ICSE 2002

Workshop on Architecting Dependable Systems

Orlando, FL, USA
May 25, 2002

Editors

Rogério de Lemos (University of Kent at Canterbury, UK)
Cristina Gacek (University of Newcastle upon Tyne, UK)
Alexander Romanovsky (University of Newcastle upon Tyne, UK)

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.

The aim of this First Workshop on Architecting Dependable Systems is 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, and to jointly formulate an agenda for future research in this emerging area.

We have received 18 submissions mainly from academic contributors. Each paper was reviewed by 3 members of the Program Committee, and a total of 12 papers have been accepted for presentation. We are thankful for the support and dedication of the Program Committee towards making this workshop a success. The Program Committee consisted of:

We highly appreciate that Valérie Issarny (France) and William L Scherlis (USA) have accepted our invitation to give talks during the Workshop on their personal views on architecting dependable systems.

We look forward to an interesting and stimulating workshop.

Rogério de Lemos, Cristina Gacek, and Alexander Romanovsky

Table of Contents

Software Architectures of Dependable Systems: From Closed to Open Systems

Valérie Issarny
INRIA, UR Rocquencourt
Domaine de Voluceau - B.P. 105 - 78153 Le Chesnay France
Valerie.Issarny@inria.fr

INTRODUCTION

Work in the software architecture domain primarily focuses on the standard (as opposed to exceptional) behavior of the software system. However, it is crucial from the perspective of software system robustness to also account for failure occurrences. The next section gives an overview of our past work towards assisting architecting of dependable distributed systems. It is then followed by a discussion on our current and future research work towards addressing dependability requirements of *open* distributed systems, which are expected to become a major class of future distributed systems.

AIDING THE ARCHITECTING OF DEPENDABLE SYSTEMS

Failures may be handled through the integration within the system architecture of components and connectors that provide fault tolerance capabilities. Practically, this means that failures are handled by an underlying fault-tolerance mechanism (e.g., transparent replication management) at the middleware level. Such fault tolerance support must further be coupled with software fault tolerance that relies at least on an exception handling mechanism, which enables the software developer to specify the actions to be undertaken under the occurrence of application-specific and underlying runtime exceptions. We have then carried out research in the two following complementary directions towards assisting architecting of dependable systems.

Systematic aid in the development of middleware architectures for dependable systems: The use of middleware is the current practice for developing distributed systems. Developers compose reusable services provided by proprietary or standard middleware infrastructures to deal with non-functional requirements. However, developers still have to design and implement middleware architectures combining available services in a way that best fits the application's requirements. In order to ease this task, we have developed an environment that provides [1]: (i) an ADL for modeling middleware architectures, (ii) a repository populated with architectural descriptions of middleware services, and (iii) automated support for composing middleware architectures out of available services according to target non-functional properties, and for quantitatively assessing the composed architectures in terms of performance and reliability.

Architecture-based exception handling: As previously raised, it is necessary to complement fault-tolerance support provided by the underlying middleware architecture, with support for software fault tolerance so as to enable application-specific fault-tolerance. We have thus proposed a solution to architecture-based exception handling [2], which complements exception handling implemented within components and connectors. Our solution lies in: (i) extending the ADL so as to enable the specification of required changes to the architecture in the presence of failures, and (ii) associated runtime support for enabling resulting dynamic reconfigurations.

FUTURE RESEARCH DIRECTIONS

The above results have been proven successful for assisting the architecting of robust distributed systems that are closed, i.e., systems whose components depend on a single administrative domain and are known at design time. However, future distributed systems will increasingly be open, which raises new issues for making them dependable. In this context, we are in particular undertaking research in the following directions: (i) Architecting open distributed systems in a way that accounts for mobility, which requires support for the dynamic composition and quality assessment of architecture instances; and (ii) Design of fault-tolerance mechanisms for open distributed systems considering that the systems span multiple administrative domains and hence cannot accommodate locking-based solutions as, e.g., enforced by transactional processing [3].

REFERENCES

[1] Issarny, V., Kloukinas, C. and Zarras, A. Systematic Aid in the Development of Middleware Architectures. *Communications of the ACM*. 2002. To appear.

[2] Issarny, V. and Banâtre J-P. Architecture-based Exception Handling. *Proc. of HICSS'34*. 2001.

[3] Tartanoglu, F., Issarny, V., Levy, N. and Romanovsky, A., Dependability in the Web Service Architecture. *Proc. of WADS*. 2002.

Dependability and Architecture: An HDCP Perspective

William L Scherlis
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA
scherlis@cmu.edu

Dependability and architecture

It is generally accepted in engineering practice that dependability, like security, is best designed into a system from the outset—that certain architectural design characteristics positively correlate with overall system dependability. Dependability ("reliance that can justifiably be placed") necessarily comprises, in this respect, a broad range of attributes such as fault tolerance, robustness, code safety, safe concurrency, usability, and self-healing behavior. And, in the interests of scalability, our definition of dependability must be meaningful relative both to overall system behavior and to the behavior of internal (sub)systems with respect to clients.

For the most part, we lack a systematic scientific linking of architectural characteristics with overall dependability outcomes, and this applies to most of the attributes mentioned above. There is only preliminary literature identifying concrete hypotheses concerning "favorable" architectural characteristics. But, perhaps more frustrating, even in the presence of such hypotheses we lack the ability to measure directly the critical variables to evaluate their validity, and must instead rely on weak surrogates. In this respect even small successes in measurement can help engineers develop more prescriptive approaches to architecting and implementing high dependability systems. (The substantial premiums paid by customers, for example, for higher availability in data centers supports this claim.)

Another, perhaps greater, challenge may be described as "dependability remediation"—a rubric meant to include both the evaluation and the improvement of existing systems with respect to particular dependability attributes. How can overall dependability be evaluated, and the relative contributions be determined for various identified design decisions, and with respect to particular attributes? It is tempting to dismiss this as an almost hopelessly broad question. But some focus can be achieved by addressing specifically the development of "incremental techniques" for remediation (i.e., measurement, improvement, assurance)—in which individual actions of engineers to make local improvements yield increments of overall improvement. This incrementality is a feature of successful open source engineering practice.

The High Dependability Computing Project (HDCP) was recently initiated by NASA Ames to address some of these issues in the context of future NASA systems. The research is directed at understanding dependability issues in larger systems and developing practicable techniques for evolution and improvement. The program combines research on measurement (correlative process/product measures for various dependability attributes), assurance (analytically based dependability claims), and technological intervention (techniques for designing dependability systems, or improving the dependability of existing systems).

The HDCP is meant to be a genuinely collaborative effort—NASA systems and projects are objects of study, in order to understand the challenges of moving techniques for measurement, assurance, and improvement from laboratory into practice. A diverse portfolio of research teams and approaches in HDCP reduces the risk of engagement for NASA mission organizations engaging in testbed projects. In order to achieve the diversity of approaches and the extent of logistical support required to support the testbed projects, Carnegie Mellon is collaborating with five other universities, including University of Southern California, University of Maryland, MIT, University of Washington, and University of Wisconsin Milwaukee. In addition, the team will be augmented by researchers funded through a recently-announced solicitation from the National Science Foundation—Highly Dependable Computing and Communication Systems Research (HDCCSR)—that builds on HDCP testbed projects.

Within HDCP, for example, a number of aspects of architecture are addressed, including both architectural design and architecture implementation. Examples include self-healing architecture designs, robustness testing of internal services and the interfaces through which they are delivered, development of architectural metrics, and model-based evaluation of API compliance.

Acknowledgement. The author wishes to acknowledge support through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

Dependability in the Web Service Architecture

Ferda Tartanoglu[1], Valérie Issarny[2]
INRIA, UR Rocquencourt
Domaine de Voluceau - B.P. 105
78153 Le Chesnay France
[1]Galip-Ferda.Tartanoglu@inria.fr,
[2]Valerie.Issarny@inria.fr

Alexander Romanovsky
University of Newcastle upon Tyne
Department of Computing Science, NE1 7RU, UK
Alexander.Romanovsky@newcastle.ac.uk

Nicole Levy
Université de Versailles Saint-Quentin en Yvelines
45 avenue des Etats-Unis
78035 Versailles Cedex, France
Nicole.Levy@prism.uvsq.fr

ABSTRACT

In comparison with the state of the art in the field of Web Services architectures and their composition, we propose to exploit the concept of CA Actions to enable to dependable composition of Web Services. CA Actions introduce a mechanism for structuring fault tolerant concurrent systems through the generalization of the concepts of atomic actions and transactions, and are adapted to the composition of autonomous services.

1. INTRODUCTION

The Web service architecture targets the development of applications based on the XML standard [15], which eases the construction of distributed systems by enabling the dynamic integration of applications distributed over the Internet, independent of their underlying platforms. Currently, the main constituents of the Web service architecture are the following: (i) WSDL (Web Service Description Language) is a language based on XML that is proposed by the W3C for describing the interfaces of Web services [14]; (2) UDDI (Universal Description, Discovery and Lookup) is a specification of a registry for dynamically locating and advertising Web services [12]; (3) SOAP (Simple Object Access Protocol) defines a lightweight protocol for information exchange [13]. SOAP sets the rules of how to encode data in XML; it also includes conventions for partly pre-scribing the invocation semantics (either synchronous or asynchronous) as well as the SOAP mapping to HTTP.

There already exist platforms that are compliant with the Web service architecture, including .NET [6] and J2EE [10]. In addition, integration within CORBA is being addressed [9]. Even though the Web service architecture is quite recent and not fully mature, it is anticipated that it will play a prominent role in the development of next generation distributed systems mainly due to the strong support from industry and the huge effort in this area. However, there is clearly a number of research challenges in supporting the thorough development of distributed systems based on Web services. One such challenge relates to using Web services in developing business processes, which requires a support for composing Web services in a way that guarantees dependability of the resulting composed services. This calls for developing new architectural principles of building such composed systems, in general, and for studying specialized connectors "glueing" Web services, in particular, so that the resulting composition can deal with failures occurring at the level of the individual service components by allowing co-operative failure handling.

