operation is defined by the **IAbnormalActivityComponent** interface: `handleException()`. This operation must be implemented by the developer and defines the exception handler of the iC2C. This operation takes the exception to be handled as argument. If an exception is successfully handled, `handleException()` returns a message object which is sent to the NormalActivity component. Processing is then resumed. Otherwise, an exception is thrown from the body of `handleException()`. This exception is caught by FaTC2 and a failure exception message is sent to the components in the lower levels of the architecture. In case the exception handler for a component which is in the lowest level of an architecture is unable to handle a given exception, it should notify an external user about this fact.

In case the handling of an exception requires the AbnormalActivityComponent to request services from other components, or from the NormalActivityComponent in the same iC2C, class **AbstractAbnormalActivityComponent** provides methods which allow synchronous requests to be carried transparently, similarly to the **AbstractNormalActivityComponent** class.

3.4. iC2C_top, iC2C_bottom and iC2C_internal Connectors

The **IC2CTopConnector**, **IC2CBottomConnector**, and **IC2CInternalConnector** classes are default implementations for the iC2C_top, iC2C_bottom, and iC2C_internal connectors, respectively.

IC2CTopConnector and **IC2CBottomConnector** may be extended in order to implement filtering of notifications in the top domain of an iC2C, or requests in its bottom domain, respectively. A filtering scheme is defined by implementing the `accept()` method in a subclass of **IC2CTopConnector** or **IC2CBottomConnector**. A message *m* is processed only if `accept(`*m*`) == true`.

Subclasses of **IC2CTopConnector** may also implement domain translation in the top domain of the iC2C. The methods `translateIncomingMessage()` and `processOutgoingMessage()` are responsible for this task and are called by FaTC2, respectively, immediately after a message has been *accepted* by the iC2C_top connector, and immediately before a given message is sent by it.

The iC2C_bottom connector is not expected to perform

domain translation. In the C2 architectural style, an element placed in an upper layer of an architecture should make no assumptions about elements in the lower layers[15].

In case no filtering or domain translation is necessary, the default implementations for the iC2C_top and iC2C_bottom connectors may be used.

The **IC2CInternalConnector** class is reused without needing any specialization, since its only task is to route messages inside an iC2C.

4. An Application Example

In order to show the usability of FaTC2, we present a small example extracted from the Mine Pump Control System[13]. The problem is to control the amount of water that collects at the mine sump, switching on a pump when the water level rises above a certain limit and switching it off when the water has been sufficiently reduced. In this section, we describe an implementation for the example application which uses the infrastructure provided by FaTC2.

4.1. Description of the Architecture

The C2 architecture of our example is shown in Figure 4. The **Pump** component commands the physical pump to be turned on/off. Component **LowWaterSensor** signals a notification when the water level is low. **WaterFlowSensor** checks whether water is flowing out of the sump. The **IdealPumpControlStation** component controls the draining of the sump by turning on/off the pump, according to the level of the water in the sump. It includes an exception handler which is executed when the pump is turned on but no water flow is detected. The error handler is implemented by the **AbnormalPumpControlStation** component. The **Pump**, **LowWaterSensor** and **WaterFlowSensor** components have been implemented as simple C2 components, while **IdealPumpControlStation** is an iC2C. In order to build the **IdealPumpControlStation**, five classes are implemented: **NormlPumpControlStation**, **AbnormalPumpControlStation**, **PumpControlStationTop**, **IdealPumpControlStation** and **TranslationConnector**.

Class **NormalPumpControlStation** implements the NormalActivity component of **IdealPumpControlStation**, that is, the methods defined by the **INormalActivityComponent** interface(Section 3.2). Due to the support provided by FaTC2, no messages need to be explicitly sent by any of the methods in **NormalPumpControlStation**; that is, the architect does not need to understand the internal protocol of the iC2C or the way it is implemented.

The **AbnormalPumpControlStation** class implements the exception handler of the **IdealPumpControlStation**. When an exception message is received by the

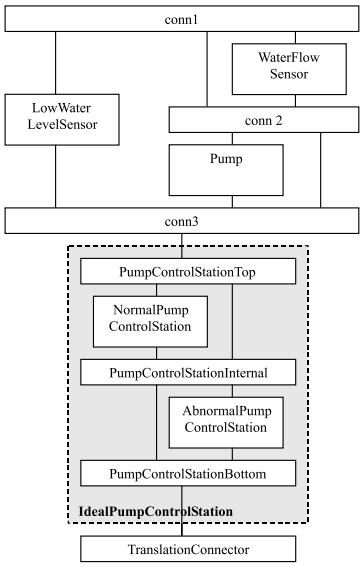

Figure 4. C2 configuration for the Fault Tolerant Mine Pump Control System.

`handleException()` method, the latter keeps sending new requests to **Pump** until either water flow is detected or the maximum number of retries permitted is reached. In the former case, normal activity is resumed(the method simply returns). In the latter, a failure exception message is sent downwards the architecture(the method throws an **IC2CFailureException**). The following code snippet partially illustrates this situation.

```
public Message handleException(Exception e)
 throws Exception {
 (...)
 if(this.retries >= this.MAX_RETRIES) {
   throw new IC2CFailureException(e);
 }
 (...)
```

In order to send an exception message downwards the architecture, the architect should throw a Java exception. In the example above, an exception of type **IC2CFailureException**, a subtype of **Exception**, is thrown.

The **PumpControlStationTop** class provides the **IdealPumpControlStation** component with an extension of the **IC2CTopConnector** class which performs filtering. When a request is issued by the **IdealPumpControlStation**, **PumpControlStationTop** records the type of the request sent, so that only a notification which is a response to that request is allowed to be processed. To build this filtering scheme, two

methods had to be implemented: `accept()` and `processOutgoingMessage()`(Section 3.4).

IdealPumpControlStation is a subclass of **IC2C**. The **IdealPumpControlStation** class defines a public constructor which takes as argument the name of the **IdealPumpControlStation** instance to be created. **TranslationConnector** translates requests and notifications at the bottom interface of the **IdealPumpControlStation** (Figure 4).

It is important to note that *no classes* other than the one which originally implemented the **PumpControlStation** component were modified. Working versions of FaTC2 and the example application can be downloaded at http://www.ic.unicamp.br/~ra014861/FaTC2.

5. Conclusions

Component-based systems built out of reusable software components are being used in a wide range of applications that have high dependability requirements. In order to achieve the required levels of reliability and availability, it is necessary to incorporate into these complex systems means for coping with software faults. In component-based development, source code for the components which make up a system might not be available. This motivates the creation of architectural level fault tolerance mechanisms.

In this work, we have presented FaTC2, an object-oriented framework for the construction of fault-tolerant component-based systems. FaTC2 is an extension of C2.FW, a framework which provides an infrastructure for bulding applications in the C2 architectural style, but lacks support for the construction of fault-tolerant systems. FaTC2 extends C2.FW with a software architectural level exception handling system which is based on the concept of idealised C2 component. We have also presented an example demonstrating how to use FaTC2 to make a fault-tolerant system.

We plan to apply our framework to build a more complex case study where some off-the-shelf components are used. In order to meet this goal, it is necessary to expand the implementations of the NormalActivity and AbnormalActivity components of the iC2C, according to the models proposed by Guerra et al[7, 8], so as to deal with the architectural mismatches[4] which usually arise from the integration of COTS components.

Until the present moment, the iC2C has been modeled as a synchronous entity and the implementation of FaTC2 conforms to this model. That means that an iC2C is unable to handle asynchronous notifications and that requests are issued under the assumption that a response will be eventually received. This restriction might be undesirable for some applications, since a large amount of *glue code* may be necessary if a synchronous iC2C needs to interact with asynchronous components. Hence, another future work for FaTC2 is the implementation of an iC2C for which these restrictions are relaxed.

Finally, we also plan to construct a tool that facilitates the incorporation of exception handling into new and existing applications. We plan to integrate this tool with the ArchStudio environment.

References

[1] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Prentice-Hall, 2nd edition, 1990.

[2] A. Brown and K. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, September/October 1998.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Software Systems*. Addison-Wesley, 1995.

[4] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, 1994.

[5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[6] P. Guerra, C. Rubira, and R. de Lemos. An idealized fault-tolerant architectural component. In *Proceedings of the 24th International Conference on Software Engineering - Workshop on Architecting Dependable Systems*, May 2002.

[7] P. Guerra, C. Rubira, A. Romanovsky, and R. de Lemos. Integrating COTS software componentes into dependable software architectures. In *Proceedings of the 6th ISORC. To Appear.* IEEE Computer Society Press, 2003.

[8] P. A. C. Guerra, C. M. F. Rubira, and R. de Lemos. *Architecting Dependable Systems*, chapter A Fault-Tolerant Architecture for Component-Based Software Systems. Lecture Notes in Computer Science. Springer-Verlag, 2003.

[9] V. Issarny and J. P. Banatre. Architecture-based exception handling. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*. IEEE, 2001.

[10] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of off-the-shelf components in c2-style architectures. In *Proceedings of the 1997 Symposium on Software Reusability*, 1997.

[11] M. Rakic and N. Medvidovic. Increasing the confidence in off-the-shelf components: A software connector-based approach. In *Proceedings of the 2001 Symposium on Software Reusability*, pages 11–18. ACM/SIGSOFT, May 2001.

[12] T. Saridakis and V. Issarny. Fault-tolerant software architectures. Technical Report 3350, INRIA, February 1999.

[13] M. Sloman and J. Kramer. *Distributed Systems and Computer Networks*. Prentice Hall, 1987.

[14] V. Stavridou and A. Riemenschneider. Provably dependable software architectures. In *Proceedings of the Third ACM SIGPLAN International Software Architecture Workshop*, pages 133–136. ACM, 1998.

[15] R. N. Taylor, N. Medvidovic, K. Anderson, J. E. J. Whitehead, and J. Robbins. A component- and message- based architectural style for GUI software. In *Proceedings of the 17th International Conference on Software Engineering*, pages 295–304, April 1995.

[16] UCI. ArchStudio 3.0 homepage. http://www.isr.uci.edu/projects/archstudio.

A Dependable Real-Time Platform for Industrial Robotics

Goran Mustapic, Johan Andersson, Christer Norstrom

ABB Robotics, Västerås, Sweden

goran.mustapic, johan.x.andersson, christer.e.norstrom @se.abb.com

Abstract

Industrial Robots are complex systems with hard real time, high safety, reliability and availability requirements. Robot Controllers are part of these systems and they are complex hard real time computers, which control a robot's mechanical parts. To be useful, Robot Controllers must be programmable by end customers. This is typically done through a domain and vendor specific programming languages.

In this position paper, we will describe some of the architectural challenges we are facing and work we have done, in the process of turning the Robot Controller from an application platform into a dependable platform whose base functionality can be extended by a third party which is not necessarily the end customer.

1. Introduction

Industry demands for safety at work and 60.000 hours of mean time between failures put high demands on the quality of hardware and software of industrial robots. Industrial robots are systems, which consist of a mechanical unit (robot arms that can carry different tools), electrical motors, Robot Controller (computer hardware and software) and clients. Clients are used for on-line and off-line programming of the Robot Controller.

The focus of this article will be the open software architecture of the Robot Controller. According to Issarny [6], in open systems, components do not depend on a single administrative domain and are not known at design time. In this article, we describe a domain specific platform, which faces significant new challenges on the way to become an open platform.

The reason for opening up the controller for third parties is to increase the possibility for partners to provide functionality that ABB Robotics do not either find prioritized or do not have resources for. In a closed platform, the development organization responsible for the platform is the limiting factor. To increase the development speed in the future we can either increase the size of the development organization or open up the system for third party. We believe in the latter, that the number of new types of usages of the robot will increase if we let niche companies adapt the robot for a specific type of applications and customers. The challenges can be divided into:

- developing an appropriate business model,
- determining what type of extensions that should be supported,
- defining an open and dependable architecture,
- defining the certification process and technical details

In this paper we will focus on the technical challenges.

Since we have the system responsibility towards our customers we have to ensure our customers that extensions made by a third party do not have negative side effect on the delivered system. This is a big difference compare to the desktop based systems, where a company that builds a system on top of for example Microsoft Windows© is responsible towards the end customer of the system quality.

The contributions presented in the remainder of this paper are the following:

- We make a short analysis of an industrial robot system and analyze the relevance of the individual dependability attributes for the industrial robot domain.
- We present some initial thoughts on the architectural level reasoning about the open dependable platform for the Robot Controller.
- We present the results of the work to model the platform and enable early reasoning of the architectural level choices.

The outline of the paper is as follows. Section 2 presents a short background about ABB Robot Controller. Section 3 explains the relevance of the individual dependability attributes for our system. In Section 4 we discuss similarities and differences of an open Robot Controller and open desktop systems. Section 5 describes the initial ideas about an open architecture and also the results of an initial case study to model our control system. Finally, in Section 6 we make some conclusions and discuss future work.

2. Background about ABB Robot Controller

The ABB Robot Controller was initially designed in the beginning of the 1990. The requirement was that the same controller should be used for all different types of robots, and thus the architecture is a product line architecture. In essence, the controller has an object-oriented architecture and the implementation consists of approximately 2500 KLOC of C language source code divided on 400-500 classes organized in 15 subsystems. The system consists of three computers that are tightly connected: a main computer that basically generates the path to follow, the axis computer, which controls each axis of the robot, and finally the I/O computer, which interacts with external sensors and actuators.

Only one of those three mentioned computer nodes is open to the end users, and that is the main computer. End users write their programming logic in the form of an imperative language called RAPID. This can be done through off-line programming on a PC, or on-line programming on a client called Teach Pendant Unit (custom hardware).

The system was originally designed to support easy porting to new HW-architectures and new operating systems. There were no initial requirements to have an open architecture. Further, the system was not initially designed to support temporal analysis, because of its closed nature and limited amounts of changes that could only be done by the internal development groups.

3. Discussion of dependability attributes in the context of industrial robots

The analysis of the dependability attributes in this section is done using the terminology presented by Avizienis, Randell and Laprie in [2]. Dependability is described by the following attributes: Reliability, Availability, Safety, Integrity, Confidentiality, and Maintainability.

Security related attributes (confidentiality and integrity), tend to be of less importance for industrial robots as robots tend to be physically isolated, or only connected to a control network together with other industrial devices. Integrity of data which is not security related is very important, as it is unacceptable that e.g. one task in the system causes a hazard situation by damaging the safety subsystem. All other dependability attributes are very relevant.

Even though the contact of humans and robots in industrial environments is restricted (robots work in their cells, which are physically isolated by a fence), safety can never be underestimated because a lack of safety can cause substantial physical damages to the robot equipment and its environment. For example, larger types of robots are powerful machines capable of manipulating a weight of 500 kg. Industrial robots belong to the category of safety-critical systems, which do have a safe state.

Because of the nature of the application, it is crucial to have very high availability and reliability. Unreliability leads to un-availability, which means production stop and huge costs. Because of the complex setup of e.g. car production line, a stop of a single robot leads most often to unavailability of a whole production line. In a complex case, a stop of a single robot can cause up to one day production stop.

Maintainability is important in the sense that it is related to availability. The shorter maintenance time the higher is availability of the system. Ideally, the system should be upgradeable without stopping the production. System upgrades are complicated even for a closed platform, but get much more complicated in a platform which is extendable by a third party, because of the compatibility issues.

When it comes to the dependability threats – fault, error and failures, both hardware and software faults need to be considered. Robot Controller software has both roles of sending control sequences to the hardware as well as predicting preventive hardware maintenance.

There are many different fault-tolerance methods that can be applicable for industrial robots. Error recovery with temporary graceful degradation of performance is not acceptable. A robot either works or it does not work; it cannot make its tasks by working slower depending either on the task (such as arc welding) or because it is a link in the production flow.

4. Towards an open architecture

In the introduction of this paper, we presented some of the motivations for opening up the system. In this section we will try to describe similarities and differences between industrial robots and open desktop platforms where openness is taken for granted. We will conclude

this section by a short analysis about which dependability attributes are most threatened by opening up the system.

4.1. A comparison to Windows© platform

Good examples of open platforms are Microsoft Windows© and perhaps even more Linux operating systems. We shall take the example of Microsoft Windows©, which is closer to our case. It is possible to extend the base platform on three basic different levels: device driver level, win32 programs and .Net applications. This is illustrated in the Figure 1.

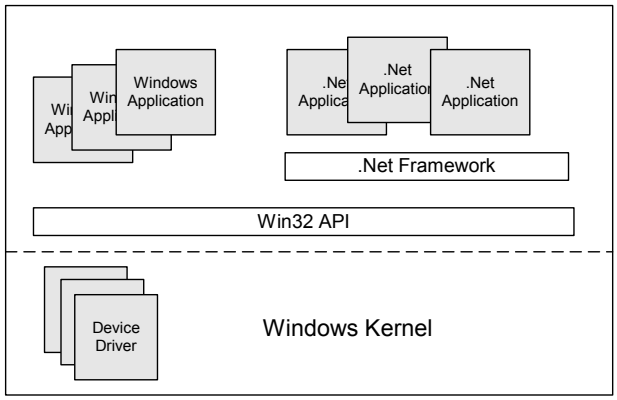

Figure 1: *Different ways to extend Windows© platform*

The architecture of the system guarantees that each of the different extensibility mechanism only can make certain amount of harm to the system, where device drivers can do the most harm and .Net application least harm. Apart from the basic or native ways to extend the platform, many of the applications define their own extensibility mechanisms, e.g. Internet Explorer and SQL Server.

The current way of adding functionality to the Robot Controller corresponds to adding .Net applications to Windows©. As previously mentioned, this is by adding RAPID programs. This is shown in the Figure 2.

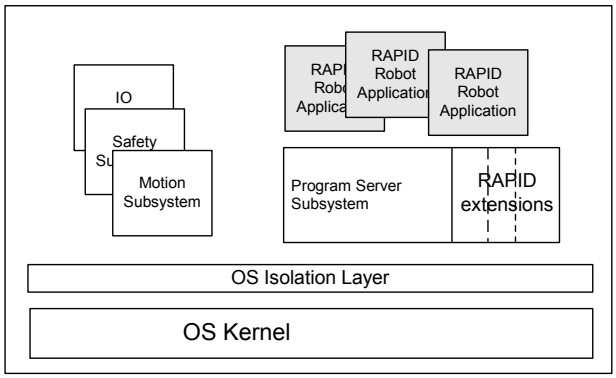

Figure 2: *Current ways to extend the Robot Controller software*

We are considering the following additional ways of extending the Robot Controller:

- Extensions to the robot programming language RAPID
- New subsystems
- Extension logic to the existing subsystems (e.g. fine-tuning of the robot path, new type of IO card, new types of sensors for fine tuning of the robot path etc.)

Let us consider the first type of extensions – RAPID instructions. Basic commands in programming of a robot are "motion" instructions instructing a robot to move its arms to different positions. Some of the basic motion commands are implemented as RAPID extensions and perform their tasks by communicating to the motion subsystem. The basic part of the Program Server contains the engine for executing RAPID programs, and has features like single stepping and executing the program backwards (that is, running the robot backwards). There is a limited set of commands that make the robot programming language. New instructions can be added to enable easier programming and facilitate e.g. very special kinds of tools that a robot is using. This has traditionally been restricted for in-house development because of the harm these extensions can do to the system and prohibitive costs of verifying the correctness of extensions. This harm is equivalent to the harm that unmanaged (native) code can do to a .Net application. This extension code can bring the .Net application down and .Net Framework has no ways to prevent it from doing this. In the Robot Controller case, situation gets more complicated when timing requirements of the Robot Controller are considered.

Other types of extensions mentioned are potentially even more dangerous to the overall system because they most likely require more open access to the lower level services in the system.

4.2. Revisiting the dependability attributes

If we shortly revisit the dependability attributes, which we have analyzed in the section 3, we will see that opening up the architecture will have some significant consequences for several of the attributes. In particular: reliability, availability and safety of the system are threatened by a third party code. Maintainability of the system gets much harder because of the need to handle versioning problems between the platform and the extensions.

Thus we have to find a dependable architecture to support extending the platform in a predictable way.

5. Initial architecture reasoning about open Robot Controller platform

We see the following as the most important architectural goals and also biggest challenges we are facing:

- Defining the dependable platform architecture
- Good support for the development of extensions
- Support for the predictable assembly of extensions and the platform

The platform will provide a Software Development Kit (SDK) for developing the platform extensions. SDK functionality may be grouped with focus on different types of extensions. Support for modeling and simulation will be a part of the SDK, as well as a framework for certification of extensions.

Because timing aspects are crucial in the Real-Time environment, we have already done some initial studies on modeling/simulation of the system, to be able to verify the architectural design in early stages. The rest of the work mentioned below is in an initial phase.

5.1. Platform architecture

The work on architecture for dependable systems is relevant in the context of defining the architecture of an open dependable platform. According to [3] Non-Functional Requirements (NFR) can be divided to: Separation, Additive (a subset of the Separation) and Integral NFR. The classification is based on the way NFR can be taken into account in the system architecture. Additive non-functional requirements (AFNR) are pure add-on components to the architecture, while integral non-functional requirements (INFR) can affect the components of the entire system.

In the case of an open dependable platform, we will need to use means for dependability to create a framework for adding extensions. The choice of the architecture and implementation of the dependability requirements, will lead to this framework. Means for dependability can be transparent, with a different degree of transparency, to extension developers. We believe that for a hard-real time system, extensions of the platform will have to be aware of the platform dependability requirements. An example is the timing requirements.

Some of the existing frameworks for implementing fault-tolerant software present interesting ideas that we may benefit from. An example is a framework for implementing complex fault-tolerant software presented by Xu, Randel and Romanovsky in [13].

It is also important for the software architecture to support good testability, which is a contradictory requirement to fault-tolerance. Example of the work in this field that is relevant for us, is the work done by Voas [9,10], and especially some of the work in the area of distributed real time systems [8].

5.2. Research in Component Based Software Engineering (CBSE)

The extensions of our platform could also be called components. Component-Based Software Engineering (CBSE) and Software Architecture research are much related [4] and experiences from this field can help us in architecting our platform. It is recognized that current Component Technologies handle only syntactic aspects of component compositions, while semantic and especially extra-functional (non-functional) aspects of component specification are open areas of research [4]. One of the biggest challenges of the CBSE is predictable assembly of components and an example of a technology in this research area is called PECT. We believe our ideas are quite inline with the PECT framework presented by the research group at SEI Carnegie Mellon University in the article “Packaging Predictable Assembly” [5]. In this article, prediction-enabled component technology (PECT) is presented, as both a technology and a method for producing the instances of the technology. A PECT instance is created by integrating a software component technology with one or more analysis models. However, focus of their work seems to be more on integration of the existing technology and models, while our focus will be more towards defining a dependable platform and a simple custom component model with good dependability characteristics.

Besides the predictable assembly, research experiences in CBSE can be very useful for handling problems of maintainability and compatibilities between platform and extensions.

5.3. Certification of extensions

An example of a certification process for platform extensions is Microsoft’s WHQL (Windows Hardware Quality Lab [7]) certification program for the device drivers. Some ideas from this process are definitely applicable to our case. We would also need to act as a certification authority. In our case, the situation is more complicated because it is not only the certification of a single component we are concerned about, but also the already mentioned predictable-assembly. One of the possible approaches to certification of COTS software is presented by Voas in [11].

5.4. Model Checking and Simulation

In an open real-time system, with third party components we need a method for extending the base system in a safe way. Since the robot system is very sensitive for timing errors, we have developed a method for analyzing the

impact of adding third party components. In a large system such as the robot controller, this temporal side-effect is hard to predict without models, due to the size and complexity of the system.

Testing and debugging of a complex real-time system is already difficult and when introducing a low-level interface for extending the system with new components makes it even harder. Now, not only the base system has to be verified, but all combinations of extending components that are to be used must be verified as well.

A component might work perfectly, but when it is combined with another component it might introduce an overload situation in some scenarios. This is dangerous since an error caused by this side-effect can be hard to find by testing and affects not only the current task, but may cause a global overload situation, possible delaying several tasks, causing multiple task deadlines to be missed, and finally that the robot fails doing its task properly.

Often the manufacturer of the base system wants to focus on the performance and features of the base system, not integration of special third party components. Letting the component manufacturer be responsible for the extended system is not better and in the industrial robot business, big customers do not accept this. The third party developers will not have access to the source code of the base system, except perhaps an SDK, and will not have the same expertise in the internal structure of the system. Also, they are probably not able to achieve the same quality on their system verification as the base system manufacturer.

A better solution is to let the component developer create a model of their component and let the base system manufacturer certify the component. Their component can through the process be certified for use with the system as well as together with other certified components. This is good for the component developer since it provides a quality label for their software and the base system manufacturer can sell more base systems.

We have developed a method and a prototype tool for describing and analyzing these models. The approach is developed for a robot controller, but the method can be used for other systems as well.

In earlier work, the language-based simulation tool-suite called ART-ML [1,12] was developed, a model creation process has been developed and using it we created a rough model of the controller. A specialized query-language for powerful analysis simulation results as well as data measured on the target system has been developed [12].

So our approach consists of one base step were we develop an initial model of the system and validate that the model represents the modeled system correct. When we have a model of the base system we can add component models to the base model and analyze the consequences of adding a particular set of components. Currently this analysis will be performed off-line but in the future we could do this even on-line.

When creating an initial model $M_{0,}$ of an existing system S_0, several distinct activities are required. These activities are depicted in Figure 4. First the structure has to be identified and modeled, i.e. the tasks in the system and synchronization and communication among them. In the next step, we measure the system and populate the structural model with data about the temporal behavior. Moreover, information needed in the validation phase is collected, e.g. response times. When tuning the model, the initial model M_0 is compared with S_0 by simulating the model and comparing the results with the validation data collected in the previous step. In this step it is possible to introduce more details about the tasks behavior in order to capture the system's behavior accurately.

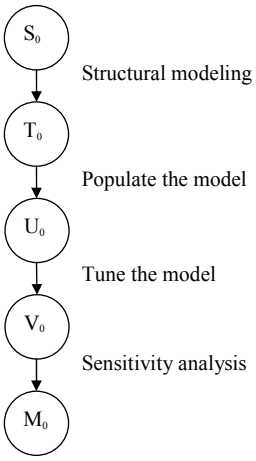

Figure 4: *The model creation process*

To validate the usefulness of the model it is necessary to perform a sensitivity analysis. The sensitivity analysis should be based on foreseen potential extensions or changes in the particular system. The extensions are introduced in the model as well as in the system and the new systems are compared. Any divergence between the behavior of the simulated model and the system indicates that more details must be introduced in the model. This increases the confidence in the created model.

In the robot controller we have studied, the following typical changes were identified:

- Change existing behavior of a task which results in changes in the execution time distribution.
- Add a task to the system.
- Change the priority of an existing task.

When a third party adds a component to the system a model of the added component has to be provided. This model is composed with all other added components' models and the basic system model. Based on this composed model, we can verify if the defined properties still hold for that particular combination of components, and if so draw the conclusion that the added components do not affect the system behavior badly.

6. Future Work and Conclusions

We intend to continue our work in the direction of defining architecture of a reliable, safe and maintainable platform. We will also continue working on the modeling and simulation to support the predictable assembly of the platform and extension components.

From the initial analysis, which we have presented in this paper, it can be seen that we will need to use experiences from multiple areas of research and to combine them to create an optimal domain specific solution.

Besides the technical challenges, there are also significant business challenges around the certification process. Without proper architecture, tool support etc, costs of certification may turn out to be larger than the benefits of having an open platform.

7. References

[1] Andersson J., Neander J., Timing Analysis of a Robot Controller, Master Thesis, Mälardalen University, Sweden, http://www.mdh.se/, 2002.

[2] Avizienis A., Randell B., and Laprie J.C., "Fundamental Concepts of Computer System Dependability", IARP/IEEE RAS Workshop On Robot Dependability, 2001.

[3] Brandozzi M. and Perry E.D., "Architectural Prescriptions for Dependable Systems", ICSE 2002 Workshop on Architecting Dependable Systems, 2002.

[4] Crnkovic I., Hnich B., Jonsson T., and Kiziltan Z., Specification, Implementation and Deployment of Components, *Communications of the ACM*, volume 45, issue 10, 2002.

[5] Hissam S., Stafford J., Wallnau K., and Moreno G., "Packaging Predictable Assembly", Proceedings of the First IFIP/ACM Working Conference on Component Deployment, Berlin, Germany, 2002.

[6] Issarny V., "Software Architectures of Dependable Systems: From Closed To Open Systems", ICSE 2002 Workshop on Architecting Dependable Systems, 2002.

[7] Microsoft Corporation, *Windows Hardware Quality Lab homepage*, http://www.microsoft.com/hwdq/hwtest/, 2003.

[8] Thane H., "Monitoring, Testing and Debugging of Distributed Real-Time Systems", Doctoral Thesis, Royal Institute of Technology, KTH, Mechatronics Laboratory, TRITA-MMK 2000:16, Sweden, 2000

[9] Voas J., "Factors That Affect Software Testability", Pacific Northwest Software Quality Conference, Inc., 1991.

[10] Voas J., *Software Assessment - Reliability, Safety, Testability*, John Wiley & Sons, Inc, 1995.

[11] Voas J., A Defensive Approach to Certifying COTS Software, report RSTR-002-97-002.01, Reliable Software Technologies Corporation, 1997.

[12] Wall A., Andersson J, and Norstrom C., "Probabilistic Simulation-based Analysis of Complex Real-Time Systems", will appear in the 6th IEEE International Symposium on Object-Oriented Real-time Distributed Computing, Hakodate Hoikado, Japan, 2003.

[13] Xu J., Randell B., and Romanovsky A., "A Generic Approach to Structuring and Implementing Complex Fault-Tolerant Software", nr 5th ISORC'02, IEEE Computer Society, 2002.