Solutions that are being investigated towards the above goal subdivide into (i) the definition of XML-based languages for the specification of Web services composition and (ii) revisiting classical transactional support so as to cope with the specifics of Web services (e.g., crossing administrative domains, Web latency), i.e., defining connectors offering transactional properties over the Internet. The two next sections respectively overview existing solutions to the two aforementioned points, and assess them with respect to Web service composition and its dependability. In particular, it is emphasized that while the transaction concept offers a powerful abstraction to deal with the occurrence of failures in closed systems, it imposes too strong constraints over component systems in open environment such as Web services. The main constraint relates to supporting backward error recovery that, firstly, requires isolating component systems for the duration of the embedded (nested) transaction in which they get involved and hence contradicts the intrinsic autonomy of Web services, and, secondly, relies on returning the service state back, which is not applicable in many real-life situations which involve documents, goods, money as well as humans (clients, operators, managers, etc.).

In the light of the above, the paper puts forward a solution based on forward error recovery, which enables dealing with dependability of composed Web services, and has no impact on the autonomy of the Web services, while exploiting their possible support for dependability (e.g., transaction support at the level of each service). Our solution, introduced in Section 4, lies in system structuring in terms of co-operative actions that have a well-defined behavior, both in the absence and in the presence of service failures. Finally, Section 5 discusses our current and future work aiming at enhancing the Web service architecture for the sake of dependability.

2. COMPOSING WEB SERVICES

Composing Web services relates to dealing with the assembly of autonomous components so as to deliver a new service out of the components' primitive services, given the corresponding published interfaces. In the current Web service architecture, interfaces are described in WSDL and published through UDDI. However, supporting composition requires further addressing: (i) the specification of the composition, (ii) ensuring that the services are composed in a way that guarantees the consistency of both the individual services and the overall composition. There are three main proposals in the area:

- WFSL (Web Services Flow Language) addresses the former issue. It enables describing the composition of Web services through two complementary models [4]: (i) a flow model that serves specifying a sequence of actions over services in a way similar to workflow schema, and (ii) a global model that further describes the interactions between service providers and requesters and hence details the realization of each action of the flow model.
- XLANG deals with the latter issue by enriching the description of Web services' interfaces with behavioral specification. It aims at allowing the formal specification of business process as stateful long-running interactions [11]. Business processes always involve more than one participant. Hence, the full description of a process must not only show the behavior of each participant but also the way these behaviors match to produce the overall process. The focus is on the publicly visible behavior in the form of exchanged messages. More precisely, the interface of a Web service is enriched with the specification of how to consistently use the Web service, stating the necessary sequence of interactions. This is quite similar to the work done in the area of Architecture Description Language [5], when concerned with the formal specification of port and role behavior for checking the consistency of the architecture.
- XL is a language targeting the specification of Web service composition. It is fully based on XML for the specification and composition of Web services [3] and is built upon concepts of imperative programming languages, the CSP process algebra and workflow management.

There are other efforts towards supporting the composition of Web services: similar to the aforementioned solutions, these proposals rely on a new language and supporting environment, which are still under definition. While there is not yet a consensus about how Web services composition should be supported, existing work allows us to identify two major trends: (i) composition based on workflow management, (ii) using transactions to enforce dependability. The former trend justifies from the concern of supporting business processes but also by the fact that the composition process applies to autonomous services belonging to distinct administrative domains. However, the needed extension to WSDL still requires investigation. For instance, the behavioral specification for individual services introduced by XLANG complements the composition specification introduced by WFSL for checking composition consistency and also to possibly automate the generation of interactions. The next section shows why, from our standpoint, transactions do not offer solutions to the dependable composition of Web services.

3. TRANSACTIONS FOR THE DEPENDABLE COMPOSITION OF WEB SERVICES

Transactions have been proven successful in enforcing dependability in closed distributed systems. The base transactional model that is the most used guarantees ACID (atomicity, consistency, isolation, durability) properties over computations. Enforcing ACID properties typically requires introducing protocols for: (i) locking resources (i.e., two-phase locking) that are accessed for the duration of the embedding transaction, and (ii) committing transactions (i.e., two or three phases validation protocols). However, such a model is not suited for making the composition of Web services transactional for at least two reasons:

- The management of transactions that are distributed over Web services requires cooperation among the transactional support of individual Web services –if any-, which may not be compliant with each other and may not be willing to do so given their intrinsic autonomy and the fact that they span different administrative domains.
- Locking accessed resources (i.e., the Web service itself in the most general case) until the termination of the embedding transaction is not applicable to Web services, still due to their autonomy, and also the fact that they potentially have a large number of concurrent clients that will not stand extensive delays.

Enhanced transactional models may be considered to alleviate the latter shortcoming. In particular, the split model where transactions may split into a number of concurrent sub-transactions that can commit independently allows reducing the latency due to locking. Typically, sub-transactions are matched to the transactions already supported by Web services (e.g., transactional booking offered by a service) and hence transactions over composed services do not alter the access latency as offered by the individual services. Enforcing the atomicity property over a transaction that has been split into a number of sub-transactions then requires using compensation over committed sub-transactions in the case of sub-transaction abortion. Using compensation comes along with the specification of compensating operations supported by Web services for all the operations they offer. Such an issue is in particular addressed by XLANG [11]. However, it should be further accounted that using compensation for aborting distributed transactions must extend to all the participating Web services (i.e., cascading compensation by analogy with cascading abort), which is not addressed by XLANG due to its focus on the behavioral specification of individual Web services for assisting their composition.

Developing transactional supports for dependable Web service composition is an active area of research that is still in its infancy. Ongoing work includes BTP (Business Transaction Protocol) [8], TIP (Transaction Internet Protocol) [2] and extension to the OMG/J2EE Activity Service [7]. However, proposed solutions do not cope with all the specifics of Web services. From our standpoint, a major source of difficulty lies in the use of backward error recovery in an open system such as the Internet, which is mainly oriented towards tolerating hardware faults but poorly suited to the deployment of cooperation-based mechanisms over autonomous component systems that often require cooperative application-level exception handling among component systems. An alternative then lies in relying on the existing support of Web services for managing internal concurrency control so as to guarantee keeping the consistency of services, while relying on forward error recovery for ensuring the dependability of service composition. The next section introduces such a solution, which builds upon the concept of Coordinated Atomic (CA) Actions [16].

4. USING CA ACTIONS FOR THE DEPENDABLE COMPOSITION OF WEB SERVICES

The CA Actions [16] are a structuring mechanism for developing dependable concurrent systems through the generalization of the concepts of atomic actions and transactions. Basically, atomic actions are used for controlling cooperative concurrency among a set of participating processes and for realizing coordinated forward error recovery using exception handling, and transactions are used for maintaining the coherency of shared external resources that are competitively accessed by concurrent actions (either CA Actions or not). Then, a CA Action realizes an atomic state transition where: (i) the initial state is defined by the initial state SP_i of the participants Pi and the states SR_j of the external resources R_j at the time they were accessed by the CA Action, (ii) the final state is defined by the state of the participants (SP_i‘) at the action's termination (either standard or exceptional) and the state of the accessed external resources (SR_j‘ in the case of either standard termination or exceptional termination without abortion, SR_j in the case of exceptional termination with abortion).

CA Action naturally fits the specification of Web service composition:

- Each participant specifies the interactions with each composed Web service, stating the role of the specific Web service in the composition. In particular, the participant specifies actions to be undertaken when the Web service signals an exception, which may be either handled locally to the participant or be propagated to the level of the embedding CA Action. The latter then leads to co-operative exception handling according to the exceptional specification of the CA Action.
- Each Web service is viewed an external resource. However, unlike the base CA Action model, interactions are not enforced to be transactional. The interactions adhere to the semantics of the Web service operations that are invoked. An interaction may then be transactional if the given operation that is called is. However, transactions do not span multiple interactions.
- The standard specification of the CA Action gives the expected behavior of the composed Web service in either the absence of failures or in the presence of failures that are locally handled (i.e., either system-level exceptions or programmed exceptions signaled by Web services operations that do not need to be cooperatively handled at the CA Action level).
- The exceptional specification of the CA Action states the behavior of the composed Web service under the occurrence of failure at one or more of the participants, that need cooperative exception handling. The resulting forward recovery may then realize a relaxed form of atomicity (i.e., even when individual operations of the Web service are transactional, its intermediate states may be accessed by external actions between such operations executed within a given action) when Web services offer both transactional and compensating operations (to be used in cooperative handling of exceptions).

To apply the general concept of CA actions in the context of composing Web services, we introduce the concept of WSCA (Web Service Composition Action). WSCAs differ from CA Actions in (i) relaxing the transactional requirements over external interactions (which are not suitable for wide-area open systems) and (ii) introducing composition of WSCAs where each participant may actually be a WSCA, which is abstracted as a single unit of computation from the standpoint of peer participants.

In order to illustrate the use of WSCAs for specifying the composition of Web services, we take the classical example of a travel service. We consider joint booking of accommodation and flights using respective hotel and airline Web services. Then, the composed Web service is specified using nested WSCA as follows. The outermost WSCA TravelAgent comprises the User and the Travel participants. The Travel participant is a nested WSCA that composes the Airline and the Hotel participants. A diagrammatic specification of the WSCAs is shown in Figure 1.

In TravelAgent, the User participant requests the Travel participant to book a return ticket and a hotel room for the duration of the given stay. Then, the two Travel WSCA participants respectively request the Hotel Web service for a hotel room and the Airline Web service for a return ticket, given the departure and return dates provided by the user. Each participant request is subdivided into reservation for the given period and subsequent booking if the reservation succeeds. In the case where either the reservation or the booking fails, the participant raises the unavailable exception that is cooperatively handled at the level of the Travel WSCA. If both participants signal the unavailable exception, then Travel signals the abort exception so that the exception gets handled by TravelAgent in a cooperation with the User (e.g., by choosing a alternative date). If only one participant raises the unavailable exception, cooperative exception handling includes an attempt by the other participant to find an alternative booking. If this retry fails, the booking that has succeeded is cancelled and the abort exception is signaled to the embedding TravelAgent WSCA for recovery with user intervention.

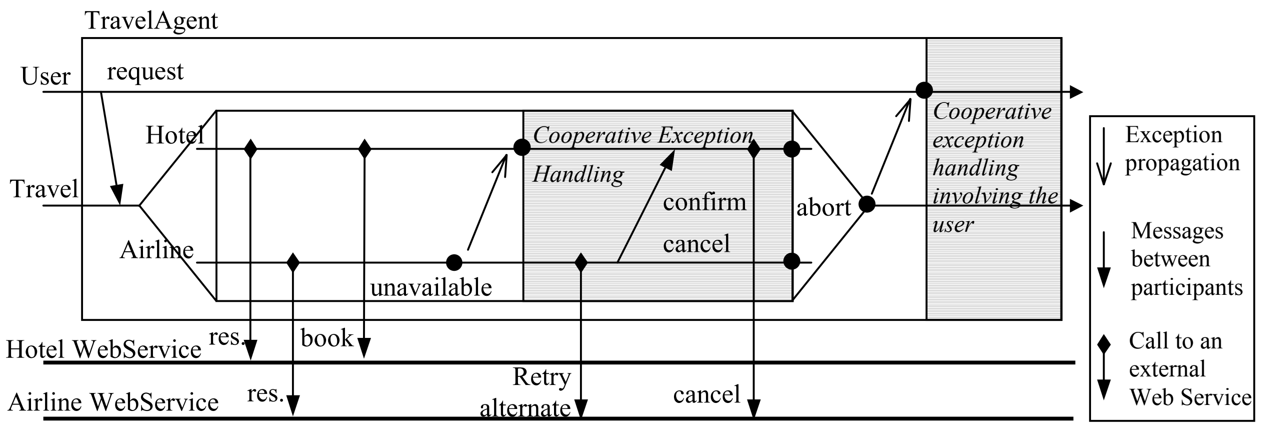


Figure 1. WSCA for composing Web Services

5. CONCLUSIONS

The Web service architecture is expected to play a major role in developing next generation distributed systems. However, the architecture needs to evolve to support all the requirements appertained to distributed systems. Addressing such requirements relates, in particular, in reusing solutions from the distributed system community. However, most solutions will not be reusable as is, mainly because of the openness of the Internet. Hence, making evolve the Web service architecture to support the thorough development of distributed systems raises a number of challenges.

This paper has addressed one of the issues raised in this context, which is the dependable composition of Web services, i.e., understanding how fault tolerance should be addressed in the Web service architecture. While dependability in closed distributed systems is conveniently addressed by transactions when concerned with both concurrency control and failure occurrences, it can hardly rely on such a mechanism in an open environment. Our solution to this concern lies in forward error recovery that enables accounting for the specific of Web services and that leads to structure Web services-based systems in terms of co-operative actions. In particular, we are able to address dependable service composition in a way that neither undermines the Web service's autonomy nor increases their individual access latency.

Further work is still needed towards offering a complete solution to dependability in the Web service architecture. Our next step is on the formal specification of WSCAs using the B formal notation [1] so as to precisely characterize the dependable behavior of WSCAs, and in particular the relaxed form of atomicity that is introduced. Our aim is to propose an architectural style for specifying architectures of systems based on WSCAs by defining associated connectors and components. We will then investigate the definition of an architecture description language and associated methods and tools for supporting the development of dependable systems based on the Web service architecture.

6. ACKNOWLEDGMENTS

This research is partially supported by the European IST DSoS project (IST-1999-11585) [1].

7. REFERENCES

[1] Abrial, J. R. *The B Book – Assigning Programs to Meanings*. Cambridge University Press. 1996.

[2] Evans, K. Transaction Internet Protocol: Facilitating Distributed Internet Applications. *Proceedings of the W3C Workshop on Web services*. 2001.

[3] Florescu, D., and Kossmann, D. An XML Programming Language for Web Services Specification and Composition. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*. 2001.

[4] Leymann, F. Web Services Flow Language (WSFL 1.0). IBM Software Group. http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf. 2001.

[5] Medvidovic, N. and Taylor, R. N. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*. 2000.

[6] Microsoft. .NET. http://msdn.microsoft.com/net/.

[7] Mikalsen, T., Rouvellou, I., and Tai, S. Reliability of Composed Web Services – From Object Transactions to Web Transactions. *Proceedings of the OOPSLA'01 Workshop on Object-Oriented Web Services*. 2001.

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

[8] Oasis Committee. Business Transaction Protocol. Draft Specification. January 2002. http://www.oasis-open.org/committees/business-transactions/

[9] OMG. Corba Web Services. OMG TC Document orbos/2001-06-07. http://www.omg.org. 2001.

[10] Sun Microsystems Inc. Java 2 Platform, Enterprise Edition (J2EE). http://java.sun.com/j2ee/

[11] Thatte, S. XLANG: Web Services for Business Process Design. Microsoft Corporation. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm. 2001.

[12] UDDI Specification. Version 2.0. http://www.uddi.org/specification.html. 2001.

[13] W3C. Simple Object Access Protocol (SOAP) 1.1. W3C Note. http://www.w3.org/TR/SOAP/. 2000.

[14] W3C. Web Services Description language (WSDL) 1.1. W3C Note. http://www.w3.org/TR/2001/NOTE-wsdl-20010315. 2001.

[15] W3C. Second Edition of the Extensible Markup Language (XML). 1.0 Specification. W3C Recommendation. http://www.w3.org/TR/2000/REC-xml-2001006. 2000.

[16] Xu, S., Randell, B., Romanovsky, A., Rubira, C. M. F., Stroud, R. J., and Wu, Z. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. *Proceedings of the IEEE Symposium on Fault Tolerant Computing*. 1995.

RAIC: Architecting Dependable Systems through Redundancy and Just-In-Time Testing

Chang Liu
Information of Computer Science
University of California, Irvine
Irvine, CA 92697, USA
+1(949)824-2703
liu@ics.uci.edu

Debra J. Richardson
Information of Computer Science
University of California, Irvine
Irvine, CA 92697, USA
+1(949)824-7353
djr@ics.uci.edu

ABSTRACT

Redundant Arrays of Independent Components (RAIC) is a technology that uses groups of similar or identical distributed components to provide dependable services. RAIC allows components in a redundant array to be added or removed dynamically during run-time, effectively making software components "hot-swappable" and thus achieves greater overall dependability. RAIC controllers use the just-in-time component testing technique to detect component failures and the component state recovery technique to bring replacement components up-to-date. This position paper gives a brief overview of RAIC and a proof-of-concept example to illustrate how problems occur during run-time can be masked by RAIC and would not affect smooth operations of the application.

1. INTRODUCTION

Software dependability has become a sufficiently important aspect of computer systems to warrant attention to the architectural level. Architectural representations capture overall designs of software systems while abstracting away low-level details [2,15]. Architectural representations can assist in improving software dependability in a number of ways. For example, architectural representations can be used in testing and verification of both the designs and the systems to help achieve higher dependability [3,13,14]. Furthermore, system dependability can be enhanced by adopting appropriate architectures and architecture styles. *Redundant arrays of independent components* (RAIC) is an attempt to achieve higher dependability and other desirable properties through a specific architecture style [8,9]. RAIC uses groups of similar or identical distributed components to provide higher dependability, better performance, or greater flexibility than what can possibly be achieved by any of those individual components.

With the introduction of Microsoft .NET platform and the release of tools such as Visual Studio .NET that bring the creation of XML web services to the masses, it is reasonable to expect more software applications to be built on top of remote third-party software components or XML web services. Unlike in-house components or off-the-shelf ones, these remote third-party components are not under the control of application developers. They can be upgraded without notice even when applications are running. This makes it even more important to ensure that application dependability is not affected by component failures. RAIC is designed to solve this problem through an architectural approach.

In this position paper, RAIC is briefly explained. A proof-of-concept *Light* example is given to illustrate the functions of RAIC controllers and how failures in *Light* components are detected and masked while the *Light* applications run smoothly.

2. RAIC OVERVIEW

A *redundant component array* (also referred to as RAIC) is a group of similar or identical components. The group uses the services from one or more components inside the group to provide services to applications. Applications connect a RAIC and use it as a single component. Applications typically do not have any knowledge of the underlying individual components.

Depending on the types and relations of components in a RAIC, it can be used for many different purposes under different types of RAIC controllers. A *RAIC controller* contains software code that coordinates individual software components in a RAIC. Not all types of RAIC controllers apply to all combinations of component types and relations. It is essential to determine component types and relations prior to configuring a RAIC.

Component types. There are mainly two types of components in terms of whether or not they maintain internal states: *stateless* components and *stateful* ones.

In a stateful component, each public method can be either *state-preserving*, *state-changing*, or *state-defining*. The return value of a method can be either *state-dependent* or *state-independent*.

A RAIC can be either *static* or *dynamic*. Components in a static RAIC are explicitly assigned by mechanisms outside the RAIC, whereas components in a dynamic RAIC may be discovered and incorporated by the RAIC controller during run-time. Dynamic RAIC controllers may use directories such as UDDI to locate new components [16]. Either way, RAIC controllers allow addition or removal of components during run-time and take care of component state recovery when necessary as new stateful components are added. Note that components may be added to or removed from a static RAIC at runtime. The difference between a static RAIC and a dynamic RAIC is that in a dynamic RAIC, the RAIC controller takes responsibility of component discovery, whereas in a static RAIC, components are explicitly assigned to the RAIC controller.

Component state recovery. Component types and method properties help RAIC controllers to decide what to do in the event of *component state recovery* [10]. For stateless components, no state recovery is necessary. A newly created component can be used in place of another component right away. For stateful components, their states must be restored before they are used in lieu of other components. There are primarily two ways to perform state recovery: *snapshot-based recovery* and *invocation-history-based recovery*. The snapshot-based approach assumes that the state of a component is represented by its snapshot, which is a copy of all of its internal variables. The invocation-history-based approach assumes that placing an exact same call sequence to equivalent components results in the same component state. This implies that components are deterministic.