A Framework for Using Component Redundancy for self-Optimising and self-Healing Component Based Systems

Ada Diaconescu[*], John Murphy[*]
Performance Engineering Laboratory, Dublin City University
{diacones,murphyj}@eeng.dcu.ie

Abstract

The ever-increasing complexity of software systems makes it progressively more difficult to provide dependability guarantees for such systems, especially when they are deployed in unpredictably changing environments. The Component Based Software Development initiative addresses many of the complexity related difficulties, but consequently introduces new challenges. These are related to the lack of component intrinsic information that system integrators face at system integration time, as well as the lack of information on the component running-context that component providers face at component development time.

We propose an addition to existing component models, for enabling new capabilities such as adaptability, performance optimisation and tolerance to context-driven faults. The concept of 'component redundancy' is at the core of our approach, implying alternate utilisation of functionally equivalent component implementations, for meeting application-specific dependability goals.

A framework for implementing component redundancy in component-based applications is described and an example scenario showing the utility of our work is given.

1. Introduction

Extensive employment of software systems in various domains raised the concern for the dependability guarantees provided by such systems (e.g. performance, reliability, robustness). Nevertheless, the ever-increasing size and complexity of modern software systems leads to more complicated and expensive system design, testing and management processes, decreasing system flexibility and making it difficult to control dependability characteristics of such systems [1].

In this context, Component Based Software Development (CBSD) has emerged as a new solution that promises to increase the reliability, maintainability and overall quality of large-scale, complex software applications. In the CBSD approach, software applications are developed by assembling made or bought (i.e. commercial off-the-shelf - COTS) components, according to a well-defined software architecture.

Consequently, the dependability of component-based software applications is determined by both the dependability of the individual components involved, as well as by the adopted software architecture. Considerable research efforts towards determining optimal software architectures ([2], [3], [4]) with respect to the system quality attributes [5], as well as towards achieving dependability guarantees for COTS components ([6], [7]), support this idea. The impact software architecture has on the overall software system performance is also demonstrated in [8]. In this paper, it is shown how different software architectures, providing the same functionalities, yielded different performance results while running in identical environmental conditions.

Information on the context in which a software component or application will run (e.g. hardware and software resources, workloads and usage patterns) is vital when taking architectural, or design decisions. At software development or integration time though, it is impossible to predict with sufficient accuracy, the environmental conditions in which software components or applications will be deployed. In addition, the initial deployment conditions can dynamically change at runtime. Using COTS components exacerbates the problem, by increasing the level of indetermination and making it hard to provide dependability guarantees for the running system [6], [7].

Assuming that different architectural, design and implementation-related choices proved optimal in different environmental circumstances, we argue that it would be beneficial for system quality if the software application could accordingly adapt at runtime, when accurate information were available. We propose the use of redundancy in order to enable such capabilities for component-based software systems. Our intent is to enhance one of the existing component platforms (e.g. EJB, .NET, or CCM) with support for software component redundancy. The predicted benefits of this approach include constant, automatic performance optimisation for running applications, as well as tolerance to certain categories of non-functional, integration-specific faults (e.g. deadlocks, data corruption). By non-

[*] The authors' work is funded by Enterprise Ireland Informatics Research Initiative 2001

functional faults, we mean faults that are not related to an application's expected functionality and therefore do not imply any application-specific behavioural knowledge or extra implementation effort to detect.

The rest of the paper is structured as follows. Section 2 provides an overview of our research proposal. An example scenario, indicating the benefit of our work, is presented in Sections 3. A general architecture for our proposed framework is described in Section 4. Section 5 places our approach in the context of similar work in the area. We conclude and present future work in Section 6.

2. Research overview

Our research goal is to enable dynamic adaptability capabilities in complex, component-based software systems, running in unpredictably changing environments, in order to automatically optimise and maintain their dependability characteristics.

Central to our solution is the concept of (software) *component redundancy*. By this concept, we mean that a number of component implementation variants, providing the same or similar services, are available at runtime. We refer to these component variants as *redundant components* and say that a set of redundant components providing an equivalent service constitutes a *redundancy group* (with respect to that service). Any component variant in a certain redundancy group can be functionally replaced with any other component variant in the same redundancy group.

Only one of the redundant components providing a service is assigned, at any moment in time, for handling a certain client request for that service (i.e. an instance of that component is forwarded the client request). This differs from other approaches (e.g. N-version programming; agent-based systems [9]), where a number of the available redundant variants work in parallel, towards a common result. We refer to a component variant that the application is currently using (i.e. sending client requests to instances of that component version) as an *active* component variant. Component variants that are not currently considered for handling client requests are referred to as *passive* component variants.

If instances of an active component variant fail, or perform poorly in a certain context, the component variant can be *deactivated* and replaced with an alternative member of the same redundancy group. This is the main means by which redundancy groups continually optimise themselves, while dealing with changing execution contexts, or context-driven faults.

We do not constrain the component redundancy concept to the level of atomic components [10] [Figure 1-a]. This concept can also be applied to composite components [Figure 1-b] (i.e. composites [10], 'containing' a number of sub-components) or to component sets, or groups (i.e. components 'using' other components) [Figure 1 – c]. Therefore, through the rest of the paper, references to redundant components can imply atomic, composite, or sets of components.

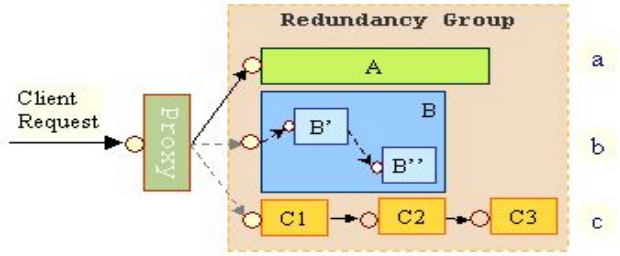

Figure 1: Redundancy granularities

We intend to implement all functionalities that are required to support and benefit from component redundancy at the component platform level. No development effort overhead is to be placed on the developers of software components that are to be deployed and run on such platforms. Of course, in order for redundancy to be enabled, alternative variants would have to be provided. However, our approach does not require that multiple redundant components be available at software application deployment or runtime. The only constraint is that at least one component version must be available for each external interface, at all times. While complying with this constraint, redundant components can be dynamically added or removed from the software system, at runtime.

We propose that a formal component description be available for every deployed component variant. The description includes information on both functional (e.g. provided and required services) and non-functional (e.g. quality attributes, recommended resources) characteristics of the component (e.g. similar to contracts as in [10], or [11]). Most system quality characteristics depend upon the execution context (e.g. response time is influenced by workload and available resources). These variations are represented in component descriptions as a list of [environment related parameters, corresponding values] pairs. Initially, component non-functional characteristics can optionally be provided by component developers, based on estimations, test results, or previous experience with the supplied components. While a component variant is active, its initial quality description is updated with run-time monitoring information, for the precise application configuration and execution environment.

3. Example

In this section, we provide an example of a possible scenario in which our approach proves to be beneficial. For this example, we opted for the EJB component

technology. However, we believe our framework is generic enough to be applied to other component models.

The example involves two different component implementations providing the same functionality: repeatedly retrieving information from a remote database. The two components differ at the design level. The first design variant involves a single Session Bean, containing SQL code for directly accessing the database. We will refer to this variant as the Direct DB variant. In the second design variant, a Session Bean uses an Entity Bean as means of interacting with the database. We will refer to this variant as the Using Entity Bean variant. A client Session Bean is used for calling these two variants, repeatedly requesting information.

We deployed our EJB example on an IBM WebSphere application server, on Windows2000, running on an Intel Pentium4, with 1.6GHz CPU and 512 MB RAM. We used a DB2 database, running on Windows2000, Intel Pentium 4, 1.6 GHz CPU and 256 MB RAM. A third machine was used for generating traffic and loading the network link to the remote database, to various degrees. We used the Tfgen traffic generator for this purpose. The three machines were connected through a switched 100 Mbps Ethernet LAN, completely separated from other traffic.

We measured the response delays for each version, in different environmental conditions (i.e. available bandwidth on the network links) and usage patterns (i.e. number of repetitive read requests per client transaction).

When the network is lightly loaded, we experience smaller delays in the Direct DB variant than in the Using Entity Bean variant, regardless of the number of repetitive client requests (e.g. 1, 10, 100, 1000 [requests per transaction]). This can be accounted for by the overhead incurred (in the Using Entity Bean variant) by the extra inter-process communication and Entity EJB management.

However, increasing the load on the network link to the remote database has significant impact on the Direct DB approach, while hardly affecting the Using Entity Bean. This can be explained by the fact that the Direct DB variant needs to access the database for each individual (client) read request. The Using Entity Bean variant, involves a single database access per client transaction (i.e. only for the first read request in the transaction), as the data is then locally stored at the Entity Bean instance level and retrieved from there for subsequent requests. Therefore, for increased network loads (e.g. 90% load) and number of read requests, the Direct DB design choice produces higher delays than the Using Entity Bean does. Using an Entity Bean to read from the database becomes, in these circumstances, the optimal choice.

The optimal variant switching point between the two implementations is reached when the inter-process communication and CPU overhead (i.e. in the Using Entity Bean variant) is exceeded by the repeated remote database access overhead (i.e. in the Direct DB variant). Figure 2 shows the response-time curves corresponding to the two redundant variants, for various network loads, when 1000 read requests were made per client transaction. For obtaining these curves, we repeatedly measured the response delays of such repetitive client requests, for different network loads. We then calculated the average delay value, for each network load.

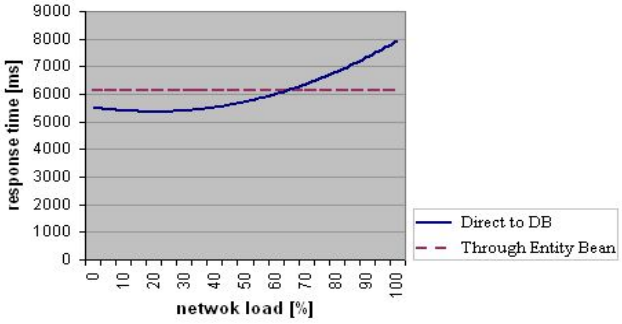

Figure 2: response-time curves

Even though simple, this example shows how alternating the activation of two redundant variants can ensure better performance than either variant could, at all times. We argue that it is hard, if not impossible to devise a component version that exhibits optimal characteristics in all possible running contexts. The optimal component variant depends on the component execution environment, which can frequently change. Our focus is on the adaptation logic for automatically determining optimal component variants and optimal combinations of component variants, in different running contexts.

4. Framework general architecture

We propose implementing component redundancy as a new service provided by component platforms (i.e. besides already provided services, such as security, transaction support, or life-cycle management). Three main functionalities were identified as needed for the support, utilization and management of redundant components and were associated with three logical tiers in our framework [Figure 3]: i) Monitoring tier; ii) Evaluation tier and iii) Action tier. In this section, we present the main roles and functionalities of each of these tiers and discuss the way they interact in order to provide the component redundancy service.

The Monitoring tier is concerned with acquiring run-time information on the software application as well as on its execution environment. Run-time monitoring implies that information is collected exclusively for the active component variants. It is also the responsibility of the Monitoring tier to analyse the collected information and identify any potential 'problem' components [1], [12].

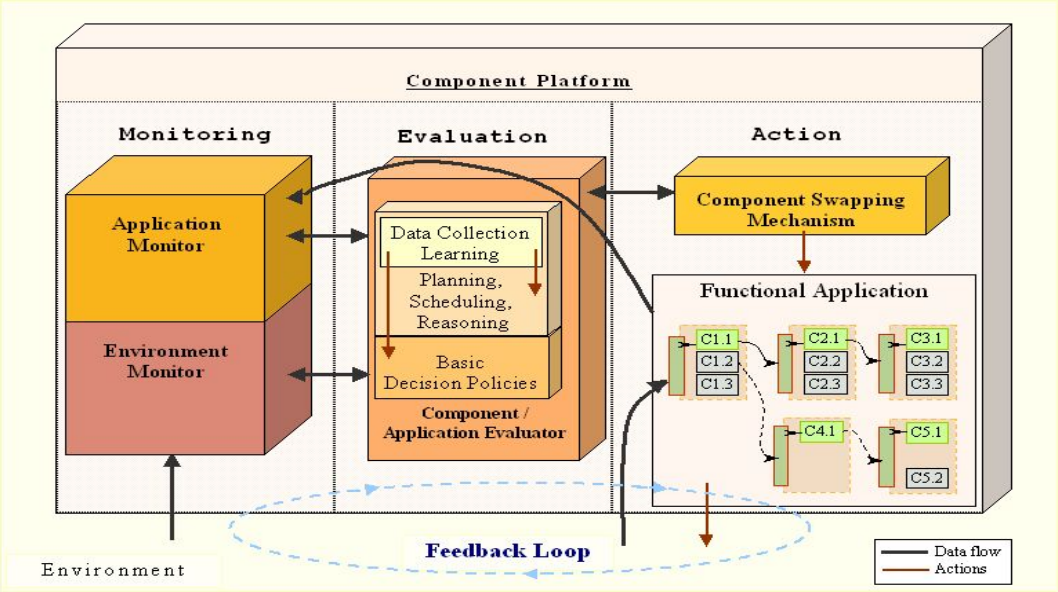

Figure 3: Framework architecture

The Evaluation tier is responsible for determining the optimal redundant component variant(s) in certain contexts, using adaptation logic, component descriptions and monitoring information on the current environment and application state. It also updates the descriptions of active component variants, with runtime information from the Monitoring tier. This helps the Evaluation tier to 'learn' in time about the performance characteristics of the software application it has to manage.

Adaptation logic, for deciding which redundant component(s) to activate (and deactivate respectively), is reified in the Evaluation tier in the form of decision policies. These are sets of rules, dictating the actions to be taken in case certain conditions are being satisfied. Decision policies can be customised for each deployed application (e.g. requested quality attributes values, default redundant components to activate) in order to serve the specific application goals and can be dynamically added, modified or deleted at runtime.

We split decision policies into two layers, based on their complexity. The bottom layer comprises basic decision policies, of the condition-action type. These policies are used to remedy poor performance or critical situations (e.g. response time thresholds are being exceeded) and take immediate effect. The top layer is reserved for decision policies concerned with application optimisations, in conditions in which the application is not necessarily evaluated as under-performing or faulty. These policies are designed for activities such as reasoning, predicting, planning, or scheduling, in order to optimise application performance, anticipate and prevent failures or emergencies. Policies in the top layer are also used to control the adaptation process. They can decide when to stop an optimisation evaluation or enforcement operation, in case it becomes too costly (e.g. in time, or resources), or it seems to have entered an infinite loop (e.g. oscillating state, chain reaction).

The Action tier encompasses the actual software application and a component-swapping mechanism. Based on optimisation decisions, the Evaluation tier sends corresponding configuration commands to the Action tier, indicating the redundant component variant(s) to be activated or deactivated respectively. The component-swapping mechanism performs the requested operations. As stated in related research on component hot swapping, two main issues occur when replacing component variants at runtime. One issue is concerned with state transfer from an executing component instance to a replacement component instance. This is only needed in case instances of different component variants handled the same client request or session, one after the other. Since in the targeted problem domain client calls are usually short-lived, we believe such action would bring little performance benefit to requests already being handled (when component replacement occurred). Therefore, in a first phase, we do not attempt to transfer state between instances of different component variants. Rather, incoming client calls are directed to an instance of the appropriate component variant, upon arrival. Instances of component variants to be deactivated finish handling current requests before being removed. This allows for instances of different redundant components to coexist. In a future phase, we will consider one of the solutions proposed in the ongoing research in this area (e.g. [13],

[14]). The other issue is maintaining client references consistency. We adopt a proxy-based solution to address this issue. Component technologies based on contextual composition frameworks [10] provide a straightforward way of implementing this. That is because clients can only call component instances through the component container, in which the component was deployed and run. The component container can consequently be modified so that to transparently (re)direct client requests to instances of active component variants. In brief, in a first phase of our research, we adopt a client request indirection strategy for implementing the component hot-swapping mechanism.

In our framework, the three presented tiers operate in an automated, feedback-loop manner [Figure 3]: the application performance is monitored and evaluated, the optimal redundant component(s) are identified and activated and the resulting application is monitored and (re-)evaluated. Decision policies at both layers can be dynamically tuned in effect. It is important to note that as these are logical tiers, the boundaries between them may not be as clearly marked when implemented.

4.1. Hierarchical adaptation mechanism

When considering large-scale component-based applications, global optimisations may not always be desirable. Evaluating an overall application, potentially consisting of hundreds of components, whenever an individual component or a group of components does not behave as expected, might induce unnecessary overhead and not scale well. We propose distributing the adaptation mechanism. That is, if a problem is detected at an individual component level, the problem is dealt with locally, by means of redundant component replacement. Nevertheless, exclusively concentrating on local optimisations might not globally optimise the system. Therefore, our framework employs (three-tiered) adaptation mechanisms with different scopes (e.g. local, group, global), organised in a hierarchal manner. Detected problems can be dealt with locally or/and signalled upwards the hierarchical tree, up to the global level. Adaptation mechanisms can be dynamically activated or deactivated, in order to reduce overhead, when possible. This idea is also presented in [12], in the context of non-intrusive, EJB system monitoring.

5. Related Work

Redundancy for increased robustness or reliability has been successfully used in various domains (e.g. hardware, mechanics, or constructions). The same concept was introduced in the software domain (e.g. [9], or as ‘design diversity’ in [6], [15]), in order to achieve fault-tolerance capabilities for software systems. A few examples of fault tolerant schemes implementing this concept are N-version programming, N self-checking software, recovery blocks [16], or exception handling approaches. However, as these schemes target system fault tolerance, they imply both the presence of knowledge of the correct system behaviour, as well as of methods for assessing system behaviour at runtime, in order to detect faults. We target a different problem domain, encompassing performance-related problems and non-functional faults, which can generally be detected without needing application semantics information. Our framework can consequently be implemented as part of the component platform layer, for the benefit of all applications deployed on such platforms.

Similar to our performance optimisation related intent, the Open Implementation initiative [17] allows clients to decide which implementation variant to use (i.e. instantiate) for optimal performance, in a specific context. We propose that the component platform automatically take such decisions. In our view, it is very expensive, or even impossible for a system manager to optimally perform such activities in due time, in the case of complex systems or frequent environmental changes.

Redundancy as a means of achieving dependability for Internet systems (i.e. Web Services based) is proposed in the RAIC [13] project. The addressed problem domain in this case however, is different in scope from our work. This is because such systems rely on Internet services offered by different providers, from different locations. No single authority owns, or has complete control over the entire system. The Internet system developer has no knowledge of, or access to the implementation, deployment platform, or supporting resources of the services it needs to use. Redundancy support cannot be implemented in this case at the component deployment platform level. Instead, redundancy support for the services that Internet systems use is implemented at the software application level of such systems.

Research in the area of dynamic component versioning presents certain similarities to our work. However, the main intent of the two research directions is different, emphasising different aspects. Component versioning is concerned with verifying whether new versions are better than old ones, before dynamically upgrading the system. In [14] for example, the best component version is determined by means of online testing. Even though the possibility of multiple versions being kept is considered, the way such versions are to be used is not elaborated.

A significant research area, closely related to our work, is concerned with specifying and building dynamic adaptability capabilities for self-repairing systems. Mostly related to our work are approaches based system architectural models [18], [19]. A feedback-loop

mechanism (separated from system business logic) is employed for adapting running systems to changing requirements, or environmental conditions. This mechanism is designed in a centralised manner. Monitoring information is centralised, evaluated using analytical methods (e.g. queuing theory) and the system is globally optimised. Our approach adopts a hierarchical adaptation approach, where global system optimisation can generally be avoided. We focus on adaptation operations related to redundant component replacements.

An important aspect of our research is the fact that we exclusively target component-based applications based on contextual composition frameworks [10]. The unique nature of such applications (e.g. soft inter-component bindings; unpredictable number of component instances) might make approaches devised for component-based systems in general (i.e. in which 'components' can mean clients, servers, or software modules), difficult to apply.

6. Conclusions and Future Work

This paper proposed the use of component redundancy for enabling self-optimisation, self-healing and dynamic adaptation capabilities in component-based software systems. A component redundancy related terminology was defined. We argued that system complexity, lack of sufficient information and changing execution conditions make it impossible to create and ascertain components that exhibit optimal dependability characteristics at all times. An example was presented to support this idea. In this example, different strategies were selected for implementing two distinct component variants providing the same functionality. Each implementation variant proved optimal (with respect to response delays) in different environmental conditions. As these results indicate, knowledgeably alternating the usage of redundant components, optimised for different running contexts, ensures better overall performance than either component variant could provide.

A framework for implementing the component redundancy concept was described. We identified the main roles and functionalities this framework needs to provide and categorised them into three logical tiers: monitoring, evaluation and action. We proposed distributing the three logical tiers, organising them (each) in a hierarchical manner, in order to reduce overhead.

As future work, we intend to provide a proof-of-concept implementation of our framework and test it against our example scenario. In addition, further scenarios and case studies will be identified and documented. The cost of acquiring multiple redundant components, as well as the impact of using redundant components on the overall application performance and resource usage will have to be analysed.

7. References

[1] J. O. Kephart, D. M. Chess, "The Vision of Autonomic Computing", IEEE Computer, January 2003

[2] C. U. Smith, L. G. Williams, "Software Performance Engineering: A Case Study with Design Comparisons", IEEE Trans. Software Eng., Vol. 19, No 7, July 1993

[3] F. Aquilani, S. Balsamo, P. Inverardi, "Performance Analysis at the Software Architectural Design Level", Performance Evaluation, Volume 45, Number 2-3, July 2001

[4] J. Bosch, P. Molin, "Software Architecture Design: Evaluation and Transformation", IEEE Conference and Workshop on Engineering of Computer-Based Systems, Nashville, Tennessee, March 1999

[5] M. Klein et al., "Attribute-Based Architecture Styles", in Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, 1999, pp 225-243

[6] P. Popov, L. Strigini, A. Romanovsky, "Diversity for Off-The-Shelf Components", International Conference on Dependable Systems&Networks, NY, USA, 2000, pp. B60-B61

[7] P. A. C. Guerra, C. M. F. Rubira, R. de Lemos, "An Idealized Fault-Tolerant Architectural Component", Workshop on Architecting Dependable Systems, Orlando, FL, May 2002

[8] E. Cecchet et al., "Performance and Scalability of EJB Applications", Proc of 17th ACM Conference on Object-Oriented Programming, Seattle, Washington, 2002, pp 246-261

[9] M.N. Huhns, V.T. Holderfield, "Robust Software", Agents on the Web, IEEE Internet Computing, March/April 2002

[10] C. Szyperski, with D. Gruntz and S. Murer, "Component Software: Beyond Object-Oriented Programming", Second Edition, Addison-Wesley Pub Co, 1 November 2002

[11] B. Meyer, C. Mingins, H. Schmidt: *Trusted Components for the Software Industry*. IEEE Computer 5/1998, pp. 104-105

[12] A. Mos, J. Murphy, "Performance Management in Component-Oriented Systems using a Model Driven Architecture Approach", The 6th IEEE International Enterprise Distributed Object Computing Conference (EDOC), Lausanne, Switzerland, September 2002

[13] C. Liu, D. J. Richardson, "RAIC: Architecting Dependable Systems through Redundancy and Just-In-Time Testing", ICSE, Workshop on Architecting Dependable Systems (WADS), Orlando, Florida, 2002

[14] M. Rakic, N. Medvidovic, "Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach", Symposium on Software Reusability: putting software reuse in context, Toronto, Ontario, Canada, 2001

[15] B. Littlewood et al., "Modeling software design diversity: a review", ACM Press, New York NY, USA, 2001, pp 177-208

[16] B.Randell and J.Xu, "The Evolution of the Recovery Block Concept", Software Fault Tolerance, JohnWiley&SonsLtd, 1995

[17] G. Kiczales, "Beyond the Black Box: Open Implementation", IEEE Software, January 1996

[18] S. Cheng et al., "Using Architectural Style as a Basis for Self-repair", Proc. Working IEEE/IFIP Conference on Software Architecture, Montreal, August, 2002

[19] P. Oriezy et al., "An Architecture-Based Approach to Self-Adaptive Software", IEEE Intelligent Systems, May/June 1999, p. 54-62

Elements of the Self-Healing System Problem Space

Philip Koopman
Institute for Software Research, International
& ECE Department
Carnegie Mellon University
Pittsburgh, PA, USA
koopman@cmu.edu

Abstract

One of the potential approaches to achieving dependable system operation is to incorporate so-called "self-healing" mechanisms into system architectures and implementations. A previous workshop on this topic exposed a wide diversity of researcher perspectives on what self-healing systems really are. This paper proposes a taxonomy for describing the problem space for self-healing systems including fault models, system responses, system completeness, and design context. It is hoped that this taxonomy will help researchers understand what aspects of the system dependability problem they are (and aren't) addressing with specific research projects.

1. Introduction

Self-healing systems form an area of research that is intuitively appealing and garnering increased attention, but not very well defined in terms of scope. At the 2002 Workshop on Self-Healing Systems [WOSS02], it became clear that researchers have differing views on what comprises research on self-healing systems. This paper attempts to document those views in the form of a description of the self-healing systems research problem space.

There is a rich set of existing knowledge on the general topic of dependable systems, and on techniques that can reasonably be considered to comprise "self-healing." For example, one view of self-healing systems is that they perform a reconfiguration step to heal a system having suffered a permanent fault. The use of standby spares in such a manner has been called "self-repair" [Bouricius69]. Systems that use modular redundancy (e.g., [vonNeuman56]) can tolerate component failures and might be considered to be self-healing.

It is premature to propose a consensus-based definition of the term "self-healing," so we do not attempt to do this beyond an appeal to intuition that such a system must somehow be able to "heal" itself. Whether this means that self-healing systems are really a subset of traditional fault-tolerant computing systems is unclear. However, the topic of self-healing systems has attracted a number of researchers who would not otherwise have been involved in the fault tolerant computing area. So, if nothing else, the label of self-healing has broadened the pool of researchers addressing the difficult problems of creating dependable systems.

To give researchers in this area a common basis for defining the scope of self-healing systems research, it seems worthwhile to set forth a description of issues being addressed by various research projects. This might provide a way for researchers to realize they mean considerably different things by their use of the phrase "self-healing," as well as to understand the similarities and differences in their approaches and domains. Toward that end, this paper attempts to describe the general problem space relevant to self-healing system research.

2. Elements of the model

Based on our experiences and observations at the WOSS02 workshop, we propose that there are four general categories of aspects to the self-healing system problem space: fault model, system response, system completeness, and design context (Table 1). (The particular categories are not important, but simply form a way to group related concepts for the purposes of discussion.) We shall discuss the elements of each category in turn.

[Avizienis01] contains an extensive taxonomy of fault tolerant computing terminology and approaches. We use this as the basis for terminology, and as the basis for the fault modeling portion of the taxonomy.

2.1. Fault model

Self-healing systems have similar goals to the general area of dependable computer systems. (Not all dependable computing research areas are "self-healing", but one can argue that all "self-healing" techniques ultimately are dependable computing techniques.)

One of the fundamental tenets of dependable computing is that a *fault hypothesis* (often called a fault model) must be specified for any fault tolerant system. The fault hypoth-

Table 1. Problem space model elements.

Fault model:
Fault duration
Fault manifestation
Fault source
Granularity
Fault profile expectations
System response:
Fault Detection
Degradation
Fault response
Fault recovery
Time constants
Assurance
System completeness:
Architectural completeness
Designer knowledge
System self-knowledge
System evolution
Design context:
Abstraction level
Component homogeneity
Behavioral predetermination
User involvement in healing
System linearity
System scope

esis answers the question of what faults the system is to tolerate. (If one doesn't know what types of faults are to be tolerated, it is difficult to evaluate whether a given system is actually "fault tolerant.")

In a similar vein, self-healing systems must have a fault model in terms of what injuries (faults) they are expected to be able to self-heal. Without a fault model, there is no way to assess whether a system actually can heal itself in situations of interest. The following are typical fault model characteristics that seem relevant.

Fault duration: Faults can be permanent, intermittent (a fault that appears only occasionally), or transient (due to an environmental condition that appears only occasionally). Since it is widely believed that transient and intermittent faults outnumber permanent faults, it is important to state the fault duration assumption of a self-healing approach to understand what situations it addresses.

Fault manifestation: Intuitively, not all faults are as severe as others. Beyond that, components themselves can be designed to exhibit specific characteristics when they encounter faults that can make system-level self-healing simpler. A common approach is to design components that are fail-fast, fail-silent. However, other systems must tolerate Byzantine faults which are considered "arbitrary" faults. (It is worth noting that Byzantine faults exclude systematic software defects that occur in all nodes of a system, so the meaning of "arbitrary" is only with respect to an assumption of fault independence.)

Beyond the severity of the fault manifestation, there is the severity of how it affects the system in the absence of a self-healing response. Some faults cause immediate system crashes. But many faults cause less catastrophic consequences, such as system slow-down due to excessive CPU loads, thrashing due to memory hierarchy overloads, resource leakage, file system overflow, and so on.

Fault source: Assumptions about the source of faults can affect self-healing strategies. For example, faults can occur due to implementation defects, requirements defects, operational mistakes, and so on. Changes in operating environment can cause a previously working system to stop working, as can the onset of a malicious attack. While software is essentially deterministic, there are situations in which it can be argued that a random or "wear-out" model for failures is useful, suggesting techniques such as periodic rebooting as a self-healing mechanism. Finally, some self-healing software is designed only to withstand hardware failures such as loss of memory or CPU capacity, and not software failures.

Granularity: The granularity of a failure is the size of the component that is compromised by that fault. (The related notion of the size of a fault containment region is a key design parameter in fault tolerant computers.) A fault can cause the failure of a software module (causing an exception), a task, an entire CPU's computational set, or an entire computing site. Different self-healing mechanisms are probably appropriate depending on the granularity of the failures and hence the granularity of recovery actions.

Fault profile expectations: Beyond the source of the fault is the profile of fault occurrences that is expected. Faults considered for self-healing might be only expected faults (such as defined exceptions or historically observed faults), faults considered likely based on design analysis, or faults that are unexpected. Additionally, faults might be random and independent, might be correlated in space or time, or might even be intentional due to malicious intent.

2.2. System response

The first step in responding to a fault is, in most cases, actually detecting the fault. Beyond that there are various ways to degrade system operation as well as attempt recovery from or compensation for a fault. Each application domain has extra-functional aspects that are important, such as reliability, safety, or security. These extra-functional concerns influence desired system responses.

Fault Detection: Fault detection can be performed internally by a component, by comparing replicated components, by peer-to-peer checking, and by supervisory checks. Additionally, the intrusiveness of fault detection can vary from nonintrusive testing of results, to execution of audit or check tasks, redundant execution of tasks,

on-line self-test, and even periodic reboots for the purpose of more thorough self tests. Systems might inject faults intentionally as on-line tests of fault detection mechanisms. A related area is that of ensuring that all aspects of a system are activated periodically so that any latent accumulated faults can be detected within a bounded time. Not all systems can achieve 100% fault detection in bounded time.

Degradation: Self-healing systems might not restore complete functionality after a fault. The degree of degraded operation provided by a self-healing system is its resilience to damage that exceeds built-in redundancy. Some systems must fail entirely operational (i.e., cannot fulfill their mission without full functionality). But many systems can degrade performance, shed some tasks, or perform failover to less computationally expensive degraded mode algorithms.

Fault Response: Once a fault has been detected, the system must select a response mechanism. Typical on-line responses include masking a fault (e.g., modular redundancy that performs a majority vote of independent computational results), rollback to a checkpoint, rollforward with compensation, or retrying an operation using the original or alternate resources. Heavier-weight responses include system architectural reconfiguration (on-the-fly or involving a reboot), invoking alternate versions of tasks, killing less important tasks, and requesting assistance from outside the system. The fault response might be optimized to maintain desired properties such as correctness, quality of service contracts, transactional integrity, or safety. Fault responses might also be preventative (such as a periodic system reboot), proactive (such as an action triggered by a burst of faults which were tolerated but are indicative of a possible near-term failure), or reactive.

Recovery: After a system has detected a fault, potentially degraded, and invoked a fault response, it must recover operation to complete the self-healing process. Recovery involves issues such as integrating newly committed resources into ongoing processes, "warming up" resources by transferring system state into them, or taking action to bring the system to a clean known state before proceeding with operations. A component might be hot-swapped, require a warm system reboot, or require a cold system reboot to finish recovery.

Time constants: The time constants of a system, along with the fault distribution assumptions, play a large role in determining what types of self-healing are feasible. The time constant of faults with respect to the forward progress of computations determines things like the frequency at which checkpoints must be taken, or whether a system can reboot itself quickly enough to prevent an overall system outage. Additionally, if intermittent or transient faults are in the majority as is typical in many systems, the speed of the detection-response-recovery cycle might need to be faster than typical fault arrival periods to avoid system thrashing.

Assurance: Every domain has a specific set of system properties of importance. Every system requires assurances of some level of functional and extra-functional correctness for normal operation. Self-healing systems additionally require a way to assure that such properties are maintained during and after fault occurrences. Challenges in this area include assurance between the time a fault occurs and the time the fault is detected (keeping in mind that not all faults are detected); assurance during degraded mode operations; and assurance during recovery operations. This assurance might be provided at design time or might involve checks at run time. Finally, the assurance might be absolute or probabilistic, and might involve all functionality or partial assurance of only a few key system properties.

2.3. System completeness

Real systems are seldom complete in every sense. Self-healing approaches must be able to deal with the reality of limits to knowledge, incomplete specifications, and incomplete designs.

Architectural completeness: Few system architectures are completely elaborated when the first implementation is built. Architectures and implementations evolve over time. Many systems are "open" in that third-party components can be added during or after system deployment. And, many systems are designed using prebuilt components that have details and behavior so opaque to the overall system designer that the architecture might as well be considered incomplete. Finally, a system might be built upon discovery mechanisms which are intended to extend the architecture or implementation at run-time. A related issue is that implementations of components evolve, are patched, suffer configuration management problems, and so on.

Designer knowledge: Designers in the typical case do not have complete knowledge of the systems they design. Any system is designed using a set of abstractions about underlying components. But beyond that the designer must deal with missing knowledge about aspects of components, and in all likelihood incorrect knowledge about system components due to documentation and implementation defects. It is common for designers to have a thorough understanding of typical system behaviors, but to have little or no understanding of atypical system behaviors – especially system behaviors in the presence of faults. A vital aspect of designer knowledge is how well the fault model for the system is characterized and whether field information about faults is fed back to the system designer.

System self-knowledge: Systems must have some level of knowledge about themselves and their environment in

order to provide self-healing. This self-knowledge is limited by the aspects of knowledge built into a component (for example, a component might or might not be able to predict its execution time in advance), the accessibility of knowledge about one component to another component, and defects in representation of such knowledge either due to initial design defects or staleness caused by system evolution. The concept of reflection is often discussed in the context of system self-knowledge; however it also seems possible to build systems that have no awareness of their state but rather exhibit emergent correctness as a consequence of the interaction of their component behaviors.

System evolution: Self-healing systems must deal with the fact that they change over time. Sources of change include designed operating mode changes, accumulated component and resource faults, adaptations to external environments, component evolution, and changes in system usage. Making use of available information on system dynamics might help with self-healing, such as being able to count on a scheduled system outage (or self-schedule an outage) to perform healing.

2.4. Design context

There are several other factors that influence the scope of self-healing capabilities that could be considered to form the design context of the system.

Abstraction level: Systems can attempt to perform various forms of self-healing to application software, middleware mechanisms, operating systems, or hardware. Self-healing techniques can be applied to implementations (such as wrappers to deal with unhandled exceptions) or architectural components.

Component homogeneity: While some systems have completely homogenous components, it is common to have systems that are heterogeneous to some degree. Server farms often have different versions of processing hardware, and might well have different versions of operating systems or other software installed, especially when changes are applied incrementally across a fleet of components as a risk management technique. Homogeneity can consist of exact component duplicates, or components that are "plug-compatible" even though they have differing implementations. Some systems are inherently heterogenous, such as the computational components within embedded systems such as automobiles. The heterogeneity of a system tends to limit its ability to simply migrate computational tasks as a self-healing strategy and requires that self-healing approaches deal with the issue of configuration management of systems both before and after healing.

Behavioral predetermination: Most systems do not have perfectly predetermined and deterministic behavior, and some self-healing approaches must be able to accommodate this. Non-deterministic behavior abounds in hardware and in software infrastructure. But, beyond that, it is often impractical to quantify things such as absolute worst-case execution time. Even things that might seem determinable in theory such as enumeration of all possible exceptions that can be generated by a software component might be impractical due to obscure component interactions or defects. In the time dimension, system tasks might be event-based or periodic, necessitating differing assumptions and approaches by healing mechanisms.

Both the system and the self-healing mechanism can have differing levels of behavioral predeterminism. For example, a rule-based application or one that employs neural networks might not be readily analyzed for behavior. Similarly, a self-healing mechanism might employ nondeterministic or analytically complex approaches that make design-time analysis of behavior impractical.

User involvement in healing: While the goal of much thinking about self-healing systems is to achieve complete autonomy, this might be an over-ambitious goal. Most systems have a limit to healing ability, beyond which users must become involved in system repair. The opportunity for self-healing system collaborations with users are twofold: users can adapt their behavior to help systems function despite failures, and users can provide advice to systems to guide aspects of their self-healing behavior.

System linearity: Overall system linearity and component coupling can greatly affect the ability of a system to self-heal. If a system is completely linear (i.e., all aspects of the system are completely composable from component aspects) then self-healing of one component can be carried out without concern for its effect upon other components. While many well-architected systems have good linearity, component interaction is a typical situation that must be addressed by self-healing approaches.

System scope: How big is the system? A single-node computing system does not have all the self-healing possibilities available to a geographically distributed computing system. Similarly, portions of the system might be considered out-of-bounds when creating a self-healing mechanism, such as a requirement to use an off-the-shelf operating system or existing Internet communication protocols. The scope of system self-healing might therefore be a single component; a computer system; a computer system plus a person; an enterprise automation suite; or the computer in the context of society including regulatory agencies, maintenance groups, and insurance mechanisms.

3. Examples of use

Because the purpose of this paper is to propose a way of structuring a complex and still relatively unexplored research area, it is unlikely that the results are complete or in-

Table 2. Self-Healing Problem Spaces Addressed By Example Research Projects.

	Property	RoSES Graceful Degradation	Semantic Anomaly Detection	Amaranth QoS
Fault Model	Fault Duration	Permanent	Permanent+Intermittent	Permanent+Intermittent
	Fault Manifestation	Fail fast+silent components Potentially correlated	Unexpected data feed values; Recovery only if uncorrelated	Resource exhaustion Potentially correlated
	Fault Source	All non-malicious sources	Representable by templates; Non-malicious	Peak resource demand; Non-malicious
	Granularity	Component failure in distributed embedded system	Failure of Internet data feed	Depletion of memory, CPU, etc. in distributed system
	Fault Profile Expectations	Random; arbitrary; unforeseen	Anomalies compared to prior experience	Random; resource consumption only
System Response	Fault Detection	State variable staleness	Anomaly detection	Resource monitoring alarm
	Degradation	Fail-operational; Maximize system utility	Not addressed	Preserve predetermined baseline functions; eject nonessential tasks
	Fault Response	Reconfigure SW based on data and control flow graphs	Substitute redundant data feed	Admission control policy: Admit "baseline" tasks and reject some enhanced tasks
	Recovery	Reconfigure SW & reboot system	On-the-fly data feed switch	Terminate enhanced tasks as necessary
	Time constants	Long time between failures; Can handle multiple failures	Valid data samples occur much more often than anomalies	Can handle multiple failures; Tasks can be terminated instantly
	Assurance	Future work; reliability-driven	"Good enough" data quality	Static analysis of baseline load
System Completeness	Architecture Completeness	Closed, complete system; Graceful upgrade/downgrade; System must work in worst case	Dynamic Internet data feeds; Unknown gaps & defects; Common case handling complete	Closed, complete system; System must work in worst case
	Designer Knowledge	Assumed to be complete	Component specifications unknown -- must be inferred	Complete; workload information is statistical distribution
	System Self-Knowledge	System knows component presence & failure; data/control flow	History used as basis for anomaly detection	Available resources and approximate task resource consumption
	System Dynamicism	Upgrades & downgrades; System stable during mission	Data feeds come and go	Workload is stochastic
Design Context	Abstraction Level	HW & SW components within distributed system	Nodes on Internet	Tasks within distributed system
	Component Homogeneity	Heterogenous components and resources	Redundant or correlated data feeds	Homogenous resources; heterogeneous tasks
	Behavioral Predetermination	Components characterized; Functions must be composable	Predetermined data feed type; Behavior of data feed discovered	System design predetermined; Workload is stochastic
	User Involvement	Fully automatic	User accepts/rejects templates	Fully automatic
	System Linearity	Multiattribute utility theory; Scalability assumes linearity; Bin-packing task approach	Not applicable	Tasks have discrete operating points; Bin-packing approach
	System Scope	Multiple computers in embedded control system	Multiple computers + user on Internet-based system	Multiple computers on Internet or closed network system

deed even apply to all research projects. Additionally, the type of information required to describe many projects is not fully available from published sources. In the interest of providing concrete examples, three of our own research projects are briefly described in terms of the proposed categories in Table 2.

RoSES (Robust Self-configuring Embedded Systems) [Shelton03] is a project that is exploring graceful degradation as a means to achieve dependable systems. It concentrates on allocating software components to a distributed embedded control hardware infrastructure, and is concerned with systems that are entirely within the designer's control.

The semantic anomaly detection research project [Raz02] seeks to use on-line techniques to infer specifications from underspecified components (e.g., Internet data feeds) and trigger an alarm when anomalous behavior is observed. An emphasis of the research is using a template-based approach to make it feasible for ordinary users to provide human guidance to the automated system to improve effectiveness.

The Amaranth project [Hoover01] is a Quality of Service project that emphasizes admission policies. A key idea is to have tasks with at least two levels of service: baseline and optimized. A system could thus be operated to guarantee critical baseline functionality via static system sizing, with idle resources employed to provide optimized performance on an opportunistic per-task basis.

All three projects are, in our opinion, "self-healing software system" research projects. But as shown by Table 2 they have widely varying areas of exploration, assumptions, and areas that are unaddressed. The area in which all three projects are substantially similar is the last attribute, in which all three systems assume a distributed computing environment. It is worth noting that the categories were created before Table 2 was constructed, so this provides initial evidence that the categories capture differences among general projects rather than being specific to just these projects. But of course since the people involved in the three projects discussed overlap, this does not prove generality and certainly does not demonstrate completeness.

4. Conclusions

It is too soon to tell whether "self-healing" system approaches are just a different perspective on the area of fault tolerant computing, and whether that perspective brings significant benefits. Resolving this issue requires better understanding of what is meant by the term "self-healing" in the first place. To that end, this paper proposes a taxonomy for describing the problem space for self-healing systems.

Relevant aspects of self-healing system approaches include fault models, system responses, system completeness, and design context. It is of course unreasonable to expect every research paper on self-healing systems to address every possible aspect discussed, and no doubt some important aspects are yet to be discovered. It remains to be seen how different aspects interact in various domains, and which aspects matter the most in practice. However, it is hoped that this taxonomy will provide a checklist for researchers to use in explaining the part of the problem space they are addressing, and perhaps to help avoid inadvertent holes in self-healing system approaches.

5. Acknowledgments

I would like to thank the participants of WOSS02 for stimulating discussions that made this paper possible, the anonymous reviewers for their comments, and the members of my research group for their additional suggestions. The techniques in this paper come from the experience of others and decades of research, especially in the fault tolerant/dependable computing community. Interested readers should consult the proceedings of the Fault Tolerant Computing Symposium (FTCS) and Dependable Systems & Networks (DSN) conference for further information.

This work was supported in part by the High Dependability Computing Program via NASA Ames cooperative agreement NCC-2-1298, and in part by the General Motors Collaborative Research Laboratory at Carnegie Mellon University.

6. References

[Avizienis01] A. Avizienis, J.-C. Laprie and B. Randell, *Fundamental Concepts of Dependability*, Research Report N01145, LAAS-CNRS, April 2001.

[Bouricius69] Bouricius, W.G., Carter, W.C. & Schneider, P.R, "Reliability modeling techniques for self-repairing computer systems," *Proceedings of 24th National Conference*, ACM, 1969, pp. 395-309.

[Hoover01] Hoover, C., Hansen, J., Koopman, P. & Tamboli, S., "The Amaranth Framework: policy-based quality of service management for high-assurance computing," *International Journal of Reliability, Quality, and Safety Engineering*, Vol. 8, No. 4, 2001, pp. 1-28.

[Raz02] Raz, O., Koopman, P., & Shaw, M., "Enabling Automatic Adaptation in Systems with Under-Specified Elements," *1st Workshop on Self-Healing Systems (WOSS'02)*, Charleston, South Carolina, November 2002.

[Shelton03] Shelton, C., Koopman, P. & Nace, W., "A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems," *WORDS03*, January 2003.

[vonNeuman56] von Neumann, J., "Probabilistic logics and the synthesis of reliable organisms from unreliable components," 1956, in Taub, A. H., (ed.), *John von Neumann: collected works*, Volume V, pp. 329-378, New York: Pergamon Press, 1963.

[WOSS02] *1st Workshop on Self-Healing Systems (WOSS'02)*, Charleston, South Carolina, November 2002, ACM Press.

Dependability Analysis Using SAM

Tianjun Shi, Xudong He
School of Computer Science
Florida International University, Miami, FL 33199
{tshi01, hex}@cs.fiu.edu

Abstract

Software architecture has been of increasing importance for designers to analyze and verify system properties. Non-functional properties reflect the quality of a software system and are essential for a successful software system, but analysis of non-functional properties is less well studied compared to functional verification in the filed of software architecture. Performance and dependability are most concerned non-functional properties in safety and mission critical systems. SAM is a framework for specifying and analyzing software architecture and has been used to verify a variety of functional properties. In this paper, we extend SAM with stochastic constructs so that it can be used to model and analyze performance and dependability.

1. Introduction

As software systems keep growing in size and complexity, software architecture as a high-level abstraction is of increasing importance for designers to reason about the software systems and make early design decisions. As a result, many architecture description languages (ADLs) have been proposed [10]. Most ADLs, however, focus mainly on the formal notation, and many of them do not offer analytical techniques to verify the properties of their designs. Some ADLs do provide analytical techniques, but tend to focus on a particular property and leave other properties unexplored. For example, Wright [2] allows deadlock detecting; MetaH [18] and UniCorn [13] support schedulability analysis [10]. To contribute to the analysis of software architectures, we extend *Software Architecture Model* (SAM) with stochastic constructs for non-functional property evaluation in addition to its existing capability of functional property verification.

SAM is a formal framework for specifying and analyzing software architectures [19]. It supports hierarchical decomposition and automatic analysis of software architectures. Different techniques and tools have been used to analyze a SAM model for the evaluation of various properties, and the SAM framework has proved to be useful and desirable in specifying and verifying software architectures. For example, in [19], reachability analysis technique was used to analyze the timeliness of a real time system. The symbolic model checker SMV was used in [14] to verify the functional correctness of a communication protocol, and the theorem prover STeP was used to reason about the correctness of an electronic commercial system [20].

To date, the SAM framework mainly focuses on functional property analysis. However, non-functional properties, such as performance and dependability, are of the same importance to the quality of a software system. Dependability, known as the collection of reliability, availability, safety and related measures, is especially concerned in safety and mission critical system. Different modeling and analyzing techniques have been exploited to evaluate various non-functional properties. For example, fault-trees were used for system reliability modeling [11]; queuing networks have been used for performance analysis of computer and communication systems [1, 12]; a variety of *Stochastic Petri Nets* (SPNs), as a convenient high level formalism of Markov chains, have become popular for the analysis of performance, dependability and performability [5, 9]. In most research work, functional properties and non-functional properties are separately modeled and evaluated because of the complexity of integration. The separation eases the work of modeling and analysis, but is prone to inconsistency of models and imprecision of evaluations. With incorporated stochastic constructs, the SAM model can be transformed to a Stochastic Reward Net [5] (SRN) model. While functional property verification is conducted using the techniques mentioned above, performance and dependability can be evaluation by solving the derived SRN model.

The rest of the paper is organized as follows. Sec-

tion 2 introduces SAM and SRNs. In section 3, a multiprocessor system is given as an running example throughout the paper. Section 4 depicts the stochastic extension on SAM, and section 5 explains how to transform the SAM model into an SRN. The results of reliability analysis for the example system are described in section 6, and conclusions are drawn in section 7.

2. Overview of Formalisms

To give readers a rough idea about the formalisms used in this paper, we briefly introduce SAM and SRN.

2.1. Software Architecture Model

SAM is a formal framework for specifying and analyzing software architectures. A SAM model consists of a set of *compositions* and a *hierarchical mapping* relating the compositions. A composition in turn consists of multiple *components* and *connectors*, and a set of composition *constraints*. Each component (or connector) consists of a behavior model and a set of component (or connector) constraints. A SAM model is said to be correct if the behavior models satisfy each constraint. For a formal definition and description of SAM, refer to [8].

There are two complement formal methods underlying a SAM model. Petri nets are used to define the behavior models of components and connector, while temporal logic is used to specify the constraints. To be flexible, the underlying formal foundation of SAM is not limited to a fixed pair of Petri net and temporal logic. The selection of a particular Petri net and temporal logic is based on the application under consideration. For example, real-time Petri nets and real-time computational tree logic were used to study software architectures of real-time systems [19], *Predicate Transition nets* (PrT nets) and first order linear time temporal logic were used to specify and verify a communication protocol [14].

In this paper, we use PrT nets as the behavior modeling formalism of the SAM framework. PrT nets are a class of high level Petri nets. Using the conventions and definitions in [8], a PrT net is defined as a tuple $(N, Spec, ins)$, where $N = (P, T, F)$ is the net structure; *Spec* defines the used sorts, operations and relations; and $ins = (\varphi, L, R, M_0)$ defines the mapping of places to sorts, the labels, the constraint of each transition, and the initial marking respectively. For a complete definition of a PrT net, refer to [8].

2.2. Stochastic Reward Net

SRN [5] is an extension to SPN. An SRN can be mapped into a Markov Reward model [5]. By definition, an SRN is an 11-tuple consisting of:

1. A finite set of places.
2. A finite set of transitions.
3. A finite set of input arcs from place to transition.
4. A finite set of output arcs from transition to place.
5. A finite set of inhibitor arcs from place to transition.
6. Enabling function for each transition. A transition is disabled if the enabling function is evaluated false.
7. Priorities for each transitions. A transition is disabled if there exists an enabled transition with a higher priority.
8. The initial marking.
9. A positive exponential distributed rate for each timed transition, and an immediate transition fires at no time.
10. A weight for each transition.
11. Reward measures.

SPNP [15] and SMART [4] are two tools for SRNs. By solving the underlying Markov chains of an SRN, performance and dependability of the model can be evaluated.

3. A Multiprocessor Example

We use a multiprocessor system depicted in [3] as a running example to demonstrate the use of extended SAM. As illustrated in Figure 1, a multiprocessor system consists of n subsystems $S_1, \ldots, S_n$. Each subsystem S_i is composed of one processor P_i, one local memory M_i, and m replicated mirrored disk units D_{ij}. A bus B connects the n subsystems and a shared memory M_g, which is shared by all subsystems. The complete system fails when the bus B fails or when k $(1 \leq k \leq n)$ subsystems fail. A subsystem fails when its processor fails or all its disks fail or both the local memory and the shared memory fail.

The reliability (i.e. probability of system down) is the main issue of the multiprocessor system. Figure 2 depicts the behavior model in SAM for the failures of the multiprocessor system. Each net element in the behavior model is described in Table 1. Initially, every element of system is working, The net inscription of the behavior model is as follows. Symbol $\wp$ denotes power set; n is the number of subsystems; m is the number of disks in each subsystem;

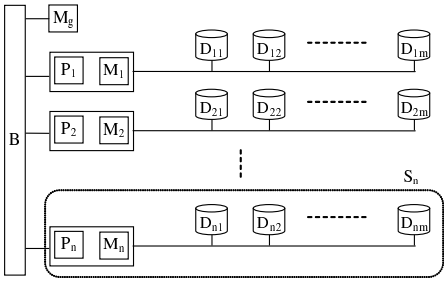

Figure 1. The multiprocessor system.

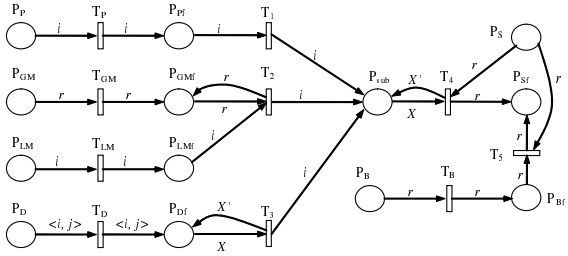

Figure 2. The SAM model for the example.

Table 1. Description of net elements.

Elements	Description
P_P/P_{Pf}	Processor P_i is working/not working.
P_D/P_{Df}	Disk D_{ij} is working/not working.
P_{GM}/P_{GMf}	memory M_g is working/not working.
P_{LM}/P_{LMf}	memory M_i is working/not working.
P_B/P_{Bf}	Bus B is working/not working.
P_S/P_{Sf}	The system is working/not working.
P_{sub}	Subsystem S_i is not working.
T_P	Processor P_i fails.
T_D	Disk D_{ij} fails.
T_{GM}	Shared memory M_g fails.
T_{LM}	Local memory M_i fails.
T_B	Bus B fails.
T_1	S_i fails due to processor failure.
T_2	S_i fails due to memory failures.
T_3	S_i fails due to disk failures.
T_4	system fails due to subsys. failures.
T_5	system fails due to bus failure.

and k is the minimum number of subsystems whose failures will result in the failure of the complete system.

$$\varphi(P_P) = \varphi(P_{Pf}) = \wp(\{i | 1 \le i \le n\})$$
$$\varphi(P_{LM}) = \varphi(P_{LMf}) = \varphi(P_{sub}) = \varphi(P_P)$$
$$\varphi(P_B) = \varphi(P_{Bf}) = \wp(\{\text{“}notempty\text{”}\})$$
$$\varphi(P_{GM}) = \varphi(P_S) = \varphi(P_{GMf}) = \varphi(P_{Sf}) = \varphi(P_B)$$
$$\varphi(P_D) = \varphi(P_{Df}) = \wp(\{\langle i,j\rangle | 1 \le i \le n, 1 \le j \le m\})$$
$$R(T_P) = R(T_{LM}) = R(T_B) = R(T_{GM}) = true$$
$$R(T_D) = R(T_1) = R(T_2) = R(T_5) = true$$
$$R(T_3) = (\exists Y \subseteq X.\ Y = \{\langle i,j\rangle | 1 \le j \le M\} \wedge X' = X - Y)$$
$$R(T_4) = (\exists Y \subseteq X.\ |Y| = K \wedge X' = X - Y \wedge r = \text{“}notempty\text{”})$$
$$M(P_P) = M(P_{LM}) = \{i \mid 1 \le i \le n\}$$
$$M(P_D) = \{\langle i,j\rangle |\ 1 \le i \le n\ \wedge\ 1 \le j \le m\}$$
$$M(P_B) = M(P_{GM}) = M(P_S) = \{\text{“}notempty\text{”}\}$$
$$M(P_{Pf}) = M(P_{Df}) = M(P_{Bf}) = M(P_Sf) = \{\}$$
$$M(P_{LMf}) = M(P_{GMf}) = M(P_{sub}) = \{\}$$

In this behavior model, the failure rate of each element of the multiprocessor system is not addressed yet. Therefore, the reliability of the system cannot be analyzed based on this model. The next section shows how SAM is extended to express stochastic information.

4. Stochastic Extension on SAM

In SAM, each behavior model is represented by a PrT net, and each requirement constraint is specified in temporal logic. The SAM framework is extended in two ways for non-functional property analysis. First, a stochastic construct is add to the behavior model (PrT net model) to express stochastic information. Second, a formalism is introduced to specify the non-functional requirements.

A special variable $RATE$ is added into the constraint of a transition (i.e. the mapping R in the net inscription) in a PrT net to denote the firing rate of the transition. A predicate $RATE = \lambda$ in the constraint of transition T denotes that T fires at rate λ, and $RATE = 0$ denotes that T is an immediate transition and takes zero time to fire. For example, if the bus B in the multiprocessor system fails at rate λ_B, the constraint of transition T_{LM} in Figure 2 can be expressed as $R(T_B) = (RATE = \lambda_B)$. The firing rate of a transition is not necessary constant, it can be expressed as marking dependent by making more than one occurrence of variable $RATE$. For example, if local memory M_1 in subsystem S_1 is more dependable than all other local memories, and M_1 fails at rate λ_1 while others fail at rate λ_2 ($\lambda_1 < \lambda_2$), then the constraint of T_{LM} in Figure 2 can be specified as $R(T_{LM}) = ((RATE = \lambda_1 \wedge i = 1) \vee (RATE = \lambda_2 \wedge 2 \le i \le n))$. The extra stochastic construct in the constraint of a transition does not affect the enabling and firing rules of this transition. That is, the PrT net model with the special variable $RATE$ has the same semantics with a original PrT net regarding the enabling and firing of a transition.

After extending the behavior model in SAM with the

stochastic construct, a formalism is desirable to precisely specify non-functional requirements. Although some non-functional properties (e.g. maintainability) cannot be quantified and can only be specified informally, performance, dependability and performability can usually be precisely specified and verified at runtime. Temporal logic is sufficient to specify functional properties. However, it is incapable of specifying dependability and performability since those properties usually involve probability and precise time. To specify dependability and performability in SAM, we introduce a logic called *Probabilistic real time Computation Tree Logic* (PCTL), proposed by Hansson and Jonsson [7]. PCTL extends *Computation Tree Logic*, a branching-time temporal logic, and is capable of expressing time and probability in systems. For example, the property "after a request for service there is at least a 98% probability that the service will be successfully finished within 5 seconds" can be expressed in PCTL as $AG[(p \rightarrow F^{\leq 5}_{\geq 0.98} q)]$, where proposition p represents that a request is issued and proposition q represents that the service is successfully finished. Since performance, dependability and performability can usually be expressed in terms of time and probability, it is sufficient to express most indices of dependability and performability to be verified using PCTL. The advantage of specifying properties formally in PCTL is that it allows flexible analytical techniques. For example, we can also apply model checking over the underlying Markov chain of an SRN to verify properties specified in PCTL [7]. However, there do exist some indices of performance and dependability that cannot be expressed in PCTL. For example, PCTL cannot express the average time it takes to finish a service. For those properties that cannot be expressed by PCTL, we specify them informally in natural language.

By assuming that each element of the same type in the multiprocessor system fails at the same rate, the constraint mapping of the behavior model in Figure 2 is modified as follow to reflect the stochastic information.

$$
\begin{aligned}
&R(T_P) = (RATE = \lambda_P)\\
&R(T_{LM}) = (RATE = \lambda_{LM})\\
&R(T_{GM}) = (RATE = \lambda_{GM})\\
&R(T_D) = (RATE = \lambda_D)\\
&R(T_B) = (RATE = \lambda_B)\\
&R(T_1) = R(T_2) = R(T_5) = (RATE = 0)\\
&R(T_3) = (RATE = 0 \wedge (\exists Y \subseteq X.\\
&\qquad Y = \{\langle i,j\rangle | 1 \leq j \leq M\} \wedge X' = X - Y))\\
&R(T_4) = (RATE = 0 \wedge (\exists Y \subseteq X.\\
&\qquad |Y| = K \wedge X' = X - Y \wedge\ r = \text{“}notempty\text{”}))
\end{aligned}
$$

For the reliability requirement of the multiprocessor system, we assume that there is at least 95% probability that the system is working continuously for 4000 hours. This reliability requirement can be specified in PCTL as $AG[F^{\leq 4000}_{\geq 0.95} M(P_{Sf}) = \phi]$, where $M(P_{Sf}) = \phi$ denotes that place P_{Sf} in Figure 2 holds no token, which means the system is working.