An invocation can have a method property of *state-defining*, *state-changing*, or *state-preserving*. Method properties help reduce the amount of call histories that are needed for state recovery purposes.

State-defining methods change the state of a component to specific states regardless of the previous state of the component. Different method parameters may bring the same components to different states. But same method parameters always bring components to the same states even though their previous state may be different.

State-changing methods may change the state of a component. Invocations of state-changing methods must be stored for future state recovery, unless invocations to state-defining methods are placed later.

State-preserving methods do not change the state of a component at all. Thus, it is not necessary to re-invoke calls to methods of this type. All state-preserving invocations can be safely trimmed off.

Component relations. There are many aspects of relations between components. Nearly universally applicable are aspects such as interfaces, functionalities, domains, and snapshots. Not applicable to all components, but important nonetheless, are aspects such as security, invocation price, performance, and others. Relations of multiple components can be derived from binary relations among components.

As an example, interfaces of two components can have the following relations: *identical* (≡), *equivalent* (=), *similar* (≈), *inclusionary* (≤), or *incomparable* (≠).

While it is possible to programmatically determine interface relations by analyzing interface specifications, other relations, such as functionality relations, sometimes can only be manually determined.

The component relations are used to determine the integration strategy, i.e. to choose how the components interact. For example, RAIC controllers can partition components inside a RAIC into equivalent classes and use only components inside the same class to replace each other until they run out.

RAIC levels. Most of these RAIC strategies and policies are configurable. RAIC levels describe the level and the purpose of the integration of components in RAIC. The following is a list of RAIC levels:

- RAIC-1: Exact mirror redundancy
- RAIC-2: Approximate mirror redundancy
- RAIC-3: Shifting lopsided redundancy
- RAIC-4: Fixed lopsided redundancy
- RAIC-5: Reciprocal redundancy
- RAIC-6: Reciprocal domain redundancy
- RAIC-0: No redundancy

RAIC controllers can also use different invocation models, including:

- RAIC-a: Sequential invocation
- RAIC-b: Synchronous parallel invocation
- RAIC-c: Asynchronous parallel invocation

RAIC controllers need to make judgment about the return values from individual components in the redundant array to determine whether or not to invoke another component, which result to select, or how to merge return values. To do that, RAIC controllers need to evaluate return values at run-time. Just-in-time component testing is designed for this purpose [6].

Just-in-time component testing is different from traditional software testing. Traditional software testing techniques use various methods to determine, through test execution, if a software application, a software component, or an even smaller unit of software code behaves as expected. Usually this is done by feeding the software-code-under-test with some pre-determined data, or test input, and comparing the result with pre-determined expected output, or test oracle. Traditional software testing happens in the development phase, when software is still under development and has not been deployed to the end user. Code that is used for testing purposes, or test harnesses, are usually removed or filtered out through conditional compilation or by other means before the final software product is deployed. Just-in-time component testing differs from traditional testing in the following aspects:

1. JIT testing happens even after application deployment. Code responsible for JIT testing is an integral part of the final software product and is shipped as such.

2. JIT testing uses mostly live input data that are unknown ahead of time. Thus it is difficult, sometimes impossible, to know if the result value is correct. Therefore, heuristics and other means must be used in place of traditional test oracles.

3. When in rare cases that predetermined test inputs are used in JIT testing, it is extremely important to ensure that test runs on these test inputs are very efficient, because any test execution on predetermined data is pure overhead during run-time and will directly place a negative impact on application performance. In

comparison, test case efficiency weighs much less in traditional software testing. In addition, any fabricated test data must not change the state of the component-under-test unless the pending invocation is a state-defining one with state-independent return results.

JIT component testing happens in run-time. This is very similar to another type of testing - perpetual testing. Perpetual testing is a class of software testing techniques that seeks seamless, perpetual analysis and testing of software products through development, deployment, and evolution [12]. The difference between JIT testing and perpetual testing is that perpetual testing is optional and removable, whereas JIT testing is an integral part of the final product. The purpose of perpetual testing is to obtain more insight of the software-product-under-test, which is usually under full control of testers, through monitoring in the real environment and thus gain data that are not available from laboratories. JIT testing, on the other hand, tries to determine on-the-fly if the result from a foreign software component is trustworthy. The foreign software component is usually not under control of the application programmer. Even their availabilities are not guaranteed.

JIT component testing is also different from certain self-checking built-in mechanisms [7,19]. The difference is that JIT testing code resides in the RAIC controller instead of the actual components.

RAIC can be used for purposes such as fault-tolerance, result refinement, and performance enhancement, to name just a few, where it is desirable to put components with incomparable interfaces or exclusionary domains in the same RAIC. When used for dependability-enhancing purposes only, however, it is likely that all components in a RAIC have similar interface relations, identical domain relations, and non-incomparable functionalities so that they can be used interchangeably. It is also unlikely that there is a need to invoke different versions of components-under-upgrade simultaneously except when just-in-time testing needs component voting to verify return results. Therefore, for dependability purposes, "RAIC-2a[$\approx_i$,$\equiv_d$]", a special case of RAIC, is most commonly used [8].

3. THE *LIGHT* EXAMPLE[1]

There is a *Light* component that provides a simple software *light* service, which simulates an adjustable light. The *light* can be turned on and turned off. The intensity of the *light* can be adjusted through another method invocation. The following is a skeleton code in C# that defines the *Light* component [4]. The *MethodProperty* attributes specify that all three methods are *state-defining*, meaning that they change the state of the component to a specific state regardless of which state the component was in prior to the method invocation.

[1] The *Light* component example was used in [18].

```
public interface ILight
{
  [MethodProperty(MthdProperty.StateDefining)]
  int TurnOn();

  [MethodProperty(MthdProperty.StateDefining)]
  int SetIntensity(int intensity);

  [MethodProperty(MthdProperty.StateDefining)]
  int TurnOff();
}

public class Light: MarshalByRefObject, ILight
{
  // ...
}
```

There are two versions of the *Light* component. The first version allows arbitrary method invocations. An upgrade to the *Light* component, however, requires *TurnOn()* to be called before *SetIntensity()* or *TurnOff()* can be called[2]. Similarly, *TurnOff()* cannot be called if the *light* is already off. An exception would be thrown if these requirements are not met.

There are also two applications that use the *Light* component. The first application, *LightApp1*, simply calls *TurnOn(), SetIntensity(),* and *TurnOff()* repeatedly.

```
public class LightApp1
{
  public static void Main(string[] args)
  {
    int pause_in_seconds = 3;

    Light light = new Light();

    for (int i=1; i<=100; i++)
    {
      light.TurnOn();
      Thread.Sleep(pause_in_seconds * 1000);
      light.SetIntensity(50);
      Thread.Sleep(pause_in_seconds * 1000);
      light.TurnOff();
      Thread.Sleep(pause_in_seconds * 1000);
    }
  }
}
```

The second application, *LightApp2*, is similar to *LightApp1*. The difference is that *LightApp2* does not call *TurnOn()* at all.

```
public class LightApp2
{
  public static void Main(string[] args)
  {
    int pause_in_seconds = 3;

    Light light = new Light();

    for (int i=1; i<=100; i++)
    {
      light.SetIntensity(50);
      Thread.Sleep(pause_in_seconds * 1000);
      light.TurnOff();
      Thread.Sleep(pause_in_seconds * 1000);
    }
  }
}
```

[2] In this example, we only consider normal states when deciding method properties. Therefore, in the new version, all three methods are still state-defining.

Apparently, both *Light* applications work well with the first version of the *Light* component. The upgrade of the *Light* component would break *LightApp2* but would not affect *LightApp1*.

In a distributed system where *LightApp1* and *LightApp2* run side-by-side, if an on-line upgrading of the *Light* component is attempted, *LightApp2* will undoubtedly be interrupted. An attempt to revert the *Light* component to its original version would fix *LightApp2*, but would deny *LightApp1*'s access to upgraded features of the *Light* component. By using RAIC, these problems can be avoided. Here is what happens with RAIC:

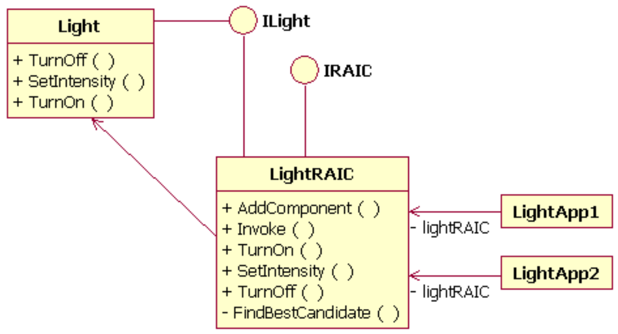


Figure 1. With RAIC, the *Light* applications uses component *LightRAIC* instead of component *Light*.

First, instead of using the concrete *Light* component directly, the *light* applications use a new component *LightRAIC*, which has the same interface *ILight* as *Light*, as shown in Figure 1.

```
public class LightRAIC
  : MarshalByRefObject, IRAIC, ILight
{
  //...
}
```

```
LightRAIC light = new LightRAIC();

for (int i=1; i<=100; i++)
{
  //...
  light.SetIntensity(50);
  //...
}
```

Second, in a system-wide configuration, *LightRAIC* is defined as "RAIC-2a[]", which means it uses the sequential invocation model and treats all components inside as stateful. Its policy is set to "latest version first". Then, the first version of the *Light* component is added to the RAIC as its only member component. After that, both *LightApp1* and *LightApp2* can run smoothly using their own instances of *LightRAIC*.

Third, during the on-line upgrading, the upgraded version of the *Light* component is added to *LightRAIC*. In *LightApp1*, the RAIC controller switches to the new component because its policy asks it to always try to use the component with the latest version. It first brings the status of the new component up-to-date by placing all calls in its trimmed invocation history to the new component. Then it places the current call to the new component and thus switches the application to the new component. *LightApp1* only experiences a brief delay during the switch. The operation of *LightApp1* continues without any disruption. The length of the delay depends on the number of items in the trimmed invocation history. In this case, since all three method invocations are state-defining, there is only one item in the trimmed invocation history no matter how long the invocation history is.