5. Transformation from SAM to SRN

In order to analyze non-functional properties, the behavior model in SAM is transformed to an SRN. The procedure of the transformation is as follows.

1. The PrT net model in SAM is first unfolded to a low level Petri net. A PrT net can be unfolded using the method proposed in [6]. First, each transition is unfolded into a set of transitions according to the set of constant substitutions that satisfy constraint of the transition. Second, places are connected to the transitions according to the substitutions. Finally, remove the dead transitions and combine equivalent elements if any. It is not necessary to unfold the whole PrT net if we are only interested in part of the PrT net model.
2. The SRN model is derived from the unfolded Petri net based on the stochastic information. After the PrT net model is unfolded, each transition is designated either as an immediate transition or as a timed transition with a proper firing rate based on the stochastic information. By making use of the features of SRNs such as enabling function, marking dependent arc cardinality and marking dependent firing rate, the SRN model can be more concise and readable.
3. After the SRN model is derived, existing solution techniques and tools can be applied to obtain the non-functional property measures, and then the numerical results are checked against the property specifications in PCTL to see whether the model satisfies the specifications. The SRN model may need to be revised for the analysis of some non-functional properties. For example, to analyze the mean time to failure (MTTF) of a system, all the failure states should be made absorbing (outgoing arcs from those states are removed) [17].

Following the procedure above, the behavior model in Figure 2 is transformed to the SRN model in Figure 3, by assuming that $n = 3$, $m = 2$, and $k = 2$. That is we assume that there are three subsystems in the multiprocessor system, each subsystem has two disks and the whole system fails if at least two subsystems fail. In the SRN model, a bar represents an immediate transition and a box represents a timed transition. Figure 3(a) is the SRN model for a subsystem S_i, and

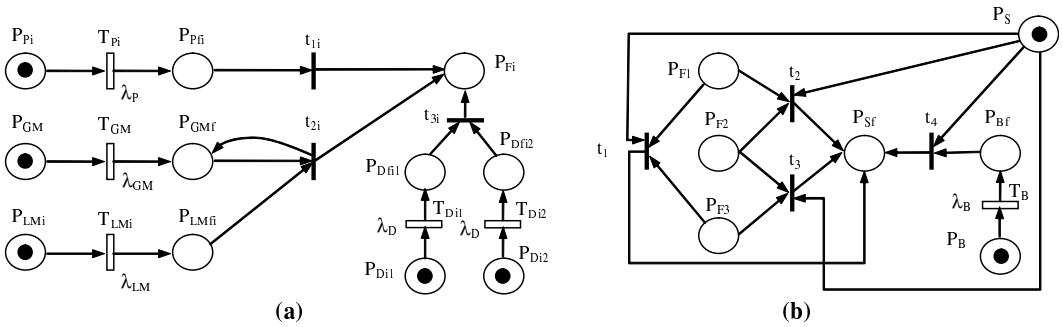

Figure 3. (a) The SRN model for subsystem P_i. (b) The SRN model for the example system.

Figure 3(b) illustrates the SRN model for the multiprocessor system with the detailed structure of each subsystem being hidden. By calculating the number of expected tokens in place P_{Sf} at time t, the probability that the multiprocess system fails at time t is obtained.

In recent research work, SRNs have been applied to evaluate a variety of non-functional properties like performance, dependability and performability. After extending SAM as mentioned above, the SAM framework is capable of modeling and analyzing all the non-functional properties that can be evaluated in SRNs. However, this methodology also inherits the limitations of SRN models. There are three main difficulties in SRN model—*largeness*, *stiffness*, and the assumption of exponentially distributed firing rates of timed transitions [16]. Largeness means the large size of the underlying Markov model of an SRN and stiffness represents the large disparity between the firing rates of an SRN. Largeness and stiffness cause difficulties in solving the underlying Markov model of an SRN. Fortunately, a number of approaches have been proposed to avoid and/or tolerate largeness and stiffness [16]. The "exponential assumption" limits the modeling power and precision of SRNs. While exponentially distributed rates are reasonable in some situations such as failure rates of a system, they are invalid in other situations like deadlines of a system. This problem, however, can be reduced either by phase approximations [16] (approximating a non-exponential distribution by a set of states and transitions such that the holding time in each state is exponentially distributed) or by allowing non-exponential distributions in SRNs.

6. Analysis Results

For the reliability analysis of the multiprocessor system, we assign the failure rates of each elements as listed in Table 2[1]. Based on the SRN model in Figure 3, we evaluate the probability of the system failure at a function of the time. The probability of system failure is obtained by using the tool SMART [4] to calculate the probability that place P_{Sf} in Figure 3 is not empty at time t. Figure 4 shows the distribution of the probability of system down. The numerical results show that there is 96.9% probability that the system keeps working at the $4000th$ hour, which means that the reliability requirement specified in section 5 is satisfied in the modeled system.

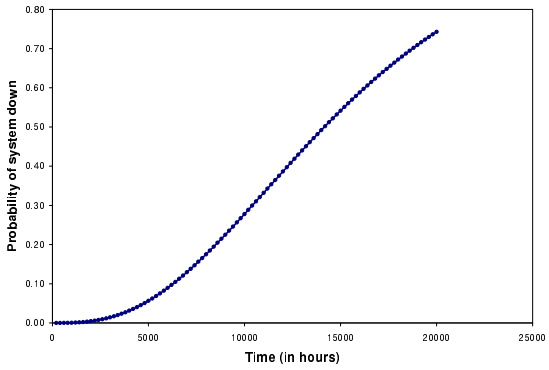

Figure 4. Reliability of the example system.

7. Conclusions

A method to extend SAM for performance and dependability analysis has been presented. After incorporating stochastic information into a SAM model, an SRN model is derived. By solving the SRN model using existing tools, a variety of non-functional properties such as performance, dependability and performability

[1] The failure rates are adopted from [3]

Table 2. Failure rates for the example system.

Elements	Failure rates (fault/hour)
processor	$\lambda_P = 5 \times 10^{-7}$
disk	$\lambda_D = 8 \times 10^{-5}$
shared memory	$\lambda_{GM} = 3 \times 10^{-8}$
local memory	$\lambda_{LM} = 3 \times 10^{-8}$
bus	$\lambda_B = 2 \times 10^{-9}$

can be evaluated. After the extension with stochastic construct, the SAM framework is capable of the analysis of both functional properties and the common non-functional properties. To illustrate the method, the multiprocessor system was used as a running example, and its reliability was evaluated. The presentation of the example and its numerical results proved the viability of the proposed method, and allowed significant insight to the system qualities to be gained.

The analysis of performance and dependability using SRNs is not new. Our contribution lies in incorporating stochastic techniques into the SAM framework so that both functional properties and non-functional properties like dependability can be analyzed under one unified framework. Currently, the transformation from SAM to SRN model is done by hand, we are interested in developing tools to automate the transformation. We also plan to do the tradeoff analysis of certain conflicting non-functional properties.

References

[1] I. F. Akyildiz. On the exact and approximate throughput analysis of closed queuing networks with blocking. *IEEE Transactions on Software Engineering.*, 14(1):62–70, 1988.

[2] R. Allen and D. Garlan. Formalizing architectural connection. In *Proc. the Sixteenth International Conference on Software Engineering*, pages 71–80, 1994.

[3] A. Bobbio, G. Franceschinis, R. Gaeta, and L. Portinale. Parametric fault tree for the dependability analysis of redundant systems and its high-level petri net semantics. *IEEE Transactions on Software Engineering*, 29(3):270–287, 2003.

[4] G. Ciardo, R. L. Jones, R. M. Marmorstein, A. S. Miner, and R. Siminiceanu. SMART: Stochastic model-checking analyzer for reliability and timing. In *Proc. International Conference on Dependable Systems and Networks (DSN'02)*, Washington, D.C., June 1992.

[5] G. Ciardo, J. Muppala, and K. S. Trivedi. Analyzing concurrent and fault-tolerant software using stochastic reward nets. *Journal of Parallel and Distributed Computing*, 15(3):255–269, 1992.

[6] H. J. Genrich. Predicate/transition nets. *Lecture Notes in Computer Science*, 254:255–269, 1987.

[7] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(4):512–535, 1994.

[8] X. He and Y. Deng. A framework for developing and analyzing software architecture specifications in SAM. *The Computer Journal*, 45(1):111–128, 2002.

[9] A. R. McSpadden and N. Lopez-Benitez. Stochastic petri nets applied to the performance evaluation of static task allocations in heterogeneous computing environments. In *Proc. 6th Heterogeneous Computing Workshop (HPC'97)*, pages 185–194, 1997.

[10] N. Medvidovic and R. Taylor. A framework for classifying and comparing architecture description languages. In *Proc. the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 60–76, 1997.

[11] Y. Ren and J. B. Dugan. Design of reliable systems using static and dynamic fault trees. *IEEE Transactions on Reliability*, 47(3):234–244, May 1998.

[12] T. G. Robertazzi. *Computer Networks and Systems: Queuing Theory and Performance Evaluation.* Springer-Verlag, second edition, 1994.

[13] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–355, 1995.

[14] T. Shi and X. He. Modeling and analyzing the software architecture of a communication protocol using SAM. In *Proc. IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture (WICSA'02)*, pages 63–77, 2002.

[15] K. S. Trivedi. *SPNP User's Manuel, version 6.0.* Department of Electronic and Computer Engineering, Duke University, 1999.

[16] K. S. Trivedi, G. Ciardo, M. Malhotra, and R. A. Sahner. Dependability and performability analysis. *Performance Evaluation of Computer and Communication Systems, Lecture Notes in Computer Science*, 729:587–612, 1993.

[17] K. S. Trivedi, M. Malhotra, and R. M. Fricks. Markov reward approach to performability and reliability analysis. In *Proc. the Second International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems (MASCOTS'94)*, pages 7–11, 1994.

[18] S. Vestal. MetaH programmer's manual, version 1.09. Technical report, Honeywell Technology Center, 1996.

[19] J. Wang, X. He, and Y. Deng. Introducing software architecture specification and analysis in SAM through an example. *Information and Software Technology*, 41(7):451–467, 1999.

[20] H. Yu, X. He, Y. Deng, and L. Mo. A formal method for analyzing software architecture models in SAM. In *Proc. the 26th Annual International Computer Software and Applications Conference (COMPSAC'02)*, pages 645–652, 2002.

Formalizing Dependability Mechanisms in B: From Specification to Development Support

F. Tartanoglu, V. Issarny
INRIA Rocquencourt
France
Galip-Ferda.Tartanoglu@inria.fr
Valerie.Issarny@inria.fr

N. Levy
Laboratoire PRISM
Université de Versailles
Saint-Quentin, France
Nicole.Levy@prism.uvsq.fr

A. Romanovsky
School of Computing Science
University of Newcastle upon Tyne
UK
Alexander.Romanovsky@newcastle.ac.uk

Abstract

The CA action concept has been proven successful for building dependable distributed systems due to its support for error recovery for both competitive and cooperative concurrent actions. This paper introduces the formal specification of dependability mechanisms offered by CA actions using the B formal method, from which an XML-based language is derived. The resulting language then allows developing dependable systems, where the B formal specification is refined to obtain an implementation of the associated runtime support.

1. Introduction

Dependability of systems is defined by the reliance that can be put on the service they deliver. Developing distributed systems that are dependable is recognized as a complex task, requiring adequate mechanisms for dealing with the occurrence of failures. Coordinated Atomic Actions (CA actions) [8] provide a general structuring mechanism for developing dependable systems through the exploitation of atomic actions and transactions. The composition of CA actions [6] further extends the base CA action model for developing open distributed systems.

Several applications have proven that CA actions are effective for building dependable concurrent systems [9, 2]. Formalization of applications based on CA actions, using Petri nets and temporal logic, further enables to prove the applications' dependability properties through model-checking [7, 5]. However, dependability properties are proved with respect to a specific application.

Our approach aims at providing a language for developing distributed systems using dependability mechanisms that are formally specified and implemented. Towards that goal, we provide a formal specification of the dependability mechanisms associated with the CA action concept using the B formal method, from which we derive an XML-based language to be used to develop dependable applications.

The paper is structured as follows. Section 2 briefly presents CA actions and their composition. Sections 3 and 4 then introduce the B formal specification of CA actions, discussing in particular the specification of dependability mechanisms offered by CA actions. Definition of the resulting XML-based development support follows in Section 5. Finally, Section 6 concludes, summarizing our contribution and discussing areas for future work.

2. Architecting Dependable Systems with Coordinated Atomic Actions

CA Actions

The CA actions [8] are a structuring mechanism for developing dependable concurrent systems through the generalization of the concepts of atomic actions [3] and transactions [4]. Atomic actions are used for controlling cooperative concurrency among a set of participating processes and for realizing coordinated forward error recovery using exception handling; transactions are used for maintaining the coherency of shared external resources that are competitively accessed by concurrent actions. Each CA action is designed as a multi-entry unit with roles activated by action participants, which cooperate within the action. A transaction is started upon each first access to a given external object by a CA action participant and it terminates at the end of the CA action. A CA action terminates with a normal outcome if no exception has been raised or if an exception has been raised and handled successfully; all transactions on external objects are then committed. If a participant raises an exception inside an action and if the exception cannot be handled locally by the participant, the exception is propa-

gated to all the other participants of the CA action for *coordinated error recovery*[1]. If coordinated recovery fails, the CA action terminates with an exceptional outcome. An exception is then signalled by the CA action and transactions on external objects are aborted.

CA actions can be designed in a recursive way using action nesting. Several participants of a CA action can enter into a *nested CA action*, which defines an atomic operation inside the embedding CA action. Accesses to external objects within a nested action are performed as nested transactions so that if the embedding CA action terminates exceptionally, all sub-transactions that were committed by nested actions are aborted as well. A CA action participant can only enter one nested action at a time. Furthermore, a CA action terminates only when all its nested actions have completed. Note that if the nested action terminates exceptionally, an exception is signalled to the containing CA action.

As an illustration, Figure 1 depicts a CA action *A1* that is entered by participants *P1-P3* and that comprises two nested CA actions, *A11* and *A12*; transaction are further executed on external objects. An exception raised by participant *P2* causes the CA action to enter an exceptional state, as showned by the greyed box, where the participants cooperate for handling the exception.

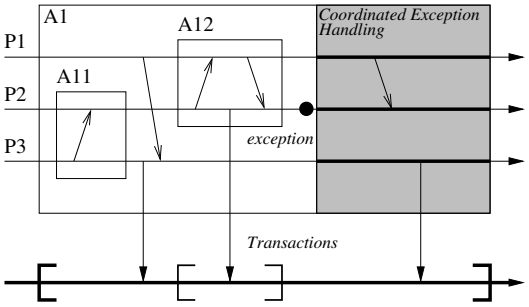

Figure 1. Coordinated atomic actions

CA actions mainly focus on structuring concurrent systems and on providing their fault tolerance by exception handling. One of the main intentions behind CA actions is to employ them as the mechanism for structuring complex distributed applications: they promote recursive view on system execution with abstracting away both normal and abnormal behaviour of the low level software.

CA Actions Composition

Composing CA actions allows the design of open distributed systems built out of several CA actions. Unlike classical action nesting where a subset of action participants enters into a nested action, *composed CA actions* are autonomous entities with their own participants and external objects. In this model, a participant of a CA action can dynamically initiate the creation of a composed CA action (or dynamically nested action).

The internal structure of a composed CA action (i.e., set of participants, accessed external objects and participants' behavior) is hidden from the calling CA action, which only has an access to the composed CA action's interface. A participant that calls a composed CA action enters a waiting state in a way similar to a synchronous RPC. The participant then resumes its execution according to the outcome of the composed CA action. If the composed CA action terminates exceptionally, its calling participant raises an internal exception which is possibly locally handled. If local handling is not possible, the exception is propagated to all the peer participants for coordinated error recovery. Note that unlike static nesting, when a composed CA action has terminated with a normal outcome, an abort operation of the containing CA action does not automatically compensate effects of the composed one; specific handling must be performed at a higher level, e.g., a composed action can be initiated to abort/compensate actions on external objects if needed.

Figure 2 illustrates the use of nested and composed CA actions, considering a travel agency system. The top-level CA action comprises the *User* and the *Travel* participants; the former interacts with the user while the latter achieves joint booking according to the user's request. The CA action has further access to the *Banking System*. In a first step, the *User* participant requests the *Travel* participant to search for a trip. This leads the participants to enter the nested action *SearchTrip* in which the *Travel* participant invokes a composed action comprising the *Hotel* and the *Flight* participants. The external objects accessed by those participants are the hotel and flight booking system. The *SearchTrip* action, if successful, returns a list of possible trips. Then, according to the *User*'s selection, the *BookTrip* nested action is executed, where another composed CA action is initiated to book the given trip. If an exception is raised within the composed CA action (e.g., *no_flight_available* for a given destination) and if it cannot be handled internally, the composed action terminates exceptionally by aborting all transactions on external objects and signals a failure exception to the higher level.

3 Specifying CA Actions in B

The B method

B is a complete formal method [1] that supports a large part of the development life cycle, from abstract specification to implementation. The B formal method is a model-based method, which is based on set theory and predicate

[1] If several exceptions have been raised concurrently they are resolved using a resolution tree imposing a partial order on all action exceptions, and the participants handle the resolved exception [3].

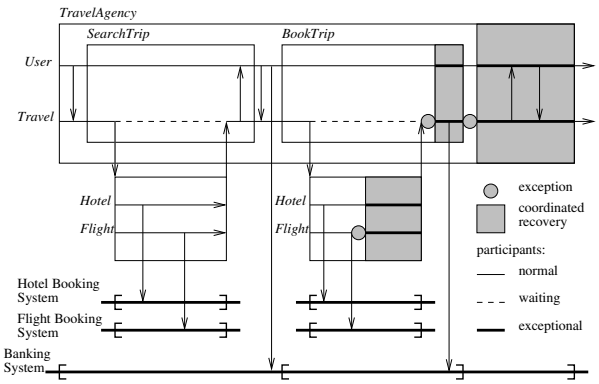

Figure 2. CA actions composition

logic and extended by generalized substitutions. B specifications are represented by abstract machines encapsulating operations and states. Generally speaking, the B method allows us to define *abstract machines* and *refinements* over them. During the refinement nondeterminism is reduced and preconditions are relaxed, but the interfaces of the operations remain the same. At the end of the refinement process, an *implementation* can be written, which corresponds to an executable code.

Proofs are an essential part of the model: it should be proven that all operations preserve the invariants of the machine, and that the implementations and refinements preserve the invariants and the behaviour of the initial abstract machine. There are various tools that help writing and proving B specifications. The main of them are B-Tool[2] and Atelier B[3]. Both tools include a type checker, an animator, a proof obligation generator, theorem provers, code translators and documentation facilities. Atelier B has been used in our investigation, however the notation we used is compatible with B-Tool.

Modelling CA Actions

Our goal in providing the B specification of CA actions is to offer a general framework that can be instantiated to describe the implementation of a specific system that is developed using CA actions. The framework thus defines the dependability properties associated with CA actions, which will be enforced for any system based on them.

The B formal specification of CA actions is given by the *CAACTIONS* abstract machine. The machine extends machines *OBJECTS* and *PARTICIPANTS*, which respectively describe external objects that can be accessed or modified by a CA action, and the participants of a CA action (see Figure 3). The *CONST* machine further contains global declarations and is seen by all the other machines. In the remainder, we introduce the main elements of the B specification, focusing on dependability properties associated with CA actions; the interested reader may find the overall specification at `http://www-rocq.inria.fr/~tartanog/dsos/`.

[2]http://www.b-core.com/btool.html

[3]http://www.atelierb.societe.com

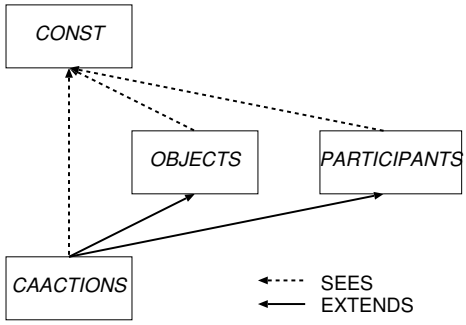

Figure 3. Structure of the B specification

The state of the *PARTICIPANTS* abstract machine characterizes the participants of a CA action as follows:

- The *PARTICIPANT* set is declared in the *CONST* machine and represents all possible participants that can be involved in a CA action.
- A participant that is activated by entering in a CA action is included in the subset *participant* of *PARTICIPANT* and removed at the end of the action:

  $participant \subseteq PARTICIPANT$

- Each participant enters in a sequence of modes (referred to as *state*), which can be normal, exceptional when an exception has been raised, or waiting if the participant invokes a composed CA action and is blocked until the action's termination:

  $participant\_state \in PARTICIPANT \rightarrow \mathbf{seq}(PARTICIPANT\_STATE)$

- Each participant has a value (local variables) that is logged for later use in case of backward recovery:

  $participant\_value \in PARTICIPANT \rightarrow VALUE \wedge$
  $initial\_values \in PARTICIPANT \rightarrow \mathbf{seq}(VALUE)$

The *CAACTIONS* abstract machine defines operations associated with the execution of CA actions: creation, termination, nesting and composition of CA actions, message exchange between participants, and exception handling. An abstract set *CAACTION* of all possible CA actions is introduced together with subset *caaction* of the CA actions that are running at a given state of the system. Three types of CA actions are distinguished: the top-level, nested and composed CA actions; two variables are used to memorize the nested and the composed CA actions:

$is\_nested \in caaction \leftrightarrow caaction \land$
$is\_composed \in participant \rightarrow\!\!\!\!\!\!\rightarrow caaction$

The state of the *CAACTIONS* abstract machine is defined by the following attributes:

- CA actions have a mode (referred to as *state*) that can be normal (if all the participants are in normal mode) or exceptional (if all the participants are in exceptional mode).

  $CAACTION\_STATE = \{caa\_normal, caa\_exceptional\}$
  $caaction\_state \in caaction \rightarrow CAACTION\_STATE$

- Each CA action has a set of participants and each of them participates to a sequence of nested CA actions:

  $participant\_of\_caaction \in caaction \rightarrow \mathcal{P}(participant)$
  $caaction\_of\_participant \in participant \rightarrow \mathbf{seq}(caaction)$

- CA actions access several external objects:

  $caaction\_ext\_objects \in caaction \leftrightarrow objects$

Several invariant properties of the CA actions have been identified and specified. They are written as constraints on the variables of the abstract machines. The state transformations associated with the execution of CA actions are further defined by the following preconditioned operations[4]:

- create_{main,nested,composed}_caaction: initiates a CA action, either top-level, nested or composed;
- {send,recv}_message(participant,participant,message): sends or receives a message from one participant to a peer participant;
- {read,write}_object(participant,object,{function}): reads, writes the value of an external object;
- raise_exception(participant,exception), propagate_exception(participant): raises, propagates an exception;
- abort_{main,nested,composed}: aborts a CA action, sending an abort message to all its external objects;
- terminate_{main,nested,composed}_{normal,exceptional} (caaction): terminates a CA action, either in a normal or in an exceptional state.

Invariant properties and operations are specified in such a way that properties of the dependability mechanisms associated with CA actions are enforced.

[4]Braces are used to denote multiple distinct operations.

4 Dependability Properties

The dependability mechanisms embedded within CA actions fall into three categories: (i) transactional access to external objects, (ii) atomicity of CA actions, and (iii) coordinated exception handling.

Transactions on External Objects

Access to external objects within CA actions are performed according to classical nested transaction rules. The operation that creates a top-level CA action initiates the transaction on external objects associated to the CA action:

add_objects(*obj*) =
PRE
  $obj \subseteq OBJECT \land obj \cap object = \emptyset$
THEN
  $values := values \lhd\!\!\!\!- obj \times \{begin\} \;||$
  $object := object \cup obj$
END;

Participants *setpar* of nested CA action *caa1* can only access subset *setobj* of external objects associated to containing CA action *caa2*. This constraint is ensured with the following precondition of the *create_nested_caaction(caa1,caa2,setpar,setobj)* operation:

$\forall obj.(obj \in setobj \Rightarrow obj \in caaction\_ext\_objects[\{caa2\}])$

Then, the operation initiates a nested transaction on the external object (*add_nested_object(setobj)*). When a (possibly nested) CA action terminates its execution normally (*terminate_caaction(caaction)*), it commits transactions on external objects:

terminate_transaction(caaction_ext_objects[{caaction}],commit)

On the other hand, if the CA action terminates exceptionally or aborts, all the transactions that it initiated on external objects are aborted as well:

terminate_transaction(caaction_ext_objects[{caaction}],abort)

Note that nested transactions are aborted recursively by the underlying transactional support of external objects.

Atomicity of CA Actions

Cooperation of participants is encapsulated inside atomic computation units using nested or composed CA actions.

The following invariant property states that participants of nested CA action *caa1* are also participants of containing action *caa2*:

$$\forall (caa1, caa2).((caa1 \in caaction \wedge caa2 \in caaction \wedge (caa1,caa2) \in is\_nested) \Rightarrow participants\_of\_caaction(caa1) \subseteq participants\_of\_caaction(caa2))$$

In the case when a participant invokes a composed CA action, participants *setpar* of the composed CA action must not be involved in any other CA action:

$$setpar \cap participant = \emptyset$$

Communication between participants *p1* and *p2* within a CA action is realized by message exchanges. Preconditions of the *send_message* operation set the rules of message exchange that is only allowed between participants of the same (possibly nested) CA action. The participants must be in the same state (normal or exceptional). Finally, a participant that is in a waiting state (i.e., waiting for a composed CA action to terminate) cannot send or receive a message:

$$caaction\_of\_participant(p1) = caaction\_of\_participant(p2) \wedge \mathbf{last}(participant\_state(p1)) \neq waiting \wedge \mathbf{last}(participant\_state(p1)) = \mathbf{last}(participant\_state(p2)) \wedge$$

Rules of nesting and composition are further specified with the following preconditions of the CA action termination operation stating, that a CA action terminates when all embedded nested and composed CA actions have terminated:

$$caaction \notin \mathbf{ran}(is\_nested) \wedge caaction \notin \mathbf{ran}(is\_composed)$$

Furthermore, a participant can only enter one sibling nested CA action at a time, which means that all participants in *setpar* willing to enter a nested CA action are in the same containing CA action:

$$\mathbf{card}(\mathbf{ran}(\{ p, c \mid p \in setpar \wedge c \in CAACTION \wedge c = \mathbf{last}(caaction\_of\_participant(p)) \})) = 1$$

Finally, the participants willing to enter a nested CA action must all be in the same state, normal or exceptional:

$$\mathbf{card}(\mathbf{ran}(\{ p, state \mid p \in setpar \wedge state \in PARTICIPANT\_STATE \wedge state = \mathbf{last}(participant\_state(p))\})) = 1 \wedge \forall (p).(p \in setpar \Rightarrow \mathbf{last}(participant\_state(p)) \neq waiting)$$

Coordinated Exception Handling

The following invariant ensures that a CA action is set to an exceptional state if all of its participants are in the exceptional state. Note that the participant can be in a waiting state following a call to a composed CA action, in which case case the test is performed on the last state of the participant before the call:

$$\forall (caa).\ (caa \in caaction \wedge caaction\_state(caa) = caa\_exceptional \Rightarrow \forall (p).(p \in participant\_of\_caaction(caa) \Rightarrow ((\mathbf{last}(participant\_state(p)) \in EXCEPTIONAL\_STATE) \vee ((\mathbf{last}(participant\_state(p)) = waiting \wedge \mathbf{last}(\mathbf{front}(participant\_state(p))) \in EXCEPTIONAL\_STATE)))))$$

Exception raising and propagation (to other participants) is realized by two operations defined in the *CAACTIONS* machine. The *raise_exception* operation requires that the participant and the CA action are in the normal state, and sets the participant's state to exceptional:

raise_exception (*p*, *exception*) =
PRE
  $p \in participant \wedge$
  $exception \in EXCEPTIONAL\_STATE \wedge$
  $\mathbf{last}(participant\_state(p)) = normal \wedge$
  $caaction\_state(\mathbf{last}(caaction\_of\_participant(p))) = caa\_normal$
THEN
  set_participants_state($\{p\}$, *exception*)
END;

The *propagate_exception* operation is then called to propagate the exception to all participants of the CA action.

If a CA action terminates in an exceptional state, all transactions on external objects are aborted. If this CA action is a nested or composed one, then the participant in the containing CA action raises an exception by calling the *raise_exception* operation (i.e., the exception is *signalled* to a higher level).

5. From the B Specification to the Development Support

In order to have an implementation of the CA action's run-time support, the abstract machines are refined. At the end of the refinement process, we have a set of executable codes that correspond to the implementation of the operations defining the B machines, offered as a programming library. Note that, when implementing the CA actions, some existing libraries such as drivers for running transactions are used. For all these libraries, what is usually known is the interfaces of the offered methods. In order to be able to prove the correctness of the implementation it would be necessary: (i) to have in addition the formal specification of the behavior of these methods and (ii), to prove that the refinements of the machines that use these methods are correct (in the B sense). During the refinement, the nondeterminism will be reduced (e.g., by introducing of message queues).

The preconditions have to be relaxed in order to take into account all the possible cases.

We introduce an XML-based language derived from the B specification in order to provide to the developer, which may not necessarily have a B knowledge, a convenient declarative language for building CA action-based systems.

Each XML document defines a main CA action which contains composed and nested CA actions, where composed CA actions are defined in a distinct document. The external objects are also declared.

```
<caaction name="nmtoken"?  >
  <composedActions> ?
    <action name="qname" /> *
  </composedActions>
  <nestedActions> ?
    <nested name="nmtoken" /> *
  </nestedActions>
  <external> ?
    <object name="nmtoken" /> *
  </external>
```

Each participant is then declared within the CA action, defining its local variables, which correspond to values used in the B specification (*participant_value*), and its behaviour composed of a normal and an exceptional parts. A participant executes this exceptional part when the CA action is in an exceptional state, which means that the exception raised by a participant has been automatically propagated to all of them.

```
<participants>
  <participant name="nmtoken"> +
    <var>
      <element name="nmtoken" type="qname" /> *
    </var>
    <behavior>
      <normal>
        Statements *
      </normal>
      <exceptional handle="qname"> *
        Statements *
      </exceptional>
    </behavior>
  </participant>
</participants>
```

The statements declared in the behaviour part of the participant's definition describe a sequence of operations to be executed. Each operation corresponds to the implementation of an operation of the *CAACTIONS* machine

```
Statements:
<invoke action="qname" input="qname"?
      output="qname"?  />
⇒ create_composed

<send rcpt="qname" input="qname" />
⇒ send_message

<recv from="qname" output="qname" />
⇒ recv_message

<call rcpt="qname" input="qname"?
      output="qname"?  />
⇒ {read,write}_object

<assign element="qname" value="XPATH" />
⇒ set_value

<raise exception="qname" message="qname"?  />
⇒ raise_exception

<nest nestedaction="qname">
  <behavior>
    <normal>
      Statements *
    </normal>
    <exceptional handle="qname"> *
      Statements *
    </exceptional>
  </behavior>
</nest>
⇒ create_nested
```

The above language enables the development of systems using CA actions. It hides the details of the dependability mechanisms such as automatic exception propagation, and more generally the behavior of the operations described in the B specification and that will be executed during runtime. Furthermore, it enables static analysis to be performed in order to verify the structural properties of a given system described in the invariant of the specification.

6. Conclusion

This paper has presented both how to specify dependability mechanisms using the B formal method and a development support relying on an XML-based language and on the refinement process of B.

We have considered the use of CA actions that have been proved useful for building dependable systems. We have defined a generic formal specification using the B method, defining systems composed of several CA actions that make concurrent accesses to external objects. B was chosen because of its powerful theorem proving ability and because of availability of a number of mature tools. We have shown how to specify the following dependability mechanisms of CA actions: (i) constraints related to the atomic access to external transactional objects, (ii) encapsulation of computations inside atomic action units ensured through action nesting and composition and (iii), properties related to the behaviour of the system in case of exception occurrences.

The XML-based language is to be used for describing a specific system instance such as a travel agency system, by giving the behavior of each participant. Refinement of the B specification is exploited for offering a correct implementation of the language. This includes static analysis and run-time support, whose correct implementation further depends on the one of third-party libraries.

Up to now several implementations of CA actions have been proposed and experimented with, but mainly on closed systems [9, 2]. We are working on an implementation of CA action-based systems to be defined as a composition of Web services such as the travel agency. This kind of systems clearly needs new dependability properties (e.g.,

relaxed atomicity properties for accessing external objects). We intend to use this initial B specification to study such properties.

Acknowledgments
This research was partially supported by the European IST DSoS (Dependable Systems of Systems) project (IST-1999-11585)[5].

References

[1] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[2] D. Beder, A. Romanovsky, B. Randell, C. Snow, and R. Stroud. An Application of Fault Tolerance Patterns and Coordinated Atomic Actions to a Problem in Railway Scheduling. *ACM, Operating Systems Review*, 34(4):21–31, October 2000.

[3] R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, SE-12(8), 1986.

[4] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[5] D. Schwier, F. von Henke, J. Xu, R. Stroud, A. Romanovsky, and B. Randell. Formalization of the CA Action Concept Based on Temporal Logic. Design for validation (deva) basic esprit project. second year report. part 2, LAAS, France, 1997.

[6] F. Tartanoglu, V. Issarny, N. Levy, and A. Romanovsky. Dependability in the Web Service Architecture. In *Proceedings of the ICSE Workshop on Architecting Dependable Systems*, Orlando, USA, May 2002.

[7] J. Vachon, N. Guelfi, and A. Romanovsky. Using COALA to Develop a Distributed Object-Based Application. In *In the 2nd Int. Symposium on Distributed Objects and applications (DAO'00), P. Drew, R. Meersman, Z. Tari, R. Zicari (Eds.)*, pages 195–208, Antverp, Belgium, 2000.

[8] J. Xu, B. Randell, A. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Proceedings of the Twenty-Fifth IEEE International Symposium on Fault-Tolerant Computing*, 1995.

[9] J. Xu, B. Randell, A. B. Romanovsky, R. J. Stroud, A. F. Zorzo, E. Canver, and F. W. von Henke. Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions. In *Symposium on Fault-Tolerant Computing*, pages 68–75, 1999.

[5] http://www.newcastle.research.ec.org/dsos/

Layered Dependability Modeling of an Air Traffic Control System

Olivia Das, C. Murray Woodside
Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada
email: odas@sce.carleton.ca, cmw@sce.carleton.ca

Abstract

Quality attributes, such as performance and dependability of a software-intensive system are constrained by its software architecture. The combined performance and dependability (called performability) effects of an architecture can be evaluated by constructing a performability model that considers the failure/repair behavior and performance attributes of its components, interactions among the components and the fault tolerant approaches adopted. This paper constructs and analyzes a Dependable Layered Queueing Network (Dependable-LQN) performability model for a large-scale Air Traffic Control system. It demonstrates the capability of the appraoch, for evaluating the performability of a large-scale software architecture.

1. Introduction

The Dependable Layered Queueing Network (Dependable-LQN) model is a performability model for fault-tolerant distributed applications with a layered software architecture and a separate architecture for failure detection and reconfiguration. It considers the failures (repairs) of its application and management components, management connections and the application's layered failure dependencies together with the application performance. It combines Fault-Tolerant Layered Queueing Networks (FTLQN) and the Model for Availability Management Architecture (MAMA) [1]. FTLQN in turn extends the Layered Queueing Network model [2] (a pure performance model) with dependability related components and attributes. This paper describes its application to an Air Traffic Control (ATC) system.

An ATC system [3, 4, 5] is a large-scale complex distributed system which demands high availability and high performance. Its software architecture plays a key role in achieving its high quality requirements; we consider the architecture described in [3]. This is an *en route* system which controls aircraft from soon after takeoff until shortly before landing. Its end users are the air traffic controllers. It has a layered software architecture with one subsystem depending on another for services and it uses a separate management architecture for automatic failure detection and recovery. It is an important case study for us because Dependable-LQN model perfectly fits the choice for modeling such a complicated and large system.

The goal of this paper is to demonstrate the use of the Dependable-LQN model on a substantial system with strong requirements for performance and dependability. It also describes the use and scalability of a tool [6] that takes the layered software description and the management component interactions as input and solves the model analytically to generate results.

2. The Dependable-LQN Model

2.1. First part: FTLQN Model

Figure 1 illustrates an example of an FTLQN model with an example of a layered console application. There are seven tasks (concurrent operating system processes, represented as rectangles), "Console", "Application Server", "Console Server", "Log Server", "Database-A", "Database-B" (backup of "Database-A") and "Data Server". Each task runs on a processor represented by an ellipse. The task "Application Server" creates a report which involves reading from the database, requesting some kind of application data from the "Data Server" and then logging the report to the "Log Server". Tasks have one or more entries which are service handlers embedded in them ("Data Server" has "Get Application Data" and "Get Console Data"). A *service request* (represented by a rounded rectangle) has a single target server if there is no server redundancy, or it may have a set of redundant target

servers with a policy on how to use them.

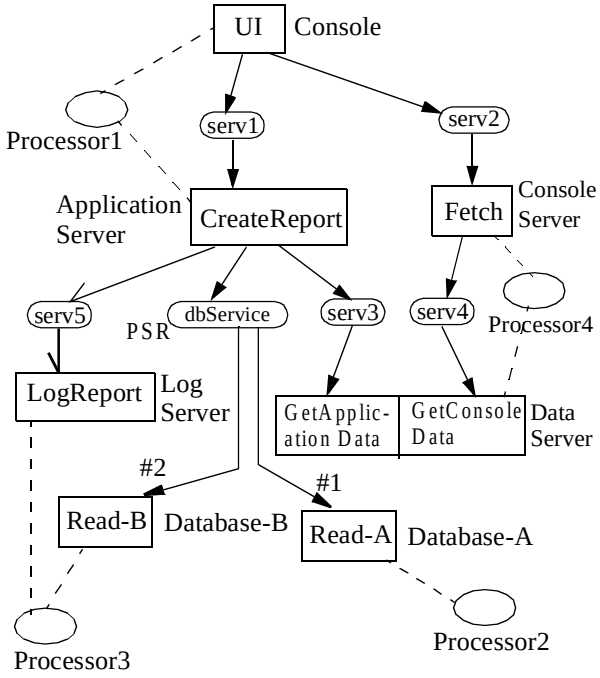

Figure 1. An FTLQN model

The different redundancy policies supported for a *service request* are:

- Primary-Standby Redundancy (PSR) with load sent only to the primary target
- Active Redundancy (AR) with load replicated and sent to all working targets
- Primary-Standby Active Redundancy (PSAR) with load replicated and sent to all the working targets where a designated primary does exist. This redundancy policy has been introduced in this paper. It is useful in cases where a client sends a service request to all the redundant servers (in Layer *i*) for close synchronization, however only the primary server of Layer *i* would send the service request to Layer *i*+1 in order to decrease the number of replicated requests to Layer *i*+1.
- Load-Balanced Redundancy (LBR) with load equally divided among all working targets

In Figure 1, "dbService" is a service requested by the entry "Create Report" for reading from the database. It has PSR policy where the priority of the target servers are labelled "#n" on the arcs going out from the service to the server(s). In this model, all the service requests are synchronous where the sender is blocked until the receiver replies. The model is restricted to being acyclic in order to avoid cycles of mutual waiting that may lead to deadlock.

Asynchronous service requests can also be accommodated where a client does not block after sending a request to the server. In Figure 1, "serv5" represents an asynchronous service request. In contrast to a synchronous service request where the failure of a client directly depends on the failure of the servers, an asynchronous service request does not add any failure dependencies. In order to add additional failure dependencies that cannot be represented by service-based dependencies, another abstraction called *depends* relationship (of types "AND" or "OR") can exist between any two entries (illustrated in Section 3).

The performance parameters for the FTLQN model are the mean total demand for execution on the processor by each entry and the mean number of requests for each interaction. The availability related parameters for this model are the probabilities of being in failed state for each component (either a task or a processor) of the application. The performance measures are usually associated with the tasks that only originate requests (e.g. "Console" task), also called *reference tasks*.

The FTLQN model shows the policy for redundant servers but the decision about where to forward the request is made by the fault management sub-system (not visible in this model) based on its knowledge of the state of the application components. This resolution of requests gives different operational configurations of the application. The probabilities of the operational configurations now depend on the fault management architecture, management subsystem failures, and as well as on the application.

The FTLQN model can be solved to compute steady-state measures, e.g. mean throughput of the system in presence of failures. The general strategy of the analysis is to compute the performance for each reachable configuration that has different choices of alternative targets for requests and combine it with the probability of each configuration occurring, to find the measures. For example, in Figure 1 if all the tasks are operational, then the configuration is the system as shown, but with Database-B, and its service requests (labelled #2) removed as they are not used. This configuration is a pure Layered Queueing Network

(LQN) performance model [2]. Each LQN model can be solved for different performance measures by the LQNS tool[2], based on extended queueing approximations. This strategy is similar to the Dynamic Queueing Network approach given in [7, 8] for queueing network models.

The next section describes the management components, their connections and how they are related to the application components.

2.2. Second part: MAMA Model

MAMA model has four classes of components: application tasks, agent tasks, manager tasks and the processors hosting them. They are connected using three different classes of connector:

- Alive-watch connector: This connector is used between two components in cases where the destination component would like to be able to detect whether the source component is alive or not. This may be achieved by periodic polls or "i-am-alive" messages. Usually, the source of this connector are the manageable application components.
- Status-watch connector: In cases where a destination component would like to know about the liveness of the source component and also wants to collect failure data about other components in the system that has been gathered by the source component, this connector is used. An example would be a connector from an agent task to a manager task.
- Notify connector: This connector is used for cases where the source component would like to send or forward reconfiguration commands to its sub-ordinate destination component (for example, a manager sending commands to an agent or an agent forwarding a command to an application task) or conveying management information to its peer (for example, a manager sending information about the domain it manages to another manager).

Cycles may occur in a MAMA model; we assume that the flow of information is managed in a way so as not to cycle or ping-pong. It is also assumed that if a task watches a remote task, then it also watches the processor executing the remote task in order to differentiate between a task failure and a processor failure.

Figure 2 shows the graphical notation used in this work for MAMA components and connectors.

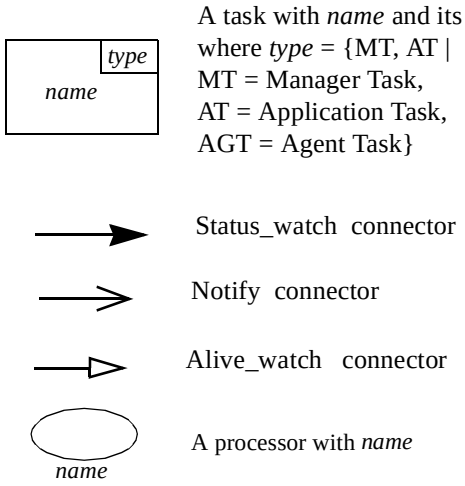

Figure 2 Notation used here for the MAMA model

For this model, the failure probabilities can be provided for all management components and the connectors between them.

3. Dependable-LQN Model of an ATC En Route System

An airspace controlled by a single ATC facility is administratively divided into sectors. For example, US airspace is serviced by 20 facilities each with a maximum of 118 sectors per facility. Each facility receives aircraft surveillance and weather data from radars and communicates with other external subsystems such as other en route facilities. Inside each facility, air traffic services are divided among four subsystems: Surveillance Processing Service (that receives radar data and correlates it with individual aircraft tracks) is provided by Radar subsystem directly connected to radars, Flight Plan processing and Conflict Alert services is provided by Central subsystem, Display service (which displays aircraft position information obtained from radars and allows inputs from air traffic controllers to modify flight plan data or change display format) is hosted by the Console subsystem, Monitoring service (which provides the monitoring and control service for other ATC services ensuring their availability policies) is hosted by the Monitor and Control subsystem. There are up to four

consoles allocated for handling each sector. Fault-tolerance is achieved by software server groups. For example, there are up to four Display Management primary/standby active redundant servers per sector, three primary/standby active redundant Surveillance Processing servers, two primary/standby Flight Plan Management servers.

All the three redundant Surveillance Processing servers receive the raw radar data from the radars in parallel, however, only the primary would send the processed radar data to the Display Management servers running in the consoles. A PSAR redundancy policy have to be used to model this case.

Figure 3 shows some parts of a Dependable-LQN model based loosely on the description in [3]. Each bubble represents a process group with replication; the service nodes and redundant servers are not shown. The communications with replicas is made transparent by a process-to-process session manager (P2PSM) in each host (not shown here).

Some failure dependencies are implicit in the server request-reply dependencies. Others can be made explicit with a *depends* relationship, for instance that "display radar data" depends on "process radar data", even though the communications is asynchronous. This would be an *OR depends* relationship on the three radar processing replicas, since any one is sufficient.

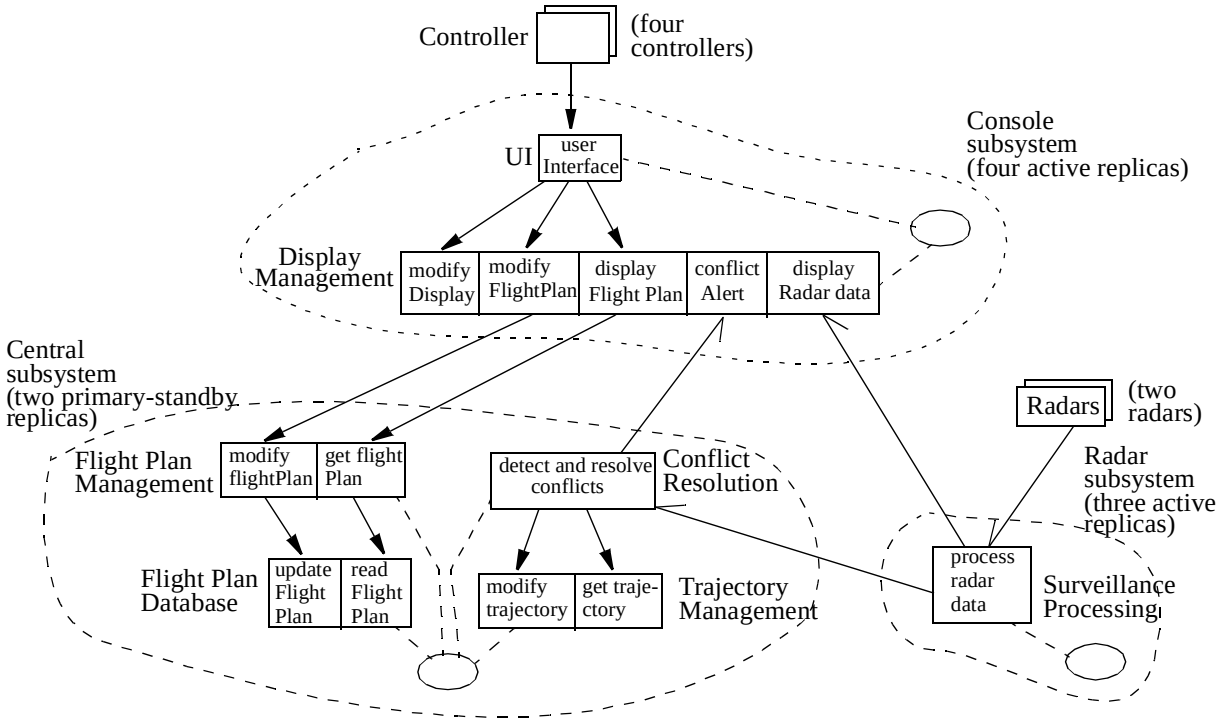

Figure 3 A Dependable-LQN model for an ATC en route system. Redundant server groups are not shown here.

The fault management architecture depends on a *group availability management* server (gSAM) on each processor which monitors all the software servers in its own processor and also monitors the other processors in its software server group. In MAMA terms, the gSAM servers maintain an *alive-watch*. This is supported by three redundant name servers which maintain a list of primary host processors for each server group. When a failure of a software server occurs in its processor, a gSAM server notifies other gSAM servers in its own group. Similarly, the failure of a processor is detected by the other gSAM servers in a group. Whenever a failure is detected, the gSAM server notifies the name servers which in turn notify the relevant P2PSM's so that they can retarget their service requests.

Figure 4 shows a portion of the MAMA model for Figure 3, including one replica of the Monitor and Control subsystem. The redundant servers and the interactions among the gSAM servers in a group are not shown here.

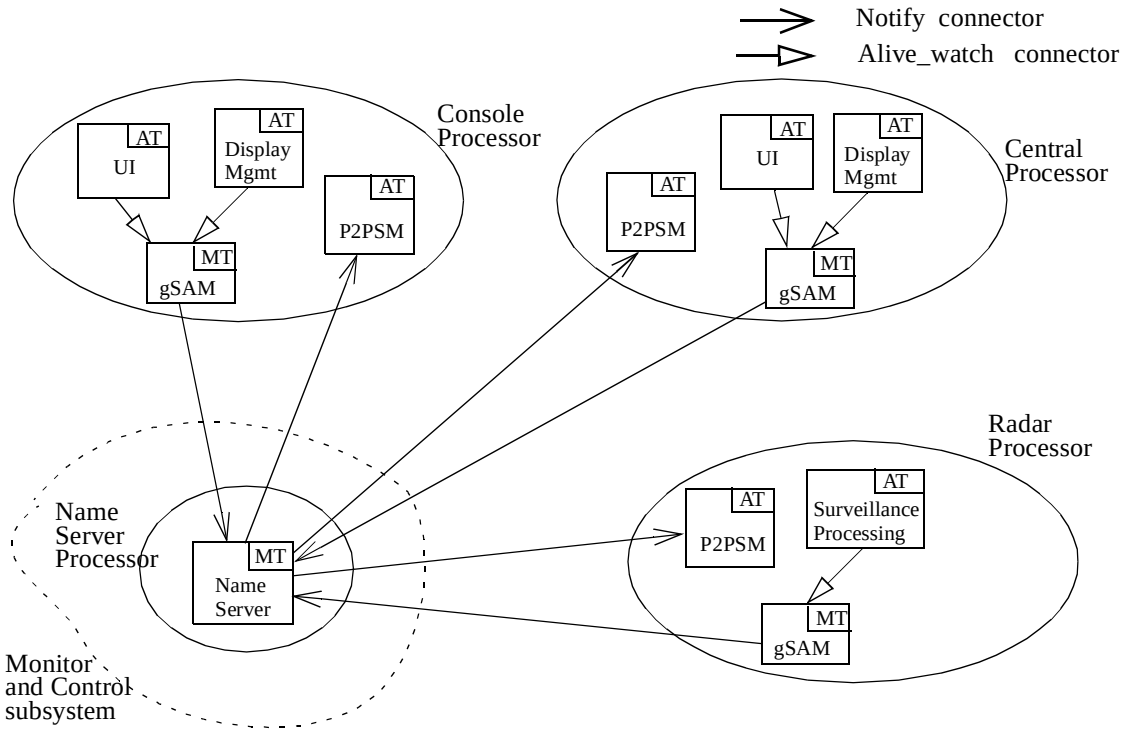

Figure 4. Portion of MAMA model for Figure 3. Redundant server groups and interactions among the gSAM servers in a group are not shown here.

A model for a sector has around 13 processors and 41 tasks (including the P2PSM and gSAM tasks). This is comparable to the other models that have been solved with the Dependable-LQNS tool [6]. The analysis results will be described at the workshop.

4. Conclusion

This paper described a Dependable-LQN model for evaluating performability of a large-scale Air Traffic Control system. The value of the work lies in showing how the Dependable-LQN model can be used to evaluate whether an architecture meets its performability goals or not and also to demonstrate the scalability of the model solving tool. Our future research includes considering the detection and recovery delays in addition to the management architectural limitations and its failures into the analysis.

5. References

[1] O. Das and C. M. Woodside, “Modeling the coverage and effectiveness of fault-management architectures in layered distributed systems”, IEEE International Conference on Dependable Systems and Networks (DSN'2002), June 2002, pp. 745-754.

[2] G. Franks, S. Majumdar, J. Neilson, D. Petriu, J. Rolia, and M. Woodside, “Performance Analysis of Distributed Server Systems”, in the Sixth International Conference on Software Quality (6ICSQ), Ottawa, Ontario, 1996, pp. 15-26.

[3] F. Cristian, B. Dancey and J. Dehn, “Fault-Tolerance in Air Traffic Control Systems”, ACM Transactions on Computer Systems, 14(3), August 1996, pp.265-286.

[4] L. Bass, P. Clements and R. Kazman, Software Architecture in Practice, Addison-Wesley, 1998.

[5] A. S. Debelack, J. D. Dehn, L. L. Muchinsky and D. M. Smith, “Next generation air traffic control automation”, IBM Systems Journal, 34(1), 1995, pp. 63-77.

[6] O. Das and C. M. Woodside, “Dependable LQNS: A

performability modeling tool for layered systems", submitted for IEEE International Conference on Dependable Systems and Networks (DSN'2003).

[7] B. R. Haverkort, I. G. Niemegeers and P. Veldhuyzen van Zanten, "DYQNTOOL: A performability modelling tool based on the Dynamic Queueing Network concept", in Proc. of the 5th Int. Conf. on Computer Perf. Eval.: Modelling Techniques and Tools, G. Balbo, G. Serazzi, editors, North-Holland, 1992, pp. 181-195.

[8] B. R. Haverkort, "Performability modelling using DYQNTOOL+", International Journal of Reliability, Quality and Safety Engineering., 1995, pp. 383-404.

Design for Verification: Enabling Verification of High Dependability Software-Intensive Systems[1]

Peter C. Mehlitz
Computer Sciences Corporation
pcmehlitz@email.arc.nasa.gov

John Penix
NASA Ames Research Center
John.J.Penix@nasa.gov

Lawrence Z. Markosian
QSS Group, Inc.
lzmarkosian@email.arc.nasa.gov

Abstract

Strategies to achieve confidence that high-dependability applications are correctly implemented include testing and automated verification. Testing deals mainly with a limited number of expected execution paths. Verification usually attempts to deal with a larger number of possible execution paths. While the impact of architecture design on testing is well known, its impact on most verification methods is not as well understood. The Design for Verification approach considers verification from the application development perspective, in which system architecture is designed explicitly according to the application's key properties.

The D4V hypothesis is that the same general architecture and design principles that lead to good modularity, extensibility and complexity/functionality ratio can be adapted to overcome some of the constraints on verification tools, such as the production of hand-crafted models and the limits on dynamic and static analysis caused by state space explosion.

1. Introduction

High dependability systems can be characterized by the need to satisfy a set of key properties at all times. This includes standard properties like absence of deadlocks, and application specific properties such as guaranteed responses or "correct" results.

Testing at various "scope levels" is usually the preferred way to check deterministic computation results, but this approach is of limited value for checking properties of concurrent programs. Since the scheduling behavior typically cannot be controlled from the testing environment, standard defects like race conditions and deadlocks can easily be missed by testing. This is an important case in which automated system verification comes into play.

If verification is left until system integration, the target system often is already too big and complex for verification tools to handle it directly, which is especially true for static analysis and model checking of concurrent programs. As a result, target systems need to be modeled in order to apply the tools, an expensive process that also has the potential for introducing fidelity problems. Because of the associated costs, model-assisted verification can easily degenerate into a one-time-effort, which simply does not match the evolutionary life cycle of large systems. Lack of support for efficiently checking formal properties in turn leads to minimal inclusion of such properties in the specification and design phases, which further decreases the effectiveness and value of automated verification.

The *Design for Verification* (D4V) hypothesis is that the same general architecture and design principles that lead to good modularity, extensibility and complexity/-

[1] The research described in this report was performed at NASA Ames Research Center's Automated Software Engineering group and is funded by NASA's Engineering for Complex Systems program.

functionality ratio can be adapted to overcome some of the constraints on verification tools.

The context of our D4V work is the development of practical tools and methodologies based on source code model checking technologies[2] such as Java PathFinder[1], but the D4V concepts are intended to be applicable with a broad range of verification approaches.

2. Traditional Approaches

One verification approach is based on architecture design documentation (e.g. UML/OCL), with the intent of producing correct architectures, from which code is more likely to correctly implemented.

Another approach to overcoming the scalability problem for verification tools is to improve the tools. In the case of model checking, this involves techniques like abstraction, slicing and partial order reduction. These are necessary techniques for handling real applications.

Even with these sophisticated approaches, it still is too easy to design applications so that they cannot be applied. This is particularly due to the fact that in contemporary programming environments, an increasing amount of functionality is shifted from stand-alone applications into libraries and frameworks, which either exceed the size constraints of the verification tools, are or unavailable in a suitable format to apply these tools.

3. The Design for Verifiability Approach

Our approach complements traditional approaches in that we explicitly add verification- and testing-specific considerations to the architecture design phase. The general idea is to map key requirements of the specification directly to dedicated, mostly invariant design components, which can be verified separately. The goal is to turn system verification into a development co-process like regression testing.

We try to achieve this goal by using domain specific design pattern collections. Each pattern instance comes with a set of formal usage rules and guarantees. Usage rules are subject to automated checks, mostly using contracts (preconditions, post-conditions, and invariants) and static analysis. The pattern selection process itself is driven by evaluation of the guaranteed properties against the key specification requirements. While this does not ensure arbitrary, application-specific properties, it gives a much better understanding of the formal correctness model early in the development phase.

Since these key patterns constitute design elements that are mostly invariant during the implementation and evolution of the system, the verification results are not lost, and the tools can be re-applied at later stages of the system lifecycle without modeling efforts.

The program design is centered around three concepts: *extension points*, *conceptual branch points*, and *check points*.

Extension points identify the components that can be used to extend the functionality of the application without breaking its design or causing feature bloat. Extension points include potential base classes with their overridable methods, and major delegation objects with their associated interfaces, both with their corresponding implementation constraints. Extension points allow property verification during later stages of the lifecycle, when system functionality is often extended without having a suitable design infrastructure for these extensions.

Conceptual branch points are the locations that are relevant for both testing and model checking. This includes non-deterministic operations, in particular potentially blocking or context switching instructions in multi-threaded programs, which are preferred targets for backtracking. We are investigating program designs that turn these branch points into choice generator calls, enabling systematic testing and model checking in the real execution environment. This is achieved by turning implicit, execution environment specific behavior (like thread scheduling) into explicit delegation objects (the generators). To verify multi-threaded programs, this can be used to effectively turn threads into co-routines, which are systematically switched inside of the generator objects. This approach is based on the assumption that (a) concurrent systems should be designed around their synchronization/communication points, and (b) these operations are usually encapsulated into APIs or specific language constructs anyway (i.e. can be easily intercepted).

Check points describe the application-specific correctness model, and map to freely-placeable assertions. They can be thought of as required-to-be consistent, usually global states, and should be mappable from/to the system specification. A typical example is a check for memory leaks after a certain operation has been completed, to verify constant-space execution properties. While evaluation of check points is straightforward (provided the programming environment has a assertion mechanism), reachability analysis and side-effect detection of check points is again subject to tool support.