In *LightApp2*, the RAIC controller also tries to switch to the new component because of the same "latest version first" invocation policy. Its just-in-time component testing mechanism detects an exception when the first *SetIntensity()* method call is placed without a preceding *TurnOn()* call. JIT testing treats the exception as a failure. The RAIC controller then tries the next available component in the RAIC, which is the original *Light* component. Since the state of that component is already up-to-date, the RAIC controller goes ahead and places the current method call and returns the result to *LightApp2*. During the on-line upgrading, *LightApp2* does not experiment any failure at all. The exception in the upgraded component was masked by the RAIC controller. *LightApp2* notices only a brief delay, the length of which is approximately one method call to the upgraded component. After that, all subsequent calls go to the original component without delay. To *LightApp2*, the on-line upgrading never happened.

Note that in this scenario, there is no application-or component-specific configuration definition that specifies which application works with which component.

In the pre-.NET era, two versions of the same component (DLL) cannot appear on one system on Windows platforms, which means it would be impossible to have *LightApp1* using the upgraded version of the *Light* component and *LightApp2* using the original one on the same system, let alone upgrading the component at run-time.

On .NET platforms, with the support for side-by-side execution of different versions of the same component, it is now possible to do so. To achieve this, however, extra efforts are required from component developers, application developers, or system administrators to explicitly specify which application should use which version of the component. In addition, to avoid problems that may be created by over-paranoid component developers, application developers, or system administrators, .NET platform allows them to override decisions made by each other, which undoubtedly could further require more efforts from all of them. In short, even on the currently state-of-art .NET platforms, this is achievable but not pain-free.

With RAIC, this scenario is not just achievable, it is trivial with the help of just-in-time testing and component state recovery.

In this example, the two *Light* components used are two versions of the same component. It demonstrates that problems in on-line upgrading can be avoided by using RAIC [11]. RAIC, however, is not limited to arrays of different versions of the same components. In fact, the two *Light* components here can be regarded as two different components that provide similar services and all the results still hold. Examples that use different Light components can be found at [5].

4. LIMITATIONS AND PENDING TASKS

Currently, both the just-in-time component testing technique and the component state recovery technique have significant limitations. For example, if a component is connected to a persistent external storage such as a database, neither snapshot-based nor invocation-history-based state recovery technique can fully recover component states[3]. While some limitations are fundamental to the approach and cannot be removed by improving these two techniques alone, we feel that both techniques work or could work under broad enough circumstances that this work could produce practical results. In addition, many limitations may be lifted by improved techniques. We are working to add better heuristics to just-in-time testing and more approaches to component state recovery. For example, we are considering using component dependency information to broaden applicability of the component state recovery technique [17].

5. SUMMARY

In summary, dependability-through-redundancy can be achieved by adopting a special RAIC architecture style. Just-in-time component testing and component state recovery techniques can be used to coordinate redundant components so that applications are not exposed to the complexity of the integration of redundant components.

6. REFERENCES

[1] R. Barga and D.B. Lomet, "Phoenix: Making Applications Robust," *Proceedings of 1999 ACM SIGMOD Conference*, Philadelphia, PA (June 1999) (562-564).

[2] L. Bass, P. Clements, and R. Kazman, "Software Architecture in Practice", SEI Series, Addison-Wesley January 1998. ISBN: 0201199300.

[3] A. Bertolino, F. Corradini, P. Inverardi, H. Muccini, "Deriving Test Plans From Architectural Descriptions", *Proceedings of the 22nd international conference on Software engineering*, p.220-229, June 04-11, 2000, Limerick, Ireland.

[4] ECMA, "Standard ECMA-334: C# Language Specification", December 2001. http://www.ecma.ch/ecma1/STAND/ecma-334.htm.

[5] C. Liu, "The RAIC Web Site," 2002, http://www.ics.uci.edu/~cliu1/RAIC.

[6] C. Liu, "Just-In-Time Component Testing and Redundant Arrays of Independent Components", *Doctoral Dissertation, Information and Computer Science, University of California, Irvine* (in progress).

[7] C. Liu and D. J. Richardson, "Software Components with Retrospectors," International Workshop on the Role of Software Architecture in Testing and Analysis, Marsala, Sicily, Italy, July, 1998.

[8] C. Liu and D. J. Richardson, "Redundant Arrays of Independent Components", *Technical Report 2002-09, Information and Computer Science, University of California, Irvine*, March 2002.

[9] C. Liu and D. J. Richardson, "The RAIC Architectural Style", Submitted to the 10th International Symposium on the Foundations of Software Engineering (FSE-10), March 2002.

[10] C. Liu and D. J. Richardson, "Specifying Component Method Properties for Component State Recovery in RAIC", *Accepted by the Fifth ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly (ICSE2002), Orlando, Florida, USA,* May 19-20, 2002.

[11] C. Liu and D. J. Richardson, "Using RAIC for Dependable On-line Upgrading of Distributed Systems", *Submitted to the Dependable On-line Upgrading of Distributed Systems Workshop held in conjunction with COMPSAC 2002 (August 26-29 2002, Oxford, England)*, March 2002.

[12] L. J. Osterweil, L. A. Clarke, D. J. Richardson, and M. Young, "Perpetual Testing," *Proceedings of the Ninth International Software Quality Week*, 1996.

[13] M. D. Rice, S. B. Seidman, "An Approach To Architectural Analysis And Testing", *Proceedings of the third international workshop on Software architecture*, p.121-123, November 01-05, 1998, Orlando, Florida, United States.

[14] D. J. Richardson and A. L. Wolf, "Software Testing At The Architectural Level", *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, p.68-71, October 16-18, 1996, San Francisco, California, United States.

[15] M. Shaw and D. Garlan, "Software Architecture: Perspectives on an Emerging Discipline", Prentice-Hall, Englewood Cliffs, NJ, 1996. ISBN: 0131829572.

[16] UDDI, "UDDI 2.0 Specification", 2001. http://www.uddi.org/specification.html.

[17] M. Vieira, M. Dias, D. J. Richardson, "Describing Dependencies in Component Access Points", *Proceedings of the 4th Workshop on Component Based Software Engineering, 23rd International Conference on Software Engineering (ICSE'01, Toronto, Canada)*, May, 2001 pp.115-118.

[18] C. H. Wittenberg, "Testing Component-Based Software", *International Symposium on Software Testing and Analysis (ISSTA'2000),* Portland, Oregon, 22-25 August 2000.

[19] S. S. Yau and R. C. Cheung, "Design of Self Checking Software," In Proceedings of the International Conference on Reliable Software, April, 1975, pp. 450-457.

[3] This problem was addressed by Phoenix [1].

An Idealized Fault-Tolerant Architectural Component

Paulo Asterio de C. Guerra
Cecília Mary F. Rubira
Instituto de Computação
Universidade Estadual de Campinas, Brazil
{asterio,cmrubira}@ic.unicamp.br

Rogério de Lemos
Computing Laboratory
University of Kent at Canterbury, UK
r.delemos@ukc.ac.uk

ABSTRACT

Component-based systems built from existing 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. However, the problem is exacerbated if we consider the current trend of integrating third-party software components, which allow neither code inspection nor changes. To leverage the reliability properties of these systems, we need solutions at the architectural level that are able to guide the structuring of unreliable components into a fault tolerant architecture. In this paper, we present an approach for structuring fault tolerant component-based systems based on the C2 architectural style.

1. INTRODUCTION

Modern computer systems require evolving software that is built from existing software components, developed by independent sources [6]. Instead of relying on traditional software assurance technology that has shown not to be effective for this kind of systems [24], alternative approaches have to be sought in order for obtaining trustworthy systems. One of these approaches is fault-tolerance, which is associated with the ability of a system to deliver services according with its specification in spite the presence of faults [12]. In this paper, we employ the concept of idealized fault tolerant component [1] for describing fault-tolerant component-based systems, at the architectural level.

For representing software systems at the architectural level, we have chosen the C2 architectural style for its ability to incorporate heterogeneous off-the-shelf components [15]. However, this ability of combining existing components is achieved through rules on topology and communication between the components (communication through broadcasting of asynchronous messages) that complicate the incorporation of fault-tolerance mechanisms into C2 software architectures, especially those mechanisms for error detection and fault containment [6, 9].

Research into describing software architectures with respect to their dependability properties has gained attention recently [17,20,21]. Nonetheless, rigorous specification of exception handling models and of exception propagation at the architecture level remains an open issue [11].

Particularly related to the architectural approach presented in this paper, there has been work on exception handling and software fault tolerance. The work on exception handling has focused on *configuration exceptions*, which are exceptional events that have to be handled at the configuration level of architectures [11]. In terms of software fault tolerance, the principles used for obtaining software diversity have also been employed in the reliable evolution of software systems, specifically, the upgrading of software components. While the core idea of the *Hercules framework* [8] is derived from concepts associated with *recovery blocks* [17], the notion of *multi-versioning connectors* (MVC) [16], in the context of C2 architectures, is derived from concepts associated with *N-version programming* [3]. The architectural approach presented in this paper is distinct from the work referred above since its focus is on structuring concepts to be applied in a broader class of exceptional conditions and fault-tolerance mechanisms. The aim is to structure, at the architecture level, fault-tolerant component-based systems that use off-the-shelf components. For that, we define an idealized C2 component with structure and behaviour equivalent to the idealized fault-tolerant component [1]. This idealized C2 component can then be used as a building block for a system of design patterns that implement the idealized fault-tolerant component for concurrent distributed systems [5].

The rest of this paper is structured as follows. Section 2 gives a brief overview of fault-tolerance and the C2 architectural style. Section 3 describes the proposed architectural solution of the idealized component, along with an small illustrative example. Final conclusions are given in section 4.

2. BACKGROUND

The capability of a system to tolerate faults is highly dependent on the software architecture [4]. Though, the structure of the system should allow fault tolerant mechanisms to operate in an orchestrated way with the system functions, without unnecessarily increasing the complexity of the system [17].

2.1. Fault Tolerance

The basic strategy to achieve fault tolerance in a system can be divided into two steps [13]. The first step, called *error processing*, is concerned with the system internal state, aiming to: detect errors that are caused by activation of faults, the diagnosis of the

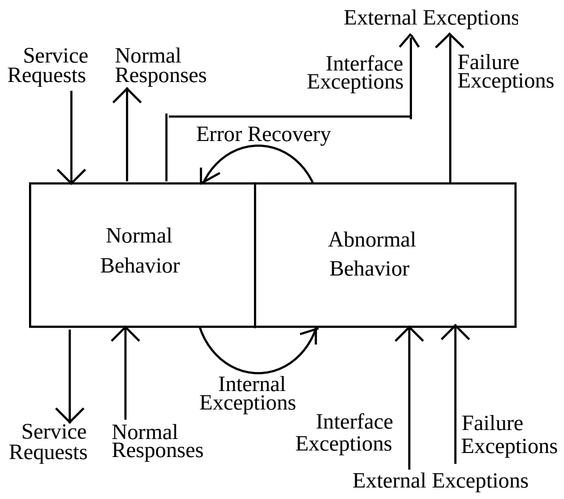


Figure 1. Idealized Fault-Tolerant Component

erroneous states, and recovery to error free states. The second step, called *fault treatment*, is concerned with the sources of faults that may affect the system and includes: fault localization, and fault removal.

Our work mainly concentrates on providing error processing at the architectural level of software systems. The *idealized fault-tolerant component* [1] is a structuring concept for the coherent provision of fault tolerance in a system (Figure 1). Through this concept, we can allocate fault-tolerance responsibilities to the various parts of a system in an orderly fashion, and model the system recursively, such that each: component can itself be considered as a system on its own, which has an internal design containing further sub-components [1].

The communication between idealized fault-tolerant components is only through request/response messages. Upon receiving a request for a service, an idealized component will react with a *normal response* if the request is successfully processed or an *external exception*, otherwise. This external exception may be due to an invalid service request, in which case it is called an *interface exception*, or due to a failure in processing a valid request, in which case it is called a *failure exception*. *Internal exceptions* are associated with errors detected within a component that may be corrected, allowing the operation to be completed successfully; otherwise, they are propagated as external exceptions.

An idealized component must provide appropriate handlers for all exceptions it may be exposed to. Thus, the internal structure of an idealized component has two distinct parts: one that implements its *normal behaviour*, when no exceptions occur, and another that implements its *abnormal behaviour*, which deals with the exceptional conditions. This separation of concerns, applied recursively to components, subsystems and the overall system, greatly simplifies the structuring of fault tolerance systems, allowing their complexity to be manageable.

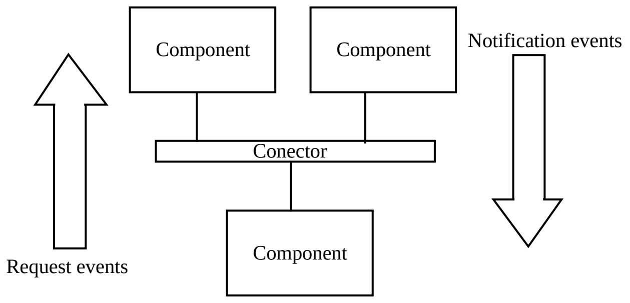


Figure 2. C2 style basic elements.

2.2. The C2 Architectural Style

The C2 architectural style is a component-based style directed at supporting large grain reuse and flexible system composition, emphasizing weak bindings between components [23]. In this style components of a system may be completely unaware of each other, as when one integrates various commercial off-the-shelf components (COTS), which may have heterogeneous style and implementation language. These components communicate only through asynchronous messages mediated by connectors that are responsible for message routing, broadcasting and filtering. Interface and architectural mismatches are dealt with by using wrappers for encapsulating each component [9].

Both components and connectors in the C2 architectural style (Figure 2) have a *top interface* and a *bottom interface*. 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.

There are two types of messages in C2: requests and notifications. Requests flow up through the system's layers and notifications flow down. In response to a request, a component may emit a notification back to the components below, through its bottom interface. Upon receiving a notification, a component may react, as if a service was requested, with the *implicit invocation* of one of its operations.

3. PROPOSED ARCHITECTURE

3.1. Overall Structure of the Idealized C2 Component

The objective of this section is to define an idealized C2 component (iC2C), which should be equivalent, in terms of behaviour and structure, to the idealized fault-tolerant component (iFTC) [1]. The implementation of an iC2C should be able to use any C2 component without any restrictions. Furthermore, it should also be possible for integrating idealized C2 components into any C2 configurations, thus allowing the interaction of iC2Cs with other idealized and/or regular C2 components.

The first task was to extend the C2 message type hierarchy to allow for the various message types defined for the iFTC. This was a relatively simple task, since service requests and normal

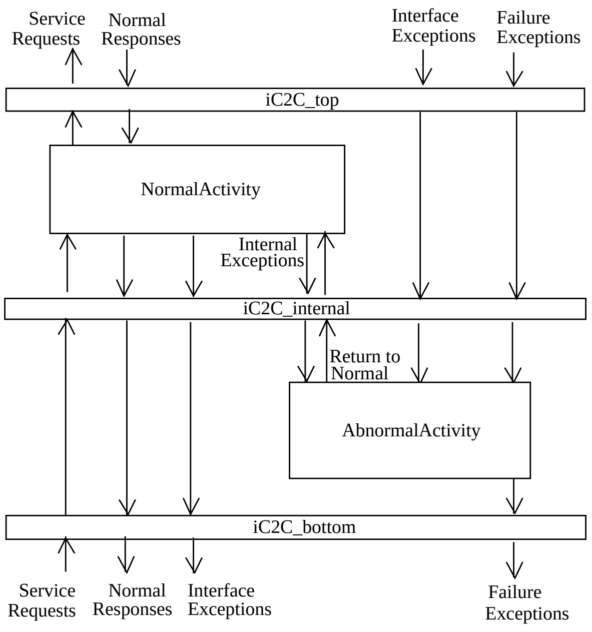


Figure 3. Idealized C2 Component (i2C2)

responses of an iFTC were directly mapped as requests and notifications in the C2 architecture. As interface and failure exceptions of an iFTC flow in the same direction as a normal response, they were considered subtypes of notifications in the C2 architecture.

In order to minimize the impact of fault tolerance provisions on the system complexity we have decoupled the normal activity and abnormal activity parts of the idealized component. This outcome has lead to an overall structure for the iC2C that has two distinct components and three connectors, as shown in Figure 3.

The iC2C NormalActivity component implements the normal behaviour, and is responsible for error detection during normal operation, and the signalling of interface and internal exceptions. The iC2C AbnormalActivity component is responsible for error recovery, and the signalling of failure exceptions. For consistency, the signalling of an internal exception by an iFTC was mapped as a subtype of notification, and, the “return to normal” , flowing in the opposite direction, was mapped as a request. In the course of error recovery, the AbnormalActivity component may also emit requests and receive notifications, which are not represented in Figure 3. More specifically, this design allows the AbnormalActivity component to be notified about state changes of the NormalActivity component and request operations which may change that state.

The connectors of our iC2C shown in Figure 3 are specialized, reusable, C2 connectors with the following roles:

(i) The iC2C_bottom connector connects the iC2C with the lower components of a C2 configuration, and serializes the requests received. Once a request is accepted, this connector queues new requests that are received until completion of the first request. When a request is completed, a notification is sent back, which may be a normal response, an interface exception or a failure exception.

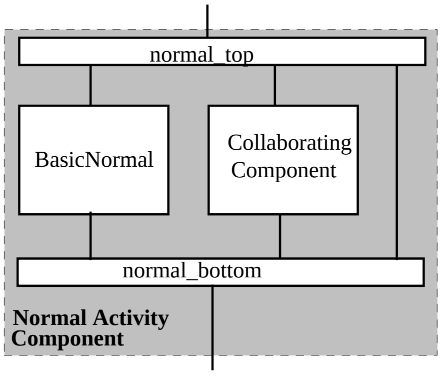


Figure 4. Normal Activity Component

(ii) The iC2C_internal connector controls message flow inside the iC2C, selecting the destination of each message received based on its originator, the message type and the operational state of the iC2C (either under normal or abnormal operation).

(iii) The iC2C_top connector connects the iC2C with the upper components of a C2 configuration, which may provide services to the NormalActivity and/or AbnormalActivity components.

The overall structure defined for the idealized C2 component makes it fully compliant with the component's rules of the C2 architectural style. This allows an iC2C to be integrated into any C2 configuration and interact with components of a larger system. When this interaction establishes a chain of iC2C components the external exceptions raised by a component can be handled by a lower level component (in the C2 sense of “upper” and “lower”) allowing hierarchical structuring of error recovery activities. An iC2C may also interact with a regular C2 component, either requesting or providing services.

3.2. Structuring the Normal Activity Component

In this section, we describe in more detail how the NormalActivity component can be implemented from existing C2 components.

As previously mentioned, the NormalActivity component is responsible for the implementation of the normal behaviour of the idealized C2 component, and the detection of errors that may affect the normal behaviour. Since a NormalActivity component should be built from existing C2 components, and these components might not have error detection capabilities, there is the need to add error detection capabilities to the existing C2 component. The architectural solution for implementing a NormalActivity component is shown in Figure 4, for a particular configuration of two components. The existing C2 component, identified as the BasicNormal component, and any other component required for the provision of additional error detection capabilities, are wrapped by a pair of special-purpose connectors

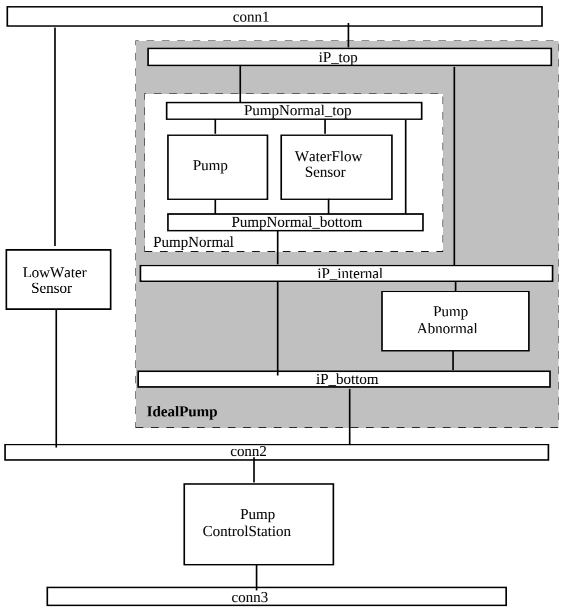


Figure 5. C2 Configuration for Fault Tolerant PumpControlStation

(normal_top and normal_bottom), following the pattern of the multi-versioning connector (MVC) [16]. These connectors coordinate the collaboration between the components, and provide the NormalActivity component with the capabilities for error detection. These capabilities can be associated to the operations provided either by the BasicNormal component or the other collaborating components. Errors are detected by checking the pre- and post-conditions, and invariants associated to the operations [21]. The proposed approach was inspired by the concepts of *coordination contracts* [2] and *co-operative connectors* [14].