It is important to note that D4V does not attempt to introduce a radically new design approach, but instead

[2] *Source code model checkers* take the source code of an application (or some transformation of it) as the model. Examples include SPIN and SLAM for C, and Java PathFinder for Java

extends existing "best design practices" towards verifiability and testability. This comes with two intentional side effects.

First, deliberate use of design patterns tends to improve modularity and reduce "accidental complexity". This in general makes the system more understandable and unit-testable, and reduces the relevant state space for verification tools.

To quantify this aspect, we have taken a small. moderately object-oriented. autonomous robot application and re-designed it using design patterns.

	old version	***new version***
classes	82	37
interfaces	1	10
NCLOC	5926	1745
max WMC	397	56
sum WMC	1426	389
threads	6	2

Both systems were written in Java. WMC stands for "Weighted Methods per Class" and represents the sum of the cyclomatic complexities of its methods.

The pattern oriented re-design not only resulted in the anticipated extensibility and test-suitability (esp. for unit tests), but also showed a significant reduction in over-all size, and a elimination of the complexity "hot spots"(max WMC). Just the decrease in threads makes the system more understandable, less error-prone (deadlocks), and more verifiable (state space).

Second, D4V attempts to overcome the traditional gap between design/development and testing/verification. Because designers gain more scalable tools and tests, they are encouraged to think more about application correctness.

4. Project status

The D4V project is in an early stage. The current focus is on the development of a suitable design pattern system. Our first target domain is event driven, observable, state-model based systems.

[1] W. Visser, K. Havelund, G. Brat, S. Park. "Model Checking Programs", *Proceedings of the 15th International Conference on Automated Software Engineering (ASE)*, Grenoble, France, September 2000.

Toward a Framework for Classifying Disconnected Operation Techniques

Marija Mikic-Rakic Nenad Medvidovic

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781 U.S.A.
{marija, neno}@usc.edu

Abstract

Distributed, decentralized, and mobile systems are highly dependent on the underlying network. Due to network connectivity failures, these systems must address the problem of disconnected operation, i.e., continued functioning in the absence or near-absence of network accessibility. A number of existing approaches provide support for disconnected operation by employing different techniques. What is currently missing, however, is a general understanding of the applicability of these techniques to different kinds of software systems, and the manner in which they affect the overall system dependability. This paper strives to improve that understanding. We present a framework for classifying disconnected operation solutions and assess several representative approaches according to the proposed classification. This study highlights several pertinent areas that are currently not supported, helping to motivate our future work.

1. Introduction

The emergence of mobile devices such as portable notebook computers, hand-held personal digital assistants (PDAs), and mobile phones, and the advent of the Internet and various wireless networking solutions make the computation possible anywhere. However, new challenges arise for software systems executing in such environments: they are becoming highly distributed, decentralized, and mobile, and therefore highly dependent on the underlying network. Unfortunately, network connectivity failures are not rare: mobile devices face frequent and unpredictable (involuntary) connectivity losses due to their constant location change and lack of wireless network coverage; the costs of wireless connectivity often induce user-initiated (voluntary) disconnection; and even the highly reliable WAN and LAN connectivity is unavailable between 1.5% and 3.3% of the time [25].

For this reason, network-dependent systems are challenged by the problem of *disconnected operation*, where the system must continue functioning in the (near-)absence of the network. Disconnected operation forces systems executing on each individual host to operate independently from other network hosts. This presents a major challenge for the software systems that are highly dependent on network connectivity, because each local subsystem is usually dependent on the availability of non-local resources. Lack of access to a remote resource can halt a particular subsystem or even make the entire system unusable.

There are several possible solutions to increasing the dependability of highly distributed and decentralized software systems in face of the connectivity losses:

- make remote data available locally,
- make remote code available locally,
- make remote dynamic system state available locally,
- reroute the communication in cases of partial disconnection from the network, and
- delay remote interactions until the connection is reestablished.

The goal behind each of these solutions is to temporarily mask the absence of connection by mimicking the system's continuous connectivity. The inconsistencies that may result from applying these solutions need to be resolved once the connection is re-established. Each of these solutions can be provided in a number of different ways, depending on the nature of the target application and on those aspects of the application's dependability that are of primary concern (e.g., availability, performance, scalability, security).

Most commonly used techniques for supporting disconnected operation are:

- *Caching* – locally storing remote data once it has been accessed in anticipation that it will be needed again [12],
- *Hoarding* – prefetching the likely needed remote data prior to disconnection [13],
- *Queueing remote procedure calls* – buffering remote, non-blocking requests and responses during disconnection and exchanging them upon reconnection [11],
- *Deployment and redeployment* – installing, updating, or relocating a distributed software system [1],
- *Replica reconciliation* – synchronizing the changes made during disconnection to different local copies of the same component [12],
- *Code mobility* – dynamic change of the bindings between code fragments and locations where they are executed [8].

What is currently missing, however, is a general understanding of the applicability of these techniques to different kinds of software systems, how and under what conditions they may be used (possibly in concert), how they affect the overall system dependability, and so forth. Also unclear is the applicability of these techniques in the growing class of architecture-centric, component-based software systems [18,19]. We believe that an understanding of these issues can help both to streamline existing and to develop future techniques in support of this area. This paper strives to improve that understanding. We present a framework for classifying disconnected operation solutions. We have performed an extensive study of existing techniques and identified a common set of criteria for their classification. In

turn, we have assessed several representative solutions according to the proposed classification. While this is still a work in progress, it has already clearly identified areas not addressed by current solutions. These areas, coupled with a study of the compatibility of different techniques, will frame our future research agenda.

The rest of the paper is organized as follows. Section 2 presents an overview of existing disconnected operation techniques. Section 3 describes our classification framework and assesses several representative solutions based on the identified criteria. The paper concludes with a discussion of open issues that will frame our future work.

2. Background

In this section we present an overview of existing disconnected operation solutions. They are organized according to the general approach they adopt.

2.1. Availability of data

2.1.1. Distributed file systems. Most of the early work on disconnected operation has been in the area of distributed file systems. Coda [12], Ficus [9], and D-NFS [6] have included extensive support for distributed file replication during disconnection and synchronization of replicas upon reconnection. These approaches use techniques such as caching and hoarding for file replication, and logging and version vectors for replica reconciliation. D-NFS and Coda also introduced the notion of an *agent*, which represents an intermediary between client and server components that handles their interaction during disconnection. An agent operates in two modes: connected and disconnected. In the connected state the agent forwards all of the client requests to the real server. This allows the agent to monitor client operations and prepare the application for disconnected operation. In the disconnected mode, the agent performs the necessary tasks (such as logging the client operations) needed to synchronize the replicas upon reconnection. While Coda and D-NFS focus on client-server applications, Ficus provides support for more general, peer-to-peer applications.

PFS [4] provides support for partially connected operation using a three-tier model, where an intermediary PFS host is inserted between a file server and a mobile client. PFS is a pseudo server for the mobile client and a pseudo client for the file server. This extra level of indirection enabled an efficient solution for partial disconnection, with a tolerable overhead during full connection. PFS provides a generic interface for application-directed adaptation to varying network quality of service requirements.

2.1.2. Distributed databases. In the area of distributed databases, disconnected operation has been addressed by approaches such as Thor [2], and Bayou [22].

In Thor, a relevant subset of database objects is cached prior to disconnection. Each transaction is logged during disconnection, but the clients can only perform "weak" transactions while disconnected. These transactions are committed only if they do not conflict with the transactions performed on the server.

Bayou is a platform of replicated, highly available, variable-consistency, mobile databases on which collaborative applications are built. Bayou focuses on providing application-specific conflict detection and resolution.

The main focus of both distributed file system and database approaches has been on supporting continuous availability of (passive) data only. Hence, they do not provide support for applications whose mode of operation during disconnection depends also on the availability of remote code and/or remote system state.

2.2. Availability of code

Providing continuous availability during disconnection by employing code mobility techniques has been the focus of approaches such as Rover [11], Jamp [23], Mobile extensions (ME) [3], Odyssey [17], and FarGo-DA [24].

The Rover toolkit provides two major programming abstractions: relocatable dynamic objects (RDOs) and queued remote procedure calls (QRPC). RDOs represent mobile code that can be dynamically loaded from a remote server and cached locally, while QRPC queues remote requests during disconnection and dispatches them upon reconnection. Rover uses version vectors to detect conflicts between different instances of a given RDO.

Jamp provides abstractions that support the migration of groups of objects and classes between nodes of the network. However, Jamp does not support object replication and, since mobile objects can only be in one location at a time, does not provide facilities for conflict resolution.

Mobile extensions (ME) provides location-independent, extensible facilities for deploying Web-based services. ME makes use of caching, hoarding, asynchronous messaging, and application-level adaptation to cope with network failures. This approach provides flexible and automatic resource management, since its employed techniques such as caching and hoarding can dramatically increase resource demands.

Odyssey is a set of extensions to the NetBSD operating system to support adaptation for a broad range of mobile information access applications. Odyssey provides monitoring of various system resources, notifies running applications of relevant changes, and enforces resource adaptation decisions. However, each application independently decides how to adapt to the notified change.

FarGo-DA is a programming model and a runtime infrastructure for automatic reconfiguration of an application during disconnection. FarGo-DA advocates disconnected operation awareness at system's design time. Additionally, since FarGo-DA is targeted at resource-constrained platforms, its main consideration is limited memory on such platforms. For this reason, FarGo-DA proposes the use of multi-modal components in which the developer specifies separate subcomponents to be used during connection and disconnection. Usually the subcomponent used during disconnection provides a subset of connected-mode functionality. Additionally, FarGo-DA assumes that the application developer provides the conflict resolution code as part of the multi-modal component.

It should be noted that none of the described code mobility techniques support automated selection of components that should be migrated, nor do they perform any analysis of the effects of code mobility on the running system.

2.3. Ad-hoc networking

In the area of ad-hoc networking, several approaches

have been proposed, including packet rerouting protocols (FORP [21]), predictive connection management with user input (PCP [14]), and adaptive wireless and mobile networking (Monarch [10]). These approaches are focusing on the mobility of the (human) user and are providing different techniques for rerouting when a mobile host changes location. However, they do not focus explicitly on disconnected operation, and therefore do not provide any support for the operation of a mobile client if it is completely disconnected from the network.

2.4. Other techniques

Some techniques have taken a more general approach to disconnected operation in which network disconnection represents only one point of failure in a given system. Failures of individual components are also examined and treated in a manner comparable to network failures. The goal of these techniques is to ensure that the system's operation will degrade gracefully in the face of failures. Replication is primarily used for increasing the reliability of individual components. An example such approach is RoSES [20], which provides a scalable framework for the analysis and design of system-wide graceful degradation.

3. Classification framework

3.1. Description

The overall structure of our proposed classification framework for disconnected operation approaches is organized around eight *categories*, as shown in Figure 1. Each category may have multiple *dimensions*, *subdimensions*, and *values*. The values are not necessarily mutually exclusive, meaning that a single approach may have zero, one, or multiple values corresponding to a given category, dimension, or subdimension. It should be noted that it was not our goal to identify all possible values in Figure 1 but rather to extract representative values from the existing approaches. We expect that the list of values will grow as we refine our classification framework. Missing approaches corresponding to values in Figure 1 indicate that no existing approach supports the corresponding property. Various other techniques exist that may be effectively applied in the context of disconnected operation (e.g., dynamic software architectures [5]). However, we feel that including such techniques in our framework at this time would be speculative as their effectiveness in this setting has not been demonstrated. In the remainder of the section we discuss each of the proposed classification categories in more detail.

3.1.1. Connectivity. Connectivity encompasses information about the nature of disconnection that a given approach supports (*type*) as well as how the *detection* of disconnection is achieved.

Connectivity type is further divided into two subdimensions: *predictability* and *degree* of connectivity. Predictability can have two values: *anticipated* and *sudden*. In cases of anticipated disconnection the system is aware that disconnection is going to occur, and usually can predict when it will happen [14]. In cases of sudden disconnection, the system is unaware of the disconnection beforehand.

We have identified the following values for the degree of connectivity: *total*, *partial*, and *low-bandwidth*. In cases of total disconnection, a given host is completely disconnected from the network. In cases of partial disconnection, the host is disconnected from the remote host with which it communicates, but there may be other hosts in the system to which this host is still connected, or can be connected. In cases of low-bandwidth connection, the host is connected, but through a low throughput connection. Low throughput connections necessitate the use of special techniques (e.g., compression) that would minimize the use of bandwidth.

Connectivity detection is further divided into two subdimensions: *accuracy* and *source*. Accuracy denotes whether disconnection is detected with no loss of data, by losing a single remote invocation (event), after which the subsystem recognizes that it has been disconnected and adjusts, or by losing multiple remote invocations. There are three possible sources of disconnection detection: (1) external agent, denoting a source (e.g., OS service) that is not a part of a given disconnected operation approach; (2) per host, denoting detection if an entire given host gets disconnected; and (3) per component, denoting detection of disconnection of individual software components.

3.1.2. Component types. The component types category includes information about the kinds of software components whose availability in the face of disconnection is supported by a given approach. This category is further divided into *active* and *passive* components. Active components can be *computation*, *communication*, *coordination*, or *interface* components, while passive components can be *files* or *dynamic data structures*.

3.1.3. Architecture. The approaches described in Section 2 use techniques such as component replication, migration, or network rerouting to increase the dependability of software systems during disconnection. However, these changes to a software system's architecture may have unforeseen effects on the running system. It is thus important to *analyze* the effects of the proposed changes prior to enacting them.

Static analysis may use (partial) architectural models to assess the validity of proposed run-time architectural changes prior to their deployment, possibly disallowing the changes. Dynamic analysis refers to the analysis performed after the deployment, of the effects of the performed modifications on the running target system. These techniques are described in more detail in [15].

In addition to the need for analyses, most of the approaches described in Section 2 impose certain architectural *topology* restrictions on the supported applications, such as client-server or peer-to peer. An important decision factor in determining the most suitable approach for a given system would be whether the system's topology is supported by the given approach.

3.1.4. Use of bandwidth. As outlined in the Introduction, partial or low bandwidth network connectivity is often present in highly mobile systems. In such cases, there is a critical need for efficient access mechanisms over networks with variable qualities of service. In our taxonomy, use of bandwidth is divided into two dimensions: *intelligence* and *efficiency*. Intelligence indicates whether a given approach uses an adaptive algorithm to optimize the use of bandwidth, while efficiency indicates whether the approach minimizes the use of bandwidth in cases of low bandwidth connection.

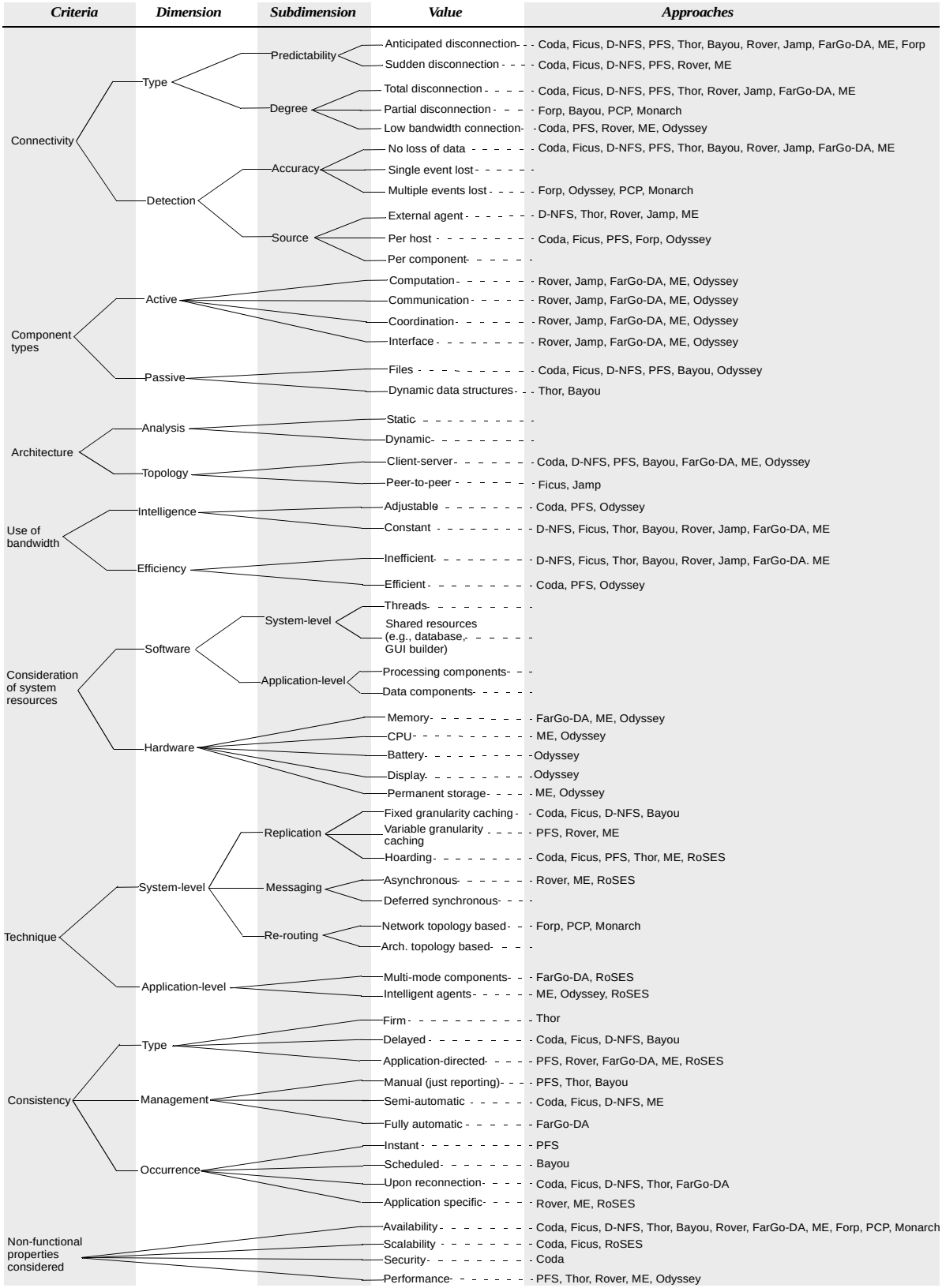

Figure 1. Classification Framework for Disconnected Operation Techniques.

3.1.5. Consideration of system resources. Dependability of a destabilized system is influenced by many factors. When selecting an approach that supports disconnected operation, it is important to know whether the approach considers the effects of the proposed changes on the system resources, and whether the available system resources on a set of affected hosts impose any restrictions on the proposed changes. If component migration is proposed by a given approach, it is important to assess whether the target device provides *hardware* resources (e.g., memory, CPU, display size) that the migrant component requires for normal operation. It is also important to assess the effects of *software* resources available on the target host (e.g., threads, existing processing components and their loads) on the migrant component. In our taxonomy, software resources are classified into *system-level* resources and *application-level* resources.

3.1.6. Technique. As outlined in the Introduction, there are a number of commonly used techniques for increasing system dependability during disconnection. We have classified these techniques into *system-level* and *application-level*.

System-level techniques are provided either at kernel- or middleware-level, and are further divided into *replication*, *messaging*, and *rerouting*. Replication subsumes techniques such as caching and hoarding, while messaging uses either asynchronous or deferred synchronous communication to delay remote interactions during disconnection. Synchronous messaging is not a feasible technique for supporting disconnected operation, since involved components would block for unpredictable periods of time.

Re-routing is a technique used to discover alternate paths of communication between mobile hosts. This subdimension can have two values: *network topology*, or *architecture topology* based rerouting. Network topology uses the information about the physical location of a given host and the network coverage of a given area. On the other hand, architecture topology uses additional information about the allowed communication paths among software components on each host to determine possible rerouting strategies. Both of these techniques can only support disconnected operation in cases of partial disconnection.

In the contrast to the above system-level techniques, several approaches [16,24] have proposed the use of application-level adaptation to increase system dependability during disconnection. For example, multi-modal components are designed with the a-priori knowledge that they may be executing in a disconnected mode. These components thus encapsulate two modes of operation: disconnected and connected. It is the responsibility of the application developer to design and implement a component's functionality such that it can be used during disconnection. The disconnected mode usually involves a subset of connected mode functionality, as well as methods for automated runtime conflict resolution.

Intelligent agents are special-purpose components whose role is to perform a set of activities which would translate a running application from a connected mode to a disconnected mode and vice versa.

3.1.7. Consistency. Several of the techniques described in Section 2 perform data, code, or system state replication to handle disconnected operation. Replication may require that changes made to different copies of a given component be synchronized upon reconnection. We have identified three dimensions of consistency: *type*, *management*, and *occurrence*.

Type denotes the extent to which the states of different replicas may diverge before they are synchronized. In *firm* consistency, the states of all replicas are always the same. This is achieved by either disallowing the updates to different replicas or by performing simultaneous, blocking updates to all replicas. In *delayed* consistency, replicas can be in different states, and consistency management (i.e., replica reconciliation) is performed upon reconnection. Some approaches also allow *application-directed* type of consistency, i.e., specification of (possibly different) consistency types for each application-specific operation.

Consistency management denotes the manner in which the reconciliation is performed. Some approaches *just report* inconsistencies, which are then resolved by the application user. In *semi-automatic* management, some conflicts are resolved automatically, while others are reported to the user for (manual) resolution. Finally, in *fully automatic* management all conflicts are resolved automatically, without the user's involvement.

Occurrence denotes the time at which the reconciliation is performed. In cases of *instant* reconciliation, each update will result in an immediate attempt to reconcile all replicas. In cases of *scheduled* updates, updates are planned and performed according to some, usually component-specific algorithm. Most of the existing approaches perform reconciliation *upon reconnection*. Finally, some approaches provide *application-directed* scheduling of reconciliation.

3.1.8. Non-functional properties. Existing disconnected operation approaches have considered different non-functional properties in the interest of increasing system dependability. Most commonly considered non-functional properties are availability, performance, security, and scalability. There are other relevant non-functional properties such as safety, reliability, utility, and so on. However, no existing disconnected operation approaches have focused on these properties.

3.2. Assessment of existing approaches

We have classified a number of representative approaches using our framework. The results of this classification are shown in the right-most column in Figure 1. In the remainder of this section we discuss these results.

Most of the existing approaches focus on anticipated disconnection, and on maximizing the system's availability during disconnection. Coda, PFS, ME, and Rover support both anticipated and sudden disconnection and provide support for low-bandwidth connection. With the exception of Bayou, partial disconnection is supported only by ad-hoc networking approaches (recall Section 2.3).

Coda, PFS, and Odyssey make intelligent and efficient use of the network bandwidth. However, none of the remaining approaches adjust their operation for a low-bandwidth connection. Instead, they assume either fully connected or disconnected mode of operation.

With the exception of Fargo-DA (for memory), only ME and Odyssey take into consideration system resources (CPU, disk space, battery, and so on). These approaches recognize that a given mobile host will not have unlimited resources to fully support techniques such as hoarding, and that certain trade-offs have to be made (e.g., providing continuous availability of only the most frequently used com-

ponents).

Fargo-DA, ME, and Odyssey use application-level disconnected operation techniques, while the remaining approaches leverage different combinations of system-level techniques. The most commonly used system-level technique is (some form of) replication, while the approaches that employ delayed communication via messaging only use asynchronous messaging.

Finally, none of the studied approaches perform any kind of analysis of the effects of the changes on the running system. They also fail to take into consideration other software resources (e.g., number of threads), or perform architecture-based re-routing (recall Section 3.1).

4. Conclusion

In this paper we have presented an attempt at classifying the existing disconnected operation approaches. A general understanding of these approaches and the techniques they employ is needed to effectively support system dependability in the face of disconnection. The existing approaches attack the problem of disconnected operation from four general perspectives: data housed in (1) static files and (2) dynamic data structures, and functionality implemented in both (3) inactive and (4) active software components. Typically, an approach will focus on a specific subset of these four categories (e.g., support for off-line access to static files only). Our classification framework is a step in the direction of understanding the (in)compatibilities among the existing techniques and suggesting the best possible approach or combination of approaches (e.g., coupling a passive file-based approach and an active component-based approach) for the problem at hand.

This work is preliminary and much remains to be done. A natural next step is to gain further experience by evaluating additional known disconnected operation approaches using the framework outlined in this paper. Such an evaluation will, in turn, be used to fine-tune the framework itself. In addition, we plan to study the compatibilities of the different criteria, dimensions, subdimensions, and values, which would help with identifying techniques that can be used in concert. Finally, further study of existing disconnected operation techniques will highlight additional areas that are currently not supported, which would motivate and help to streamline our future work.

5. References

[1] A. Carzaniga et. al. A Characterization Framework for Software Deployment Technologies. *Technical Report*, Dept. of Computer Science, University of Colorado, 1998.

[2] S. Chang and D. Curtis. An Approach to Disconnected Operation in an Object-Oriented Database. *3rd International Conference on Mobile Data Management*, January 2002, Singapore.

[3] M. Dahlin, B. Chandra, L. Gao, A. Khoja, A. Nayate, A. Razzaq, A. Sewani. Using Mobile Extensions to Support Disconnected Services. *University of Texas Department of Computer Sciences Tech Report TR-2000-20*, June 2000.

[4] D. Dwyer and V. Bharghavan. A Mobility-Aware File System for Partially Connected Operation. In *ACM Operating Systems Review*, Vol. 31, No. 1, Jan. 1997, pp. 24-30.

[5] Dynamic Software Architectures Resources. http://www.ics.uci.edu/~peymano/dynamic-arch/

[6] M. E. Fiuczynski and D. Grove. A Programming Methodology for Disconnected Operation. *Technical Report*, University of Washington, March 1994.

[7] K. Froese and R. Bunt. Scheduling Write Backs for Weakly Connected Mobile Clients. In *Proc. of the 10th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, Palma de Mallorca, Sept. 1998.

[8] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Software Engineering*, May 1998.

[9] J. S. Heidemann et al., Primarily Disconnected Operation: Experiences with Ficus. *Second Workshop on Management of Replicated Data*. IEEE, November 1992.

[10] D. B. Johnson and D. A. Maltz. Protocols for adaptive wireless and mobile networking. *IEEE Personal Communications*, 3(1), February 1996.

[11] A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, M. F. Kaashoek, Rover: a toolkit for mobile information access, *Proceedings of the fifteenth ACM symposium on Operating systems principles*, December 1995, Colorado.

[12] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, vol. 10, no. 1, February 1992.

[13] G. H. Kuenning and G. J. Popek. Automated Hoarding for Mobile Computers. Proceedings of the *16th ACM Symposium on Operating Systems Principles, (SOSP-16)* St. Malo, France, October 5-8, 1997.

[14] M. Madi, P. Graham, and K. Barker. Mobile Computing: Predictive Connection Management With User Input. *Technical Report*. Dept. of Computer Science, Univ. of Manitoba, 1997.

[15] M. Mikic-Rakic and N. Medvidovic. Architecture-Level Support for Software Component Deployment in Resource Constrained Environments. *First International IFIP/ACM Working Conference on Component Deployment*. Berlin, June 2002.

[16] W. Nace and P. Koopman. A Product Family Approach to Graceful Degradation. In *Proceedings of International Workshop on Distributed and Parallel Embedded Systems*, Germany, October 2000.

[17] B. Noble, et. al. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, St. Malo, France, October 1997.

[18] D.E. Perry, and A.L. Wolf. Foundations for the Study of Software Architectures. *Software Engineering Notes*, Oct. 1992.

[19] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. *Prentice-Hall*, 1996.

[20] C. Shelton, et al. A Framework for Scalable Analysis and Design of System-Wide Graceful Degradation in Distributed Embedded Systems. *WORDS 2003*, January 2003.

[21] W. Su and M. Gerla. IpV6 Flow Handoff in Ad-Hoc Wireless Networks Using Mobility Prediction. *Proceedings of IEEE Global Communications Conference*, pp 271-275, Rio de Janeiro, Brazil, December 1999.

[22] D. B. Terry, K. Petersen, M. J. Spreitzer, and M. M. Theimer.The Case for Non-transparent Replication: Examples from Bayou. *IEEE Data Engineering*, December 1998.

[23] M. T. Valente, R. Bigonha, M. Bigonha and A. Loureiro. Disconnected Operation in a Mobile Computation System. *Workshop on Software Engineering and Mobility*, Toronto, Canada, May 2001.

[24] Y. Weinsberg, I. Ben-Shaul. A Programming Model and System Support for Disconnected-Aware Applications on Resource-Constrained Devices. *International Conference on Software Engineering 2002*, Orlando, Florida, May 2002.

[25] Y. Zhang, V. Paxon, and S. Shenkar. The Stationarity of Internet Path Properties: Routing, Loss, and Throughput. *Technical Report*, AT&T Center for Internet Research at ICSI, May 2000.

An Architecture for Configurable Dependability of Application Services

Matthias Tichy[1] and Holger Giese[1]
Software Engineering Group
Department of computer science
University of Paderborn
Warburger Str. 100, 33098 Paderborn, Germany

E-mail: [mtt|hg]@uni-paderborn.de

Abstract

Many human activities today depend critically on systems where substantial functionality has been realized using complex software. Therefore, appropriate means to achieve a sufficient degree for dependability are required, which use the available information about the software components and the system architecture. For the special case of service-based architectures – an architecture proposed to cope with the complexity and dynamics of today's systems – we identify in this paper a set of architectural principles which can be used to improve dependability. A service-based architecture which extends Jini and employs the identified architectural principles is further proposed and realized. The dependable operation of the infrastructure services of the architecture further enables to systematically control and configure some dependability attributes of application services.