On top of the above architecture for an ideal C2 component (iC2C), the Normal Activity component could also interact with other components outside the scope of the iC2C. In this case, the component should be placed higher in the C2 configuration, and the normal_top connector should act as a proxy of the component in the context of the NormalActivity component.

Another special case is when components placed at lower levels of a C2 architecture require to access services provided by other collaborating components wrapped into the NormalActivity component. In this case, the interface of the iC2C can extend that of the BasicNormal for including the required services.

3.3. A Small Example

In order to illustrate the structuring concepts presented in this paper, we refer to a small example extracted from the Mine Pump Control System [20]. The subsystem that we consider is responsible for draining the sump of the mine, and contains the following existing C2 components:

(i) PumpControlStation - controls the draining of the sump by turning on/off a physical pump according to the level of the water in the sump.

(ii) LowWaterSensor - signals when the level of water is low.

(iii) Pump - commands the pump to be turned on/off.

(iv) WaterFlowSensor - signals whether water flows from the sump.

The fault model for the above subsystem assumes that transient faults can affect the operation of the physical pump when reacting to commands from Pump.

The C2 architecture of the subsystem is shown in Figure 5, where the IdealPump is implemented as an idealized C2 component (iC2C). The NormalActivity component of IdealPump, which is PumpNormal, consists of components Pump and WaterFlowSensor that are joined into a collaboration that is coordinated by the PumpNormal_bottom connector. This same connector is responsible for detecting errors in IdealPump, checking the WaterFlowSensor status after a pump on/off requested, and raising an internal exception when the expected condition is not met.

The AbnormalActivity component (PumpAbnormal) is responsible for processing the error, by issuing retry requests to the Pump until either the normal operation is resumed or the exception is propagated to PumpControlStation.

4. CONCLUSIONS

In this paper, we have investigated the structuring of fault-tolerant component-based systems, at the architectural level. For the purpose of our work we have employed the C2 architectural style [23], which is a style that promotes the development of component-based systems using off-the-shelf components. The intent was to provide an idealized C2 component with structure and behaviour equivalent to the idealized fault-tolerant component [1].

The communication rules between components in the C2 style, namely the synchronicity and broadcasting of messages, although desirable from the point of view of component-based design, they complicate the incorporation of fault-tolerance mechanisms into architectures that are instantiations of this style [6, 9]. Another difficulty that we encountered was the restrictions imposed by the C2 topology rules. For solving these problems we employed constructs similar to multi-versioning connector [16], consisting of pairs of collaborating connectors to define fault containment boundaries within the system, and synchronized communications within the idealized C2 component using notifications as acknowledgments of requests. In addition to the work describe above, we have also defined an idealized C2 connector. This fault tolerant architectural element is especially useful considering that connectors in the C2 architectural style are more than simple communication primitives, and that the architectural approach advocated in this paper requires connectors to be also a place of computation.

Our results demonstrate the feasibility of the proposed approach for the C2 architectural style, and suggest their application to other architectural styles also belonging to the *interacting processes style* category, which are styles dominated by communication patterns among independent, usually concurrent, processes [19].

ACKNOWLEDGMENTS

Paulo Guerra is partially supported by CAPES/Brazil. Cecília Rubira and Paulo Guerra are supported by the FINEP/Brazil “Advanced Information Systems” Project (PRONEX-SAI-7697102200). Cecília Rubira is also supported by CNPq/Brazil under grant no. 351592/97-0.

REFERENCES

[1] T. Anderson, and P. A. Lee. *Fault Tolerance: Principles and Practice.* Prentice-Hall, 1981.

[2] L. F. Andrade, and J. L. Fiadeiro. Feature modeling and composition with coordination contracts. In *Proceedings Feature Interaction in Composed System (ECOOP 2001)*, pages 49--54. Universitat Karlsruhe, 2001.

[3] A. Avizienis. The N-Version Approach to Fault Tolerant Software. *IEEE Transactions on Software Engineering*, 11(2):1491--1501, December 1995.

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

[5] D. M. Beder, B. Randell, A. Romanovsky, and C. M. F. Rubira. On Applying Coordinated Atomic Actions and Dependable Software Architectures for Developing Complex Systems. In *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, Magdeburg, Germany, May 2-4, 2001, pp. 103-112, IEEE Computer Society Press.

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

[7] T. D. Chandra. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225--267, March 1996.

[8] J. E. Cook, and J. A. Dage. Highly reliable upgrading of components. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 203--212, New York, NY, May 1999. ACM Press.

[9] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17--26, November 1995.

[10] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1--26, March 1999.

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

[12] J. C. Laprie. *Dependability: A Unifying Concept for Reliable Computing and Fault Tolerance*, chapter 1, pages 1--28. Blackwell Scientific Publications Ltd., 1989.

[13] J. C. Laprie. Dependability: Basic concepts and terminology. In *Special Issue of the Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS- 25)*. IEEE Computer Society Press, 1995.

[14] R. de Lemos. Describing evolving dependable systems using co-operative software architectures. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pages 320--329. 2001.

[15] 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 (SSR'97)*, 1997.

[16] M. Rakic, and N. Medvidovic. Increasing the confidence in o-the-shelf components: A software connector-based approach. In *Proceedings of the 2001 Symposium on Software Reusability (SSR'01)*, pages 11--18. ACM/SIGSOFT, May 2001.

[17] B. Randell, and J. Xu. The evolution of the recovery block concept, In *Software Fault Tolerance*, chapter 1. John Wiley Sons Ltd., 1995.

[18] T. Saridakis, and V. Issarny. Developing Dependable Systems using Software Architecture. Technical report, INRIA/IRISA, 1999.

[19] M. Shaw, and P. Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In *Proceedings of the COMPSAC97, First International Computer Software and Applications Conference*, 1997.

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

[21] D. Sotirovski. Towards fault-tolerant software architectures. In R. Kazman, P. Kruchten, C. Verhoef, and H. Van Vliet, editors, *Working IEEE/IFIP Conference on Software Architecture*, pages 7--13, Los Alamitos, CA, 2001.

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

[23] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, 22(6):390--406, June 1996.

[24] G. Vecellio, and W. M. Thomas. Issues in the assurance of component-based software. In *Proceedings of the 2000 International Workshop on Component-Based Software Engineering*. Carnegie Mellon Software Engineering Institute, 2000.

Tolerating Architectural Mismatches

Rogério de Lemos
University of Kent at Canterbury
UK
+44-1227-823628
r.delemos@ukc.ac.uk

Cristina Gacek
University of Newcastle upon Tyne
UK
+44-191-222-5153
cristina.gacek@ncl.ac.uk

Alexander Romanovsky
University of Newcastle upon Tyne
UK
+44-191-222-8135
alexander.romanovsky@ncl.ac.uk

ABSTRACT
The integrity of complex software systems built from existing components is becoming more dependent on the integrity of the mechanisms used to interconnect these components, in particular, on the ability of these mechanisms to cope with architectural mismatches that might exist between components. This paper is based on the assumption that architectural mismatches always exist in such systems, so the need to handle them in run-time. When developing complex software systems, the problem is not only to identify the appropriate components, but also to make sure that these components are interconnected in a way that allows mismatches to be tolerated. The resulting architectural solution should be a system based on existing components, which are independent in their nature, but are able to interact in well-understood ways. To find a solution to this problem we apply general principles of fault tolerance in the context of dealing with architectural mismatches

1. INTRODUCTION
There are several concepts that are relevant to addressing the tolerance of architectural mismatches in software systems. In this section we introduce the concepts of software architectures, architectural mismatches, and dependability which are pivotal for understanding the need for tolerating architectural mismatches in software systems, as well as the approach we are taking to tackle this problem.

Software architecture can be defined as the structure(s) of a system, which comprise software components, the externally visible properties of those components and the relationships among them [1]. A software architecture is usually described in terms of its components, connectors and their configuration. The way a software architecture is configured defines how various connectors are used to mediate the interactions among components.

As a result of combining several architectural elements using a specific configuration, architectural mismatches may occur [6]. Architectural mismatches are logical inconsistencies between constraints of various architectural elements being composed. An architectural mismatch occurs when the assumptions that a component makes about another component, or the rest of the system, do not match. That is, the assumptions associated with the service provided by a component are different from the assumptions associated with the services required by a component for behaving as specified [8]. These assumptions can be related to the nature of components and connectors (control and data models, and synchronisation protocols), the global system structure, or the process of building the system [6, 9]. Traditionally, mismatches have been dealt with statically [5, 3], by means of analysis and removal.

Dependability is a vital property of any system justifying the reliance that can be placed on the service it delivers [7]. A system *failure* occurs when a system service deviates from the behaviour expected by the user. An *error* is the part of the system state that is liable to lead to the subsequent failure. The adjudged or hypothesized cause of an error is a *fault*. Fault tolerance is a method of achieving dependability working under assumptions that a system contains faults (e.g. ones made by humans while developing or using systems, or caused by aging hardware) and aiming at providing the required services in spite of them. Fault tolerance is carried by error processing, aiming at removing errors from the system state before failures happen, and fault treatment, aiming at preventing faults from being once again activated. Error processing typically consists of three steps: error detection, error diagnosis and error recovery. Providing system fault tolerance plays an ever-growing role in achieving system dependability as there are many evidences proving that it is not possible to rid the system and system execution from faults. These include the growing complexity of software, operators' mistakes, and failures in the environment in which the system operates.

There are many reasons to support our claim that it is impossible to detect and correct all possible architectural mismatches statically, and because of this, we believe that it is vital to be able to build systems that can tolerate such mismatches. This is mainly due to the complexity of modern systems and restricted applicability of the static methods of correcting mismatches (c.f. software design faults). First of all, complex applications have complex software architectures in which components are interconnected in complex ways and have many parameters and characteristics to be taken into account while building, they have to meet many functional and non-functional requirements which often have to be expressed at the level of software architecture. Secondly, architects make mistakes while defining software architectures, in general, and while dealing with mismatches, in particular. Thirdly, there is a strong trend in using off-the-shelf elements while building complex applications and because of the very nature of such elements some information about their architectural characteristics may be unavailable. Lastly, current software systems may undergo dynamic reconfiguration, adding uncertainty about the various architectural elements present at any point in time. In this paper we show that architectural mismatches can be tolerated.

2. ARCHITECTURAL MISMATCHES

All Architectural mismatches occur because of inconsistencies among the given architectural elements. These inconsistencies can be stated in terms of the features (or characteristics) exhibited by the parts at hand. Features of architectural elements and their groupings may be inherent to the architectural style(s) used, or specific to the application at hand. This occurs because architectural styles impose constraints on the kinds of architectural elements that may be present and on their configurations [9], yet they do not prescribe all the features that may be present in an application [5]. During software development, the software architecture is incrementally refined following the refinement of the system's definition. Initially, the software architecture is defined in terms of architectural styles, thus binding the style specific features. Subsequently, as the architecture is further refined towards the life-cycle architecture, application specific features are bound. This is exemplified on Table 1 (adapted from [4]). Every time an architectural feature is bound there exists the potential for an architectural mismatch to be introduced. Hence, we refer to architectural mismatches as being: *style-specific* if their presence is brought about by some architectural feature(s) the style(s) imposes; or as *application-specific* if their presence is due to architectural decisions imposed by the application at hand (not the particular style(s) used).

We believe that in the context of dependability, an architectural mismatch is an undesired, though expected, circumstance, which must be identified as a *design fault* (in the terminology from [7]). When a mismatch is activated, it produces an *error caused by mismatch* (ECM) that can either be latent or detected. Similarly to errors, only a subset of ECMs can be detected as such (see Figure 1). Additional information is needed to allow an error to be associated with a mismatch. Eventually, there is a system failure when the ECM affects the service delivered by the system.

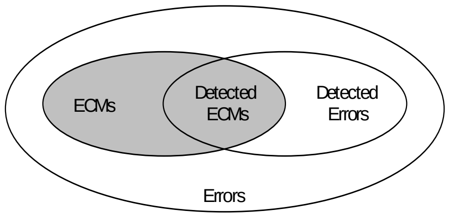


Figure 1. Detected errors caused by mismatches

For describing the means for dealing with architectural mismatches, we draw an analogy with faults, which can be avoided, removed or tolerated. Faults are tolerated when they cannot be avoided, and their removal is not worthwhile or their existence is not known beforehand. The same kind of issues happens with architectural mismatches. Mismatches can be *avoided* by imposing strict rules on how components should be built and integrated, which leads to bespoke products. Mismatches can be *removed* when integrating arbitrary components by using static analysis methods and techniques [5, 3]. However, this does not guarantee the absence of mismatches since risk and cost tradeoffs may hinder their removal, or system integrators may not be aware of their existence (similarly, research has shown that residual faults in software systems are inevitable). Consequently, mismatches should be *tolerated* by processing ECMs and treating mismatches, otherwise the system might fail.

3. MISMATCH TOLERANCE

The tolerance of architectural mismatches during runtime relies on *ECM processing*, which comprises three steps [7]. These are:

- *The detection of ECMs*, which identifies erroneous states that were caused by mismatches.
- *The diagnosis of ECMs*, which assesses the system damages caused by the detected ECMs.
- *The recovery from ECMs*, which brings the system to an error-free state.

However, error processing is not sufficient if we want to avoid the recurrence of the same architectural mismatch, also there is the need to treat mismatches, in the same way that faults are treated [7]. *Mismatch treatment* involves diagnosis, which determines the cause (localization and nature) of the ECM, isolation that prevents a new activation of the mismatch, and reconfiguration, which modifies the structure of the system for the mismatch free components to provide an adequate, perhaps degraded, service.

The intent of fault tolerant techniques is to structure systems to inhibit the propagation of errors, and to facilitate their detection and the recovery from them. Similarly, when dealing with architectural mismatches, there is the need to structure systems at the architectural level to avoid the propagation of ECMs, to facilitate ECMs detection and recovery, and to make difficult the reactivation of architectural mismatches.

The particular problem associated with mismatch tolerance is that we are dealing with two levels of abstraction: the architectural level where the mismatches are introduced, and the execution level where the ECMs detection and recovery takes place. Hence, the needs for identifying what are the potential consequences upon the state of the system when an architectural mismatch is activated. This additional information is fundamental for distinguishing ECMs from other system errors, thus providing the basis for defining an architectural solution for tolerating mismatches.

A motivation for specifying mechanisms for handling architectural mismatches at the architectural level, instead, for example, during implementation, is that the nature of mismatches and the context in which they should be fixed would be lost at the later stages of software development. Making once again an analogy with fault tolerance, it has been shown that the same type of problem exists when exception handling is not considered in the context of the software lifecycle [2]. Moreover, we cannot expect that a general runtime mechanism would be able to handle a wide range of architectural mismatches, in the same way that there is no sufficiently general fault tolerant mechanism that can handle all classes of faults. It is envisaged that different classes of mismatches will require different types of detection mechanisms and fixes that have to be specified at the architectural level.

3.1 ECMs' Processing

As previously discussed, the detection of an ECM implies the presence of a mismatch. For a mismatch to be activated some preconditions must be satisfied, which can be defined in terms of systems states and features' inconsistencies [5].

Upon error detection, one must first determine whether that particular instance is an ECM or not. For an error to be detected as an ECM we need additional information at run time, about system's states and features of the relevant architectural elements, that would enable to associate that error with a mismatch. This ought to be done based on the particular error observed and on the presence of the preconditions required to activate it. The level of difficulty encountered on recovering from ECMs will very much depend on the individual error's characteristics and the architecture at hand. The treatment of faults that do cause ECMs will depend on whether the relevant features are style or application specific. It is our current belief that ECMs caused by style-specific features would require more fundamental changes to the system at hand, but this conjecture requires further study to be properly supported or contradicted.

3.2 Examples

In the rest of this section we briefly outline several simple examples demonstrating how our approach can be used.

First of all, let us consider a simple application-specific mismatch. Mismatch "sharing or transferring data with differing underlying representations" (mismatch 42 in [5] occurs when, for example, a component provides a value in feet and another component requires it in meters. In this case, "data transfer" is the conceptual feature used for ECM detection and recovery at the level of the application. The meta information that is required for detection and recovery concerns types of data to be transferred. Fault treatment may consist of replacing a connector with a new one that transforms the data.

The second example is that of application and style-specific mismatches. It happens, for example, when several components are connected in a system but only some of them can be backtracked (mismatch 28 from [5]). If any of such interconnected components backtracks, it has to inform all the components with which it interacts. To detect such a mismatch during run-time it is necessary to have additional information on the ability of each component to backtrack (conceptual feature "backtracking"), on the fact that backtracking is initiated by a component and on a set of interconnected components to be involved in backtracking. The detection of the ECM can be at the style level, but the recovery should be at the application level because, generally speaking, the application decides how to proceed with inconsistent data.

A style-specific mismatch happens, for example, when a non-reentrant component is called without waiting for the previous call to be completed (mismatch 24 from [5]). In this case we have a style-specific mismatch that can be detected and recovered if additional mechanisms are incorporated into the basic style (which can be based on the conceptual feature "re-entrance"). For example, an instantiation of a style should be able to detect any attempts to re-enter a process being executed. In the context of the pipe and filter architecture, it can happen that two filters try to access a single process in a third filter (see Figure 2). An extended style should provide means for detecting the ECM and for local recovery by either ignoring the second request or introducing additional concurrency control into the style (the simplest of which would be just delaying the second request).

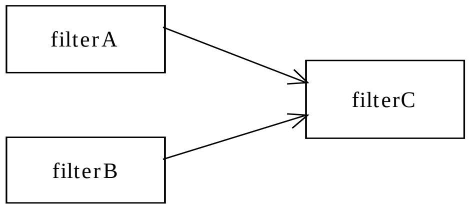


Figure 2. A non-reentrant component in a pipe-and-filter architecture

4. CONCLUSIONS

The problem of tolerating architectural mismatches during runtime can be summarised as follows. When an error caused by mismatch (ECM) is detected in the system, mechanisms and techniques have to recover the state of the system to an error free state, otherwise the erroneous state of the system can propagate, eventually leading to a system failure. However, the detection and recovery of an error is not enough for maintaining the integrity of the system services because if the mismatch, which has caused the detected error, is not treated, it can yet again be activated and be the cause of other errors. Similarly to fault tolerance in which one cannot develop techniques that can tolerate any possible faults, it is difficult to develop techniques that are able to deal with all types of architectural mismatches, hence assumptions have to be made about the types of mismatches that caused the errors to be detected and handled during runtime.

In this paper, we have mainly stated the problems and outlined a general approach to handling architectural mismatches during run time. Our preliminary analysis shows that a number of particular mismatch tolerance techniques can be developed depending on the application, architectural styles used, types of mismatches, redundancies available, etc. It is clear for us that there will always be situations when mismatches should be avoided or removed rather than tolerated. In our future work we will be addressing these issues, trying to define in a more rigorous way the applicability of the approach and to develop a set of general mismatch tolerance techniques. Some of the possible approaches are to modify how existing architectural styles are applied, to