1. Introduction

The dependability of today's complex systems often relies on the employed computers and their software components. Availability, reliability, safety and security (cf. [10]) are the attributes of dependability that are used to describe the required system characteristics. These four attributes in practice often depend on each other. Availability and reliability can in principle be systematically studied at the level of components and their composition in form of specific architectures. The ever increasing system complexity and the increasingly ubiquitous character of computing, however, render such an analysis a difficult task.

For complex systems the required prediction models for availability and reliability become quite complex when maintenance activities and component modifications are also taken into account. When further considering dynamic systems where no static *a priori* known system configuration exists, the analysis and prediction of the reliability or availability is thus usually not possible. We therefore propose to build dynamic and dependable complex systems not by relying on design-based quantitative analysis of its static architecture. Instead the observed obstacles should be addressed by a dynamic reconfiguration of the architecture to prevent system reliability and availability to decrease below the required level. Such a software tries to compensate failures (originated from defects of its hardware and software components) by means of adaption. In accordance with [12], which defines *self-adaptive software* as software that modifies its own behavior in response to changes in its operating environment, we thus classify it as *self-healing software*.

We further restrict our considerations for dependability on reliability and availability and study how the dynamic management of redundant component instances with identical implementation can contribute to improvements for these two dependability attributes. The questionable impact of using multiple diverse implementations (cf. [7]) is not considered. The application services are further treated as black-boxes with given dependability characteristics. We make the strong simplification that hardware and software component failures simply result in the inability of the affected services to fulfill the regular behavior. Thus, failures can be detected externally by monitoring the services.

A number of architectural principles which permit to enhance the dependability of service-based architectures are presented in Section 2 and their benefits are motivated referring to the Jini architecture. Then, we propose in Section 3 to enhance the Jini architecture by a number of infrastructure services that systematically employ the identified principles. We then discuss the benefits achieved for application specific services concerning availability and reliabil-

[1]This work was developed in the course of the Special Research Initiative 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

ity in Section 4 and demonstrate the systematic application of the identified architectural principles within the enhanced architecture. For a special class of services the possible design alternatives are studied by means of the detailed design of two infrastructure services in Section 5. Related work is discussed in Section 6 and we close the paper with a final conclusion and some comments on future work.

2. Architectural Principles for Dependability

Software systems typically consist of different parts. Since dependencies between these parts exist, problems occur if a part fails. Service-based architectures handle the increasing complexity of today's systems by means of online lookup and binding of services. The integral part of a service-based architecture is a *service registry*. The use of such a service registry is a key factor for availability, since service instance connections are not hard-wired. Instead they can spontaneously connect to recover from failures. One example of a self-healing service-based architecture is the Jini architecture [1, 13]. It has been designed (cf. [19]) to support the development of dependable distributed systems. One of its features is a *lookup service* that remains operational even when single nodes in the network have crashed.

The *leasing* principle extends the allocation of resources with time [18]. The lease represents a period of time during which the resource is offered. Therefore this lease needs to be extended (renewed) if the resource remains to be offered after the timeout of the lease. If the owner of the resource fails to renew the lease, a client can assume that the resource is no longer available. Leasing is the principle which provides the self-healing behavior of the Jini lookup service. Every service registration on the lookup service is accompanied by a lease. If this lease expires, the lookup service removes the accompanied service registration from its lookup tables. Thus no service gets this apparently failed service instance in response to a search request. If this service is restarted or the communication system is repaired, the service can re-register on the lookup service.

A *proxy* provides a surrogate or placeholder for another object [4]. In distributed systems a proxy typically acts as a local placeholder for a remote object encapsulating the forwarding of requests via network communication (e.g. as the stub in Java Remote Method Invocation (RMI) [15]). In the Jini architecture the proxy pattern is an integral part of every service. A service is divided into a proxy and an optional backend. The proxy instance is registered in the lookup service. If a service is to be used by a client, the proxy instance is downloaded as mobile code to the client and executed there.

Redundancy of service instances is a key factor to achieve a required degree of availability. A non redundant service is a single-point-of-failure. Thus in case of a failure of this service or a communication subsystem failure, which results in a network partition, all dependent clients of that service cease to work. In the Jini architecture more than one lookup service can be used. Thus a failed lookup service does not compromise the dependability of the complete system.

This leads us to the concept of a *smart proxy* [9, 11]. A smart proxy is not restricted to forwarding but can be used much more flexible. Thus in the context of availability the proxy may communicate with multiple backends at once to recover from or mask failures. Hence a smart proxy can be used to encapsulate and hide the complexity of self-adapting code and therefore the use of complex concepts becomes transparent to the user of the service. For example the service registration in the Jini architecture is sent to all available lookup services by the proxy at once using multicast messages.

Analogue to the redundancy of services a key point for dependability is the availability of data in a distributed system. This can be achieved by the use of *replication*. Replicating is the process of maintaining multiple copies of the same entity at different locations. In the Jini architecture the service registrations are replicated in multiple lookup services.

The maintenance of these distributed copies depends on the required consistency for the entity. There exist different consistency models (for an overview see [16]). A consistency model provides stronger or weaker consistency in the sense that it affects the values, a read-operation on a data item returns. There is a trade-off between consistency and availability and no general solution can be given. The weaker the consistency model the easier availability can be achieved. The possibility to use different consistency models for different data aids in the development of a self-healing architecture as we will show in the next section.

3. Architecture

In this section we will show the application of the introduced architectural principles. We give a short introduction of the proposed architecture and the requirements of the different infrastructure services. More details and the description of the implementation can be found in [17].

The Jini architecture supports ubiquitous computing in ad-hoc networks and provides a dependable infrastructure for service lookup and operation. However, the basic infrastructure only avoids to provide any element that can compromise the dependability of application components. But to achieve the required dependability for any specific service or application remains to be realized by the application developer. Our proposed architecture provides availability for application services.

A key to the improved availability of the infrastructure services is the idea to have redundant instances of every

service type running concurrently in the system to prevent a single-point-of-failure as proposed in the last section. Keep this idea in mind while we describe the specific infrastructure services next.

Four different services are building the overall architecture on top of Jini. Ideally on every computation node of the system one instance of each infrastructure service is executed and will be restarted automatically during each reboot (see Figure 1).

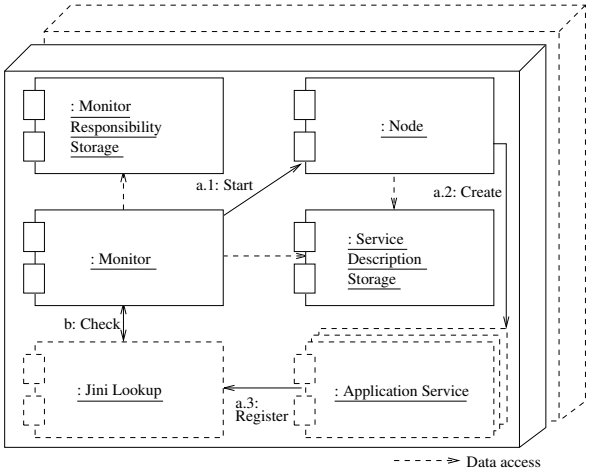

Figure 1. Architecture

Basically on every node of the distributed system an instance of the *node* service is running. Using this node service, new application service instances can be created (and old ones stopped).

A *service description storage* contains service descriptions (like name of the service, package-path, used and provided interfaces, deployment constraints, etc.) for all services which have to be executed. Each instance of the service description storage contains one replica of the service descriptions. A strong consistency model for these data is required since a weaker consistency model would result in a possible loss of service descriptions in case of failures. This in turn would cause the unavailability of the affected application services.

Monitors supervise that the services contained in the service description storage are indeed executed in the system. The availability of the services will be checked periodically. The detection speed of service failures can be configured by changing this period. If more than one instance of one specific application service has to be executed in the system, each instance is monitored by a different monitor. To control which monitor is supervising which service, every monitor needs to acquire a responsibility for a service (i.e. to assure a new instance is started, if a service instance is not available).

These responsibilities are stored in a *monitor responsibility storage*. Responsibilities are accompanied by a configurable lease, which is used to detect failed monitors (i.e. the lease times out). Each instance of the monitor responsibility storage contains a copy of these monitor responsibilities. Inconsistencies between these copies only result in changed responsible monitors and potentially additional started service instances. Therefore we trade reliability for overhead and weaken the consistency requirements for these copies. Additionally after a repaired network partition failure merging the responsibilities in the former partitions must be possible. The monitors whose behavior depends on these responsibilities must be able to cope with a weaker consistency model.

4. Evaluation

After this short introduction to the different infrastructure services we show how the architecture achieves availability for application services in case of node failures and network partition failures. Afterwards we show how to achieve a required degree of reliability for different categories of application services based on the availability provided by the architecture.

4.1. Availability

In case of a node failure different scenarios, w.r.t. failures of a responsible monitor and monitored services, are possible. The case that neither a responsible monitor nor a monitor service is affected by the node failure is trivial. If a node is affected by the failure which does host only application services, the monitors responsible for these application services will detect the services' failures because they do not renew their leases with the lookup service. The monitors will choose new nodes for the application services and start new instances there. Figure 2 shows this scenario. Note the displayed monitor and lease periods, which influence the achievable degree of availability.

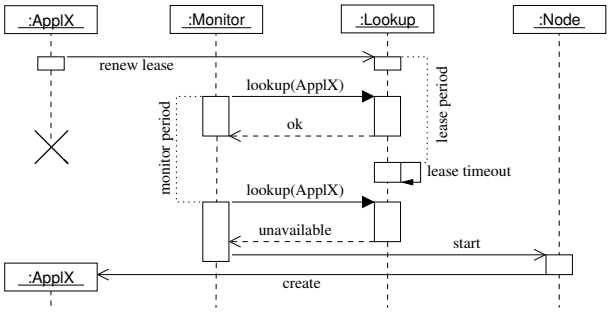

Figure 2. Node Failure

In the case of a failed responsible monitor and failed monitored services the responsibility lease of this monitor times out and another monitor steps in. This monitor replaces the failed monitor, and starts supervising the currently unmonitored services which includes starting new in-

stances when needed. Figure 3 shows the leases and the events in a condensed form.

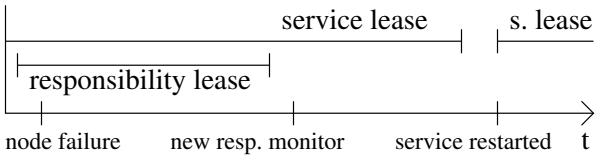

Figure 3. Monitor Failure

During a network partition failure, communication is only possible inside the different partitions and no communication can cross the borders between the different partitions.

A monitor, which has been responsible for services in the complete network, is in one part of the system during the partition. In this part the monitor recognizes the absence of some monitored services and restarts them. In the other parts the monitor's responsibility times out, other monitors step in, and create all needed service instances. Thus in each partition a responsible monitor and all service instances are available after a certain amount of time (cf. Figure 3).

After reuniting the different partitions, the responsibility storages are merged to determine a new unified responsible monitor. This new monitor takes over the service instances started by the other responsible monitors in the other partitions. Additionally it can consolidate the number of service instances in the reunited network. The monitors formerly responsible in the other partitions stop monitoring their started service instances.

As seen the availability of application services (and the Mean Time to Repair (MTTR)) can be configured by changing the lease given by the Jini lookup service, the monitoring period and the responsibility lease. Therefore the proposed architecture can be customized for a high degree of availability.

4.2. Reliability

The presented architecture ensures the availability of the application services in the system. Nevertheless for each application service there must exist a concept to achieve the required degree of reliability based on the availability provided by the architecture. Different categories of services require different approaches to achieve reliability.

According to [3] services can be categorized in terms of modeling an *entity* or containing an activity (*session*). A session service may either be stateless or stateful. The state of a session is the history of relevant actions executed throughout this session. If an action is independent of the previous actions, the service is stateless, if not it is stateful. An entity service always has a state (its local data).

For stateless session services it is irrelevant which service instance is used for a given action, since the actions are independent of each other. Thus the availability provided by the architecture is sufficient. If a service instance fails, another instance can be used.

If a used stateful session service instance fails, just using another service instance from thereon does not work. Essentially the last state of the failed service instance must be recreated on the newly used service instance. Thus the history (relevant actions) until the point of failure needs to be replayed.

To achieve reliability for entity services it is necessary to replicate copies of the entity over a number of nodes to be able to mask failures. Additionally the consistency of these entities according to a suitable consistency model must be assured. This replication is highly application-specific and thus no general solution can be given. For example in our architecture we have data with two very different requirements (service descriptions and monitor responsibilities) which can be provided by appropriate consistency models (see section 5).

The implementation of the above mentioned concepts leads to a reliable system, but unfortunately the maintainability of the resulting system deteriorates. We propose the usage of the smart proxy pattern to encapsulate the complexities of achieving reliability. Since the proxy does not fail independently of the using application, the client does not need to handle a failed proxy. Thus a service client only needs to know the interface to the service and nothing about the different means of accomplishing reliability. It uses the smart proxy via an interface and all additional processing for reliability is done internally in the smart proxy (see Figure 4).

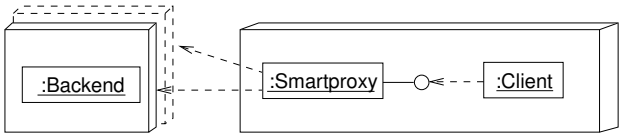

Figure 4. Smart proxy

5. Design of Infrastructure Services

In the following we further describe in detail the design of two infrastructure services. These services serve as examples how to achieve the required degree of reliability for entity services according to the last section.

Service Description Storage This storage contains the descriptions of the services which must be available in the system. These descriptions are replicated in the system on a number of service backends. A strong consistency model is required for this replication. Write operations are only executed by an administrator whereas read operations are regularly executed by the infrastructure services.

Since changes in the service descriptions happen rarely, the number of read operations on these descriptions surpasses the number of write operations. For a certain degree of the system's reliability, it is necessary that the read operations of the infrastructure services succeed with a very high probability in case of failures whereas the write operations are unimportant. To exploit this bias for read operations we have chosen to implement the *weighted voting* approach [5] which provides sequential consistency.

This approach offers the possibility to configure the reliability, based on the assumed distribution of read and write operations. Each node has a number of votes to weight its data. Additionally this number of votes can be changed to match the reliability of that node. The weighted voting approach uses a voting where the needed read (n_r) and write quorums (n_w) can be adjusted as long as read-write quorums ($n_w + n_r > n$) and write-write quorums ($2n_w > n$) overlap to prevent inconsistencies (n : number of votes). For our scenario we choose a high n_w and a low n_r to achieve a high probability for a successful read operation.

Multiple node failures can be masked as long as the required number of votes is available to reach the required quorum. In case of a network partition read operations are possible in every partition containing more than n_r votes. Write operations are only possible in the rare case that one partition contains more than n_w votes.

The weighted voting approach is implemented in a smart proxy. Thus a client does not need to know about the specific implementation; it just calls read and write operations on the proxy and all replication and consistency management is done internally.

Monitor Responsibility Storage Storing the monitor responsibilities is a problem similar to storing the service descriptions. In contrast write and read operations are equally important. In case of failures it is necessary that another monitor can take over the responsibility of a broken monitor and needs to write that information back into the responsibility storage.

Therefore we can weaken the consistency requirements for the responsibility storage to be able to read and write to it anytime especially in the failure case. An appropriate weaker consistency model is *eventual consistency* [16]. Eventual consistency demands that in absence of write operations the storages eventually stabilize in a globally consistent state after a certain amount of time.

Our approach to achieve eventual consistency is based on multicast messages and a decentral majority voting on every responsibility storage in the network. Because of the multicast messages, every message is received by every storage. Thus, in case of a read operation, all available storages receive the read request and respond by returning their local data as a multicast message. Therefore every storage and the requester get the responsibilities stored in every storage. Since the number of storages is unknown in case of failures a timeout is used to finish waiting for responses. After that, all storages and the requester do a decentral majority voting on the received data. In case of parity each participant chooses the data with the highest hashcode to achieve a consistent result. A write operation simply sends a write multicast message which is processed by all storages, which receive the message.

Before a globally consistent state is reached there may exist local inconsistencies. For example, during a network partition failure the local data in the storages in the different partitions diverge because updates are only visible within one partition. After the failure is repaired the conflicts between all partitions are resolved by the next read operation. After the decentral majority voting the data of only one partition holds, the others are discarded. Therefore only one monitor is responsible for a specific service description. All other, former responsible monitors notice their responsibility loss on their next responsibility check.

From a user point of view this complex dealing with multicast messages and the voting is completely encapsulated within a smart proxy.

6. Related Work

In the Jini-context the problem of availability is somewhat supported by use of the RMI-Daemon [15]. This daemon supports the on demand creation of remote objects. Additionally if the node fails, after a reboot and a restart of the daemon all remote objects are recreated. Nevertheless this daemon only restarts the remote objects on the same node. Therefore this is not a solution if a node fails permanently or if the remote objects should be available during the repair of the node.

The RIO-Project [14] uses a somewhat similar approach compared to ours. One single monitor is loaded with the service descriptions and ensures the availability of the contained services in the system. The fact that the service descriptions are only available inside of the monitor makes the monitor a single-point-of-failure in the system. If the monitor process fails, the service descriptions are lost since they are not replicated. No other monitor can use those service descriptions and replace the existing monitor without manual intervention. Thus the reliability of the RIO approach depends heavily on the reliability of one monitor instance. Additionally during a network partition failure the approach does not work since the monitor instance cannot be in more than one partition of the network. Hence this approach is not applicable for dependable systems.

The Master-Slave pattern [2] can be applied when services are replicated and a result must be selected which is returned to the client. This is similar to our smart proxy approach. The slaves are the different service instances

whereas the smart proxy is the master in our approach. The Master-Slave pattern is aimed at stateless session services whereas our approach can also be used for the consistent management of entity services.

The Willow-Architecture by Knight et al. [8] provides survivability for critical distributed systems. As a response to faults reconfiguration is used to ensure the survivability of the system. The response to failures is based on a monitor/analyze/respond-control loop which is similar to our behavior of the monitor.

Gustavsson and Andler describe in [6] a distributed real-time database which uses eventual consistency. Similar to our approach they use this consistency model to improve the availability and efficiency and to avoid blocking for unpredictable periods of time.

7. Conclusions and Future Work

For the proposed architecture implemented on top of Jini, we have shown that the infrastructure services itself build a dependable system. This includes that in contrast to related proposals no single-point-of-failure for node crashes or network partition failures is possible. The number of parallel running service instances and lease times for registry and monitoring can be chosen. Thus for any architecture conform application specific service availability can be configured. For different kinds of application services we presented appropriate concepts to also realize a higher reliability. The required additional efforts are systematically hidden to the service clients using the smart proxy concept of Jini. The smart proxy concept itself can be used in every service-based architecture. But the reliability provided by the presented architecture highly depends on the robustness of the underlying service-based architecture. To adapt it to other architectures than Jini, the leasing concept of Jini needs to be reimplemented in the different application services to offer a Jini-like robustness.

In addition to the implementation of the presented dependable architecture and its run-time system, tool support by means of UML component and deployment diagrams has been realized [17]. This includes code generation for the services, generation of XML deployment descriptions, and the visualization of the current configuration by means of UML deployment diagrams. We further plan to evaluate the architecture in the context of complex embedded and real-time systems. For small examples a formal analysis using Markov models will be performed. Additionally, we will look how run-time measurements of node, network and component dependability characteristics can be employed to adjust the system parameters such as monitor supervision periods accordingly. In a next step, we want to employ classical approaches for learning and adaption to automatically use this feedback to improve the system's dependability.

Acknowledgments

The authors wish to thank Sven Burmester, Matthias Gehrke, and Matthias Meyer for comments on earlier versions of the position paper.

References

[1] K. Arnold, B. Osullivan, R. W. Scheifler, J. Waldo, A. Wollrath, and B. O'Sullivan. *The Jini(TM) Specification*. The Jini(TM) Technology Series. Addison-Wesley, June 1999.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern Oriented Software Architecture*. John Wiley and Sons, Inc., 1996.

[3] L. G. DeMichiel, L. Ü. Yalcinalp, and S. Krishnan. *Enterprise JavaBeans^{TM} Specification*. Sun Microsystems, August 2001. Version 2.0.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[5] D. K. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the seventh symposium on Operating systems principles*, volume 7 of *ACM Symposium on Operating Systems Principles*, pages 150–162. ACM press, 1979.

[6] S. Gustavsson and S. F. Andler. Self-stabilization and eventual consistency in replicated real-time databases. In *Proceedings of the first workshop on Self-healing systems*, pages 105–107. ACM Press, 2002.

[7] J. Knight and N. Leveson. A reply to the criticisms of the Knight & Leveson experiment. *ACM SIGSOFT Software Engineering Notes*, 15(1):25–35, January 1990.

[8] J. C. Knight, D. Heimbigner, A. Wolf, A. Carzaniga, J. Hill, P. Devanbu, and M. Gertz. The Willow Architecture: Comprehensive Survivability for Large-Scale Distributed Applications. In *The International Conference on Dependable Systems and Networks (DSN-2002)*, Washington DC, June 2002.

[9] R. Koster and T. Kramp. Structuring QoS-Supporting Services with Smart Proxies. In *Proceedings of Middleware'00*. Springer Verlag, April 2000.

[10] J. C. Laprie, editor. *Dependability: basic concepts and terminology in English, French, German, Italian and Japanese*, volume 5 of *Dependable computing and fault tolerant systems*. Springer Verlag, Wien, 1992.

[11] P. Ledru. Smart proxies for jini services. *ACM SIGPLAN Notices*, 37(4):57–61, Apr. 2002.

[12] P. Oreizy et al. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, May/June 2002.

[13] Sun Microsystems. *Jini Specification*, October 2000. Revision 1.1.

[14] Sun Microsystems. *RIO - Architecture Overview*, 2001. 2001/03/15.

[15] Sun Microsystems. *Java^{TM} Remote Method Invocation Specification*, 2002. Revision 1.8, JDK 1.4.

[16] A. Tanenbaum and M. van Steen. *Distributed Systems, Principles and Paradigms*. Prentice Hall, 2002.

[17] M. Tichy. Durchgängige Unterstützung für Entwurf, Implementierung und Betrieb von Komponenten in offenen Softwarearchitekturen mittels UML. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, July 2002.

[18] J. Waldo. The jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, 1999.

[19] J. Waldo, G. Wyant, A. Wollrath, and S. Kendal. A Note on Distributed Computing. Technical report, Sun Microsystems Laboratories, November 1994. TR-94-29.

An Approach to Manage Reconfiguration in Fault–Tolerant Distributed Systems

Stefano Porcarelli[1], Marco Castaldi[2], Felicita Di Giandomenico[1], Andrea Bondavalli[3], Paola Inverardi[2]

[1]Italian National Research Council, ISTI Dept., via Moruzzi 1, I-56134, Italy
stefano.porcarelli@guest.cnuce.cnr.it, digiandomenico@iei.pi.cnr.it
[2]University of L'Aquila, Dip. Informatica, via Vetoio 1, I-67100, Italy
{castaldi, inverard}@di.univaq.it
[3]University of Florence, Dip. Sistemi e Informatica, via Lombroso 67/A, I-50134, Italy
a.bondavalli@dsi.unifi.it

Abstract

This paper deals with dynamic resource management for real–time dependability–critical distributed systems. Requirements for such kind of systems span many domains such as time, survivability, and scalability and point out formidable challenges in terms of their fulfillment. An architecture is proposed, based on the agent distributed infrastructure Lira, and enriched with statistical models for decision-making capabilities. The aim of the proposed architecture is to provide adaptive system reconfiguration, resorting to a hierarchy of resource managers to cope with fault tolerance and scalability issues.

1 Introduction

Dependability has become a crucial requirement of current computer and information systems, and it is foreseeable that its importance will increase in the future at a fast pace. We are witnessing the construction of complex distributed systems, which are the result of the integration of a large number of low–cost, relatively unstable COTS (Commercial Off–The–Shelf) components, as well as previously isolated legacy systems. The resulting systems are being used to provide services, which have become critical in our everyday life. Since COTS and legacy components are not designed to achieve high dependability by themselves, their behavior with respect to faults can be the most disparate. Thus, it is paramount for these kinds of system to be able to survive failures of individual components, as well as attacks and intrusions, although with degraded functionalities. This paper contributes to fault tolerance in such framework, by focusing on fault handling strategies [1], particularly on system's reconfiguration.

The aim of system reconfiguration is to provide control capabilities over unanticipated events in order to maintain the system in a certain desirable state. An effective reconfiguration policy highly depends on an accurate diagnosis of the nature of the unanticipated event, namely if a hard, physical fault is affecting the system, or environmental adverse conditions are causing a soft fault which will naturally disappear in some time. It is out of the scope of this paper to address diagnosis issues; we concentrate on reconfiguration only, which is triggered on the basis of information on the healthy status of system components, assumed accessible to our management architecture.

The rest of the paper is organized as follows. Section 2 describes our architectural approach towards a fault tolerant resource management policy. Section 3 details the overall system management architecture and Section 4 introduces a simple case study to illustrate our approach. Section 5 briefly concludes the paper and points out some future research directions.

2 Our Approach for Adaptive Resource Management

A basic characteristic of our resource management architecture is its ability to adapt to dynamic system conditions. Different degrees of adaptivity may be (theoretically) possible, ranging from picking up the reconfiguration policy from a look-up table where policies have been pre-stored on the basis of off-line analysis, to completely dynamic definition of the best reconfiguration to perform. An intermediate solution is that where a number of strategies are pre-planned, but the choice of which one is the most appropriate given certain run-time system conditions is decided dynamically. The degree of dynamic decision making has to be carefully harmonised with other possible system con-

straints. In fact, the design of a resource management infrastructure for distributed systems is influenced by several factors, among which the nature of the event that triggers a reconfiguration action, the size of the computational effort for decision making, and timeliness requirements.

Adaptive resource management is very demanding nowadays; the approach we propose is mainly characterized by two aspects. Firstly, we introduce a model-based activity, which provides on-line quantitative evaluation of the impact of different reconfiguration strategies and thus helps in the selection of the most appropriate one. Secondly, the decision making process is decomposed in a hierarchical fashion, each level differing in the visibility of, and the ability to act on, the portion of the system under its control. Resorting to a hierarchical approach brings benefits under several aspects, among which: i) favoring fault tolerance by distribution of control; ii) avoiding heavy computation and coordination activities whenever faults can be managed at local level; iii) facilitating the construction and on-line solution of analytical models; iv) favoring scalability.

In our framework, the Light–weight Infrastructure for Reconfiguring Applications (Lira) [2] is used to perform remote control and dynamic reconfigurations over single components or applications. Lira does not formally describe the procedure of decision making, it only assumes that a proper subsystem is in place, in charge of deciding when and in which way the system needs to be reconfigured. In this paper, we are enriching the Lira framework with a hierarchical Decision Maker (DM) in charge of online selection of the reconfiguration policy to apply. The DM exploits a model-based support to guide its decision. Lira monitors system and environment conditions passing the state of components and applications to the DM as input for taking decisions.

Upon receiving inputs requiring a reconfiguration, the DM activates the model-based evaluator to devise the most appropriate configuration and behavior to face the current situation [4]. For example, it permits to evaluate the dependability of a new architecture of the system achieved rearranging resources due to faults, or to carry out cost-benefit tradeoff choices. The output provided by the decision maker is a new system configuration; such output is then managed by Lira, to put into action the selected reconfiguration.

3 The Lira Management Infrastructure and the Decision Maker

Lira is inspired to the Network Management [9] in terms of reconfiguration model and basic architecture. The reconfiguration model of the Network Management is quite simple: a network device, such as a router or a switch, exports some reconfiguration variables and functions through an Agent, which is implemented by the device's producer. These variables and functions exported by the agent are defined in the MIB (Management Information Base) and can be modified using the *set* and *get* messages of SNMP (Simple Network Management Protocol) [9]. For software components, a reconfiguration is any allowed change in the component behavior, while an application reconfiguration is any change in the application's topology in terms of number and location of components [6][5][11].

In the next sections we will first provide an overview of the Lira infrastructure, and then integrate in it the Decision Maker.

3.1 The Lira Infrastructure

The Lira architecture specifies three elements: (i) the **Agent**, which acts on the managed components, implements the reconfiguration logic and communicates with other Agents, (ii) the **MIB** which contains the list of variables and functions exported by the agent, and (iii) the **Management Protocol**, which allows the communication among the agents.

A Lira Agent is a program that runs on its own thread of control and communicates with other Agents in an asynchronous way, using the Management Protocol. There are four kinds of agents organized in a hierarchical way [10] (see Figure 1) and specialized in different tasks: (i) **Component Agents** (CompAgents), associated to software components, (ii) **Host Agents**, associated to the host where the components are deployed, (iii) **Application Agents** (AppAgent), which control part of the system (different components and hosts), (iv) and the **Manager Agent** (Manager), on the top of the hierarchy, that controls the whole system. Accordingly to the hierarchical structure, an agent has manager capabilities on the portion of the system under its own control (helped in the decisions by the Decision Maker), while it is a simple actuator with respect to the higher level agents. In this paper we are discussing a logical architecture without addressing the deployment of the agents on the hosts. Of course, it is a delicate issue for fault tolerance, since it is necessary to have separate containment regions for the monitoring units and the monitored ones.

The *Component Agent* (CompAgent) directly controls and manages the component. Lira does not specify how the component is attached to the agent, but it only assumes that the agent is able to act on the component. The logical model of communication between CompAgent and component is through *shared memory*; in fact, the Component shares a part of its state with the Agent and explicitly allows the reconfiguration. To avoid synchronization problems, the component has to provide atomic access to the shared state. The CompAgent manages the component's life-cycle by exporting the functions *start, stop, suspend, resume* and *shutdown*. The function *shutdown* stops the component and kills the

agent. For monitoring purpose, the CompAgent exports a predefined read-only variable *STATUS*, which maintains the current state of the component (starting, started, stopping, stopped, suspending, suspended, resuming). Each agent is able to notify the value of a variable to its manager, addressed by the variable *NOTIFYTO* also defined in the MIB.

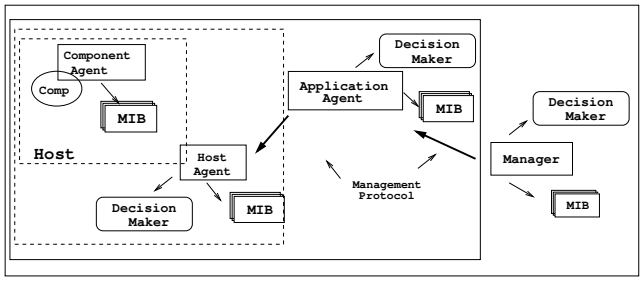

Figure 1. Lira general architecture

The *Host Agent* runs on the host where components and agents must be deployed. It dynamically installs and activates components and agents by exporting the functions *install, uninstall, activate, deactivate*. Moreover, this agent maintains the lists of both installed and activated components on the host: these lists are exported in the variables *ACTIVEAGENTS* and *INSTALLEDAGENTS*. Note that the Host Agent does not manage components, but it monitors and controls host's resources and parameters. All the variables and functions are specified in the MIB.

The *Application Agent* (AppAgent) is a higher level agent which controls a set of components through the associated CompAgents. These agents manage a subsystem as an atomic component, hiding the reconfiguration's complexity and increasing infrastructure scalability.

The *Manager* is the highest level agent which has the global knowledge to control the whole system. AppAgents communicate with the controlled components through the Management Protocol, so they are independent from the specific components and can be programmed by means of an interpreted language: the *Lira Reconfiguration Language*. By means of this language, the agent's developer defines a reconfiguration as a function: the function is invoked internally when the agent is acting in a manager role, or it is exported in the MIB and called by a manager when the agent acts in the actuator role.

The *Management Protocol* is inspired to SNMP, with some modifications and extensions. Each message is either a *request* or a *response*, as shown in the following table:

request	*response*
SET(*var_name, var_value*)	ACK(*msg_text*)
GET(*var_name*)	REPLY(*var_name, var_val*)
CALL(*func_name, par_list*)	RETURN(*ret_value*)

Requests are sent by higher level agents to lower level ones and responses are sent backwards. There is one additional message, which is sent from agents to communicate an alert at upper level (even if in the absence of any request):

NOTIFY(*variable_name, variable_value, agent_name*)

As in the Network Management, the *MIB* represents the agreement among agents that allows them to communicate in a consistent way. The MIB provides the list of variables and functions exported by the agent which can be remotely managed. A function is usually a reconfiguration process that the agent makes available for its manager. Note that also the predefined variables and functions that characterize the different agents (for example, the variable *STATUS* or the function *stop* in the CompAgent) are defined in the MIB. In [3] the MIB is presented in detail.

The Lira infrastructure described here was not created with dependability requirements, so we are investigating how to modify the infrastructure providing new features which guarantee continuous monitoring and distributed management of components and hosts, making Lira dependable itself.

3.2 Decision Maker

The decision maker takes decisions about system's reconfiguration. Decisions can be taken at any level of the agents hierarchy as proposed by Lira (four levels) and, consequently, the power of the reconfiguration is different.

The first, bottom level is that of a Component Agent. At this level, the Decision Maker can only autonomously decide to perform preventive maintenance on the controlled component. At the second level, that of the Host Agent, the DM can decide about installation and de-installation of such components. The third level concerns the Application Agent; at this level, the DM's reconfiguration capabilities span all software and hardware resources under its responsibility. At the highest level there is the Manager agent, which has a "global" vision of the system; therefore the DM at this level can perform an overall reconfiguration. After taking the decision on reconfiguration at a certain level, it is sent to the lower level agents which act as actuators on the controlled portion of the system.

For the sake of simplicity, the status of each monitored system unit may assume three values: *Up* indicating that the component is well working; *Degraded* indicating that the component is working in a degraded manner (e.g., in the case the component is hit by a transient fault which reduces its functionalities); and *Down* indicating that the component is definitely wrongly working (e.g., it is hit by a permanent fault). At each level of the decision making hierarchy, the DM perceives the behavior of each system unit visible at the one-step lower level in terms of up, degraded or down. To make an example, at the manager agent level, the DM

has knowledge of the behavior of each application running in the system, each one seen as single system unit; in turn, at application agent level, the DM has knowledge of the behavior of each host involved in that application, again each one seen as a single system unit, and so on.

According to the depicted hierarchical reconfiguration process, when an event triggering a reconfiguration action at a certain level occurs, the DM at that level attempts the reconfiguration, if possible. In case it cannot manage the reconfiguration, it notifies the upper level DM about both detected problem and its healthy status. In turn, the upper level DM receiving such request to trigger a reconfiguration, uses such heathy status information, together with those of the other system units under its control.

As introduced in Section 2, the way to make decisions may be different. If it is possible to assume stochastic independence among failure and repair processes of various components, the new reconfiguration scheme can be simply retrieved from a look-up table where pre–evaluated policies (e.g. by means of combinatorial models, like fault tree) have been stored. If some environmental parameters may change at the moment of the reconfiguration, combinatorial models must be solved each time. In case the failure of a component may affect other related components, space–state models are necessary. These are solved by the DM on the basis of the information collected from the subordinated agents. In this case, each unit component is modeled with a simple Petri net which describes its forecasted behavior given its initial state. It is in charge of the Decision Maker to solve such overall composed model as quickly as possible to take appropriate decisions online identifying the most rewarding reconfiguration action among a pool of pre–defined options.

Therefore, the status of any controlled component (provided by Lira) is used as *input* for the appropriate decision maker, that reasons, decides and gives back as *output* (to Lira) the *optimal reconfiguration action*. Decisions are taken resolving analytical models (not shown for brevity), essentially based on Deterministic and Stochastic Petri Nets (DSPN) [7], and solved by means of the tool DEEM [8].

Obviously, the correctness of the decisions depends both on the accuracy of the models and on its input parameters.

4 A Simple Example: Path Availability of a Communication Network

To better motivate our methodology a possible scenario is presented. A simple, but meaningful scenario, is the case of distributed computing where two peer–to–peer clients on the network are communicating. To prevent service's interruption, we need to provide an adequate level of paths redundancy among the clients involved in the communication. We suppose to have a network topology where six hosts are physically (wired) connected as shown in Figure 2. For management purpose, we consider the network divided in two subnetworks Net_1 and Net_2, which contain the hosts H_1, H_2 and H_3, H_4 respectively. The hosts H_5 and H_6, where the clients are deployed, are not included in the managed network.

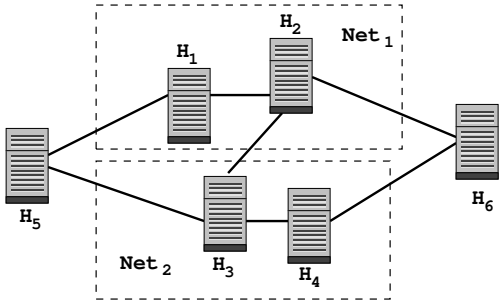

Figure 2. Hosts physical connection

A logical communication network composed by logical nodes connected through logical channels is installed on the managed hosts. The nodes N_1, N_2, N_3, N_4 connected through the channels *a, b, c, d, e, f, g* are deployed on the subnetwork Net_1, as shown in Figure 3. These channels provide different choices for establishing the communication among the clients, as listed in Table 1.

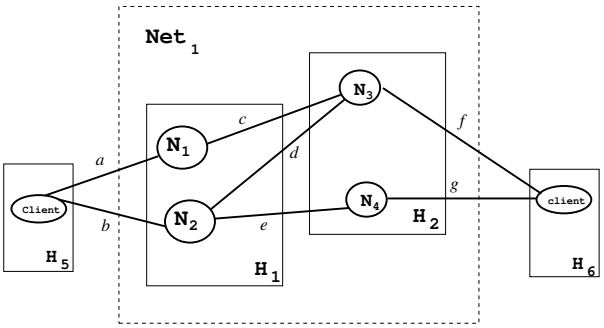

Figure 3. Logical infrastructure topology

Path	Route
1	a–N_1–c–N_3–f
2	a–N_1–c–N_3–d–N_2–e–N_4–g
3	b–N_2–e–N_4–g
4	b–N_2–d–N_3–f

Table 1. Available paths

These path options can be used to improve the overall availability of the logical network by providing redundancy. The goal of the management infrastructure is to keep *at least* two paths available between the clients involved in the

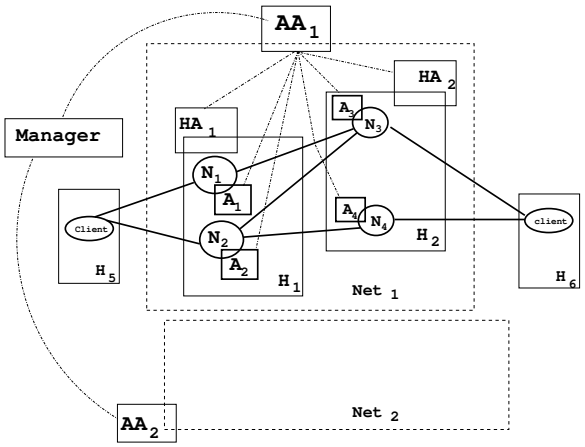

Figure 4. Lira infrastructure for the controlled network

communication. When a hardware or software fault causes paths failure, a reconfiguration is triggered to re-establish path's redundancy. In this example, we consider that manifestation of both a hardware fault (such as a wired connection's interruption or a damage in the physical machine) and a software fault (at operating system, application and logical communication level) have a fail-stop semantics, that is the component stops working.

We are interested in monitoring paths availability, so for a path to be available, all the nodes and links in the corresponding route must be available. Note that failures of a particular link or node may result in unavailability of more than one path. For example, if node N_3 fails, path 1, 2, and 4 become unavailable.

4.1 Lira infrastructure

In this section we describe how the Lira infrastructure is instantiated to manage the network previously described (see Figure 4). Each host H_i is controlled by a Host Agent HA_i, each subnetwork Net_i is controlled by an Application Agent AA_i, while the whole network is controlled by the Manager. The hosts H_5 and H_6 are considered outside the network, so they are not controlled by host agents.

The logical network is also controlled by the Lira agents. The Component Agents A_i control the logical nodes N_i, and they are managed by AA_1. AA_1 may decide to perform a reconfiguration (following the policies specified by the Decision Maker) if it has the necessary information, while it has to ask the general Manager when a global reconfiguration is needed and the local information is not enough. Figure 4 details the Lira management infrastructure.

Each *CompAgent* associated to a logical node exports the enumerated variable *HEALTH_STATE*, which can assume the values *Up, Degraded,* and *Down*. In addition to the defined variables and functions, each CompAgent A_i exports the variable *CONNECTED_NODES*, i.e. the address list of connected nodes, and the function *connectTo(Node nextNode)*, able to connect the local node with the (remote) node specified as parameter. The CompAgent manages also the life cycle of the logical node, by exporting the functions *start, stop, suspend, resume, shutdown*, as defined in Section 3.

Each *Host Agent* exports the functions *install, uninstall, activate, deactivate* and the enumerated variable *HEALTH_STATE*, whose possible values are *Up*, *Degraded* or *Down*. The result of diagnosis over each component is accessible by the agent, and it is notified through a Lira NOTIFY message before a complete crash of the machine. Moreover, the host agent exports the read-only variable *CONNECTED_HOSTS*, which contains the hosts physically connected with the variable's owner. For the host H_2, this variable contains the list H_1, H_3. Note that a host agent can install and activate new logical nodes, creating new routing paths, increasing redundancy and repairing software faults.

The *Application Agent* monitors the subsystem's state and makes it available by exporting the read-only variables *AVAILABLE_PATHS*, *ACTIVE_NODES* and *WORKING_HOSTS*. The first one contains the number of available paths between the clients: for the subsystem controlled by AA_1, the value is 4 (see Table 1). The second one maintains the list of active nodes in the controlled network: in the situation depicted in Figure 4, this variable for AA_1 is $\{N_1, N_2, N_3, N_4\}$. The third one contains the list of still working hosts in controlled network: when HA_i notifies that H_i is down, this variable is modified by the application agent. To change the network topology, the application agent exports the function *connect(Node source, Node dest)*, which is able to connect a source node to a destination node.

The *Manager Agent* controls the subnetworks Net_1 and Net_2 by checking the *WORKING_HOSTS* variable exported by the AppAgents. Thus, it can arrange reconfigurations on the two networks.

4.2 Performing reconfigurations

Reconfigurations can be triggered both at AppAgent and Manager Agent levels by their associate Decision Makers. Decisions are taken when a lower level agent notifies that its controlled component is faulty. Moreover, to prevent faults of the agent itself, higher level agents proactively ask to the controlled agents for the value *HEALTH_STATE* with a frequency T.

As an example of reconfiguration at the ApplAgent level, let's suppose that the node N_3 is starting to work in a degraded manner: the associated agent A_3 notifies the variable

HEALTH_STATE with the value *Degraded*. AA_1 receives the NOTIFY message, and it checks the path availability on the controlled network. There are still more than 2 paths between the clients (see Figure 3) even if one is degraded. In this case three different solutions can be pursued. The first is continuing in the same degraded configuration. Another is to temporarily bypass the node N_3 creating a new logical channel between N_1 and N_4 and waiting for restarting N_3. In this case, the redundancy in terms of paths is reduced because only the first link of the paths is replicated. The third can be to activate a new node N_5 on the host H_2, and to connect it to the client and to the nodes N_1 and N_2, creating new paths. Obviously the different solutions have different costs in terms of time, CPU consuming and paths redundancy. It is responsibility of the DM to select the best one.

We suppose to be in a case where there are not failure/repair dependencies among components and transient phenomena tied to reconfiguration are negligible; then, simple combinatorial models can be evaluated to take the appropriate decision. Assuming that, at given time, the failure probability of a link or component is 10^{-3} when in the Up state, and 10^{-2} when in the $Degraded$ state; for a component which undergoes restart the failure probability becomes $5 * 10^{-3}$ and links and components belonging to the new path have probability $5*10^{-3}$, then the three configurations options can be compared in terms of failure probability P_F. Table 2 summarizes the results of a fault tree analysis, and points out that the best choice is to restart the node N_3.

Policy options	P_F
Working in degraded manner	$1.73848 * 10^{-8}$
Restart node N_3	$5.19695 * 10^{-9}$
Set–up a new path	$4.77510 * 10^{-8}$

Table 2. Policy comparison

In this simple example, the decision can be based on combinatorial models computed a priori. Relaxing the above assumptions makes the analysis more complex by requiring dynamic resolution of state–based models.

5 Conclusions

This work presents an architecture for dependability provisioning which integrates Lira, a light-weight infrastructure for reconfiguring applications, with a model–based Decision Maker. In particular, our goal is to provide a distributed real–time systems with fault–tolerance capabilities. We concentrate only on system reconfiguration as consequence of both faults of software components and host computers that can affect the system.

The work presented in this paper is still ongoing activity and several extensions are currently under investigation to improve and validate our approach. Firstly, Lira infrastructure has to be fault–tolerant itself and different solutions are currently under investigation in this direction. Secondly, for validation purposes a prototype of the case study and a performability evaluation campaign have been planned. Another possible research direction is to improve error diagnosis capabilities of each agent, to better calibrate system reconfiguration.

Acknowledgements This work has been partially supported by the Italian MIUR in the framework of the project "High Quality Software Architectures for Global Computing on Cooperative Wide Area Networks".

References

[1] J. C. Laprie A. Avizienis and B. Randell. Fundamental concepts of dependability. Technical Report 01-145, LAAS, April 2001.

[2] M. Castaldi, A. Carzaniga, P. Inverardi, and A.L. Wolf. A light-weight infrastructure for reconfiguring applications. In *Proceedings of 11th Software Configuration Management Workshop (to appear), Portland, Oregon (USA)*, May 2003.

[3] M. Castaldi and N. D. Ryan. Supporting Component-based Development by Enriching the Traditional API. In *Proc. of Workshop on Generative and Component-based Software Engineering, Erfurt, Germany*, October 2002.

[4] S. Porcarelli F. Di Giandomenico A. Chohra and A. Bondavalli. Tuning of database audits to improve scheduled maintenance in communication systems. In *Proc. of 20th SAFECOMP*, Budapest, Hungary, 2001.

[5] J. Kramer and J. Magee. Analysing dynamic change in software architectures: A case study. *Proc. 4th Int. Conf. on Configurable Distributed Architecture*, pages 91–100, 1998.

[6] J. Magee. Configuration of Distributed Systems. In M. Sloman, editor, *Network and Distributed Systems Management*. Addison-Wesley, 1994.

[7] M. A. Marsan and G. Chiola. On petri nets with deterministic and exponentially distribuited firing times. In *LNCS*, volume 266, pages 132–145. Springer Verlang, 1987.

[8] A. Bondavalli I. Mura S. Chiaradonna R. Filippini S. Poli and F. Sandrini. DEEM: a tool for the dependability modeling and evaluation of multiple phased systems. In *Proc. of Dependable Systems and Networks*, New York, USA, 2000.

[9] Marshall T. Rose. *The Simple Book: An Introduction to Networking Management*. Prentice Hall, April 1996.

[10] Michel Wermelinger. A Hierarchic Architecture Model for Dynamic Reconfiguration. In *2nd Int. Workshop on Software Engineering for Parallel and Distributed Systems*, 1997.

[11] A. J. Young and J. N. Magee. A flexible approach to evolution of reconfigurable systems. *Proc. of IEE/IFIP Int. Workshop on Configurable Distributed Systems*, March 1992.

Dependability in Software Families

Frank van der Linden
Philips Medical Systems, Veenpluis 4-6, 5684 PC Best, NL
Frank.van.der.linden@philips.com

abstract

The paper gives an overview on dependability issues in software families. Most of these issues and the corresponding approach was originating from a series of industrial and research co-operation projects. Many of the results can only be regarded as being preliminary.

1. Introduction

The organisation of software in families is a means to plan and manage reuse. Moreover by building systems in families the quality of the resulting products is expected to improve, because a large part of the software is (re-) used within many products, and therefore it is tested in many real-life situations. Initial involvement of the author with family development, incorporating dependability issues, [12][13] resulted in a series of projects, on the topic of system family development, ARES [9], ESAPS [15] and CAFÉ [16]. Within these projects we have identified the interdependence of 4 software development concerns, BAPO [1]:

- **B**usiness, how to make profit from your products
- **A**rchitecture, technical means to build the software
- **P**rocess, roles, responsibilities and relationships within software development
- **O**rganisation, actual mapping of roles and responsibilities to organisational structures.

2. Families, BAPO and dependability

A set of products is forming a software family if it is based upon the same technological basis (architecture), and there is a clear management of the variation between them. The main reason to set up a family is to save cost and labour, by not making the same functionality twice. Family assets are built independently from the products that use these assets. This means that in order to maintain a dependable family, the dependability issues have to be managed both for the family assets and for the resulting products. In the next sections we discuss the dependability considerations for the different BAPO aspects.

Business

An important *business* decision is the definition of the scope of the family [21]. The scope determines which systems will be part of the family. Dependability issues play a role in the scoping decisions in several ways:

- What is commercial value of a certain quality,
- What are the costs of the introduction of, or maintaining, a specific quality level in (a part of) the family,
- Which parts of the family should exhibit which quality level,
- How to assess the quality levels of the resulting products

In order to relate business originated quality goals to architecture decisions in a family development context, a divide and conquer approach is often useful. For instance, in [19] an investigation is made on the quality goals a business may have, and how these goals relate to architecture attributes. It results in a set of separate quality related taxonomies, each with different characteristics. These taxonomies are related to business originated quality goals. It considers, for example, performance, cost, and variability taxonomy. Such taxonomy is used to determine what are the business originated quality goals, and how to solve them. Part of the taxonomy concerns architecture parameters to support the decision how to satisfy the quality goals.
In [3] a scenario-based method, Attribute Driven Design (ADD), is introduced that support architecture decisions based upon business generated quality goals. Instead of quantifying quality attributes, generic scenarios are introduced dealing with stimuli, the responses, and the architecture elements related to them. An example stimulus is the desire of the user to cancel the previous operation(s), for instance during editing.

Architecture

The *Architecture* has to be set up in such a way that the family can be built in a technical sense. We see a distinction between common software and variant parts, where there is an aim to reduce the variant parts to a minimum. This way the reuse is increased. Moreover, variation will be allowed only in a regulated way, e.g., variants should carry the same interface. This simplifies reasoning about properties of the products in the family, also for dependability aspects. Explicit variation points in the architecture [8],[20] help to determine and manage where and how variation is allowed. Family architectures are often based upon frameworks [22],[24],[25], and/or software busses [12],[13],[17]. Both approaches allow decoupling to a large extend of independent variant parts.
Dependability is related to:

- The selection of variation mechanisms to be used, based upon their dependable properties
- The global mechanisms used within the family to secure dependability levels.
- The use of design decision models [9],[11] to be able to determine the right dependability level for the systems to be built.
- The use of Architecture assessment [7] to evaluate (qualitatively) the dependability properties of the family architecture.

Several case studies have reported the dependability analysis at the system family architecture level. Of course not all causes of failures can be determined at the family architecture level, but at least part of them can be mitigated at the family architecture. In [18] the main topic of investigation is the flexibility (an important family related property) in the case that real-time performance is crucial. In [12],[13] the notion of aspect is introduced. Each aspect relates to a technical solution of a set of related requirements. Often these requirements are dependability related. It is shown that independent design decisions may be made for each of the aspects, resulting in a desired dependability level. Only at a late moment, e.g. system building time, the chosen designs for the different aspects are combined into a working system. This approach allows the developers to focus on a single concern at a time, but still take dependability into account. The case study in [2] reports determines a set of reliability patterns, which may be used in family architecture. Such a pattern can be seen as a solution for a single aspect. As an example a general redundancy pattern is presented. Qualitative analyses of the used parameters, that may influence the redundancy, are classified into distinct groups, each influencing the redundancy in a different way.
The papers [6] and [20] propose a notation for explicit variation points in the architecture. Related to the variation points a decision model is introduced that help the system builder to select the right system in certain circumstances. One of the concerns is to distinguish between functional and non-functional (dependability) properties that may be supported by the architecture. An important observation is to derive from the requirements both *capabilities* of the system (functional) and *forces* (non-functional) which both influence the resulting architecture choices.

Process

The *Process* has to secure the relationship between the stakeholders and the developers of systems in the family based upon the given architecture. It determines the work products to be used during the development of systems in the family. In many cases a family is started with a set of already existing systems, which will become part of the family, and which have to deliver legacy software to be reused within the family [10]. The process around the integration has to deal with ensuring the properties of the constituent systems will be migrated into the family. Dependability aspects related to the process are:

- The presence of feedback loops which inform the family asset developers whether and how their assets in product development are used satisfactorily.
- The use of a quality process (change control board) to ensure the quality of the product.
- The use of a testing process to test the produced assets and systems.
- Qualitative & quantitative analysis of the system during early development phases [2]

Several processes are proposed to support the transition of dependability requirements towards the architecture. In [2] the process supports the prediction of reliability of alternatives failure cause analysis and the relationship of this to the domain analysis process. In [3] the Attribute Driven Design (ADD) process takes scenarios analysis as the basis of the process. In [7] the Architecture Tradeoff Analysis Method (ATAM) is applied within a system family development process. This was useful, however, the investment was high, and not all risks could be discovered by the method.
In [6] and [20] describes a complete family development process, Split. Within this process there are specific activities related to decisions on dependability issues.
The validation and testing in family context is the subject of [23]. It proposes additions to v-model to be able to test efficiently family assets.

Organisation

The *Organisation* groups and separates the people that execute the process. Several forms of organisation will result in different responsibilities between people, which will have an effect in how well the process will be executed [4]. Specific dependability aspects related to the organisation are:

- Separation and co-operation between family and product dependability issues.
- The relationship between testers and other developers.
- The maturity of the organisation [5] reflects itself in the quality of products.

In [5] the maturity of the organisation is related to the variability aspects of the products delivered and how the organisation reflects the family architecture. In [14] it is shown how to obtain dependability of the family in a given organisation structure which, on purpose, does not mirror the process structure exactly. Because the responsibilities of the different developments are spread over many departments, the concern for the quality is shared. As a consequence the quality level is increased.

3. Summary and conclusions

This paper presents a short overview of work on dependability in system family development. It is not claimed to be a complete overview.

Although architectural issues are important in a family development also the business, process and organisation concerns in relation to dependability have to be taken into account. Each of these concerns has reflections on the chosen architecture. The business sets the scope to the family, which has as direct consequence to the architecture itself. In particular it influences the variability available within the family. The process introduces assessment of the dependability issues of the system, often through consideration of architectural properties. The maturity of the organisation relates to architectural issues like the variability of the products delivered and thus into deependability of the family.

Many of the contributions are only preliminary.

4. Literature

[1] Pierre America, Henk Obbink, Rob van Ommering, Frank van der Linden, "CoPAM: A Component-Oriented Platform Architecting Method Family for Product family Engineering", *Proceeding SPLC-1,* Kluwer, Denver, August 2000, pp. 167-180

[2] Marko Auerswald, Martin Herrmann ,Stefan Kowalewski, Vincent Schulte-Coerne, "Reliability-Oriented Product Line Engineering of Embedded Systems", *Proceedings PFE4,* Springer LNCS 2290, Bilbao, October 2001, pp. 83-100

[3] Len Bass, Mark Klein, Felix Bachmann, "Quality Attribute Design Primitives and the Attribute Driven Design Method", *Proceedings PFE4,* Springer LNCS 2290, Bilbao, October 2001, pp. 169-186

[4] Jan Bosch, "Organizing for Software Product Lines", *Proceedings IW-SAPF-3,* Springer LNCS 1951, Las Palmas de Gran Canaria, March 2000, pp. 117-134

[5] Jan Bosch, "Maturity and Evolution in Software Product Lines", *Proceedings SPLC2,* Springer LNCS 2379, San Diego, August 2002, pp. 257-271

[6] Michel Coriat, Jean Jourdan, Fabien Boisboudin, "The SPLIT Method", *Proceeding SPLC-1,* Kluwer, Denver, August 2000, pp.147-166

[7] Stefan Ferber, Peter Heidl, Peter Lutz, "Reviewing Product Line Architectures: Experience Report of ATAM in an Automotive Context", *Proceedings PFE4,* Springer LNCS 2290, Bilbao, October 2001, pp. 364-382

[8] Ivar Jacobson, Martin Griss, Patrik Jonsson, *Software Reuse*, Addison Wesley, 1997

[9] Mehdi Jazayeri, Alexander Ran, Frank van der Linden, *Software Architecture for Product Families,* Addison Wesley, 2000

[10] Isabel John "Integrating Legacy Documentation Assets into a Product Line", *Proceedings PFE4,* Springer LNCS 2290, Bilbao, October 2001, pp. 113-124

[11] A. Karhinen, J. Kuusela, "Structuring Design Decisions for Evolution", *Development and Evolution of Software Architectures for Product families-2,* Springer LNCS 1429, Las Palmas de Gran Canaria, February 1998, pp. 223-234

[12] Frank van der Linden, Jürgen Müller, "Composing Product Families From Reusable Components", *Proceedings 1995 International Symposium and Workshop on Systems Engineering of Computer Based Systems*, 1995, pp. 35-40

[13] Frank J. van der Linden, Jurgen K. Muller, "Creating Architectures with Building Blocks", *IEEE Software* **12** no. 6, Nov. 1995 pp. 51-60,

[14] Frank van der Linden, Jan Gerben Wijnstra, "Platform Engineering for the Medical Domain", *Proceedings PFE4,* Springer LNCS 2290, Bilbao, October 2001, pp. 224-237

[15] Frank van der Linden, "Engineering Software Architectures, Processes and Platforms for System Families", *Proceedings SPLC2,* Springer LNCS 2379, San Diego, August 2002, pp. 383-397

[16] Frank van der Linden, "Software Product Families in Europe: The ESAPS and CAFÉ projects", *IEEE Software,* July/August 2002, pp. 41-49

[17] E. Niemelä, H. Perunka, T. Korpipää, "A Software Bus as a Platform for a Family of Distributed Embedded System Products", *Proceedings Development and Evolution of Software Architectures for Product families-2,* Springer LNCS 1429, Las Palmas de Gran Canaria, February 1998, pp. 14-23

[18] Robert L. Nord, "Meeting the Product Line Goals for an Embedded Real-Time System", *Proceedings IW-SAPF-3,* Springer LNCS 1951, Gran Canaria, March 2000, pp. 19-29

[19] Anu Purhonen, "Quality Attribute Taxonomies for DSP Software Architecture Design", *Proceedings PFE4,* Springer LNCS 2290, Bilbao, October 2001, pp. 238-247

[20] Serge Salicki, Nicolas Farcet, "Expression and Usage of the Variability in the Software Product Lines", *Proceedings PFE4,* Springer LNCS 2290, Bilbao, October 2001, pp. 304-318

[21] Klaus Schmid, "Scoping Software Product Lines", *Proceeding SPLC-1,* Kluwer, Denver, August 2000, pp. 513-532

[22] Clemens Szyperski, *Component Software*, Addison Wesley, 1997

[23] Josef Weingärtner, "Product family engineering and testing in the medical domain - validation aspects", *Proceedings*

PFE4, Springer LNCS 2290, Bilbao, October 2001, pp. 383-387
[24] Jan Gerben Wijnstra, “Critical Factors for a Successful Platform-Based Product Family Approach”, *Proceedings IW-SAPF-3,* Springer LNCS 1951, Las Palmas de Gran Canaria, March 2000, pp. 4-18
[25] Jan Gerben Wijnstra, “Component Frameworks for a Medical Imaging Product Family”, *Proceedings SPLC2,* Springer LNCS 2379, San Diego, August 2002, pp. 68-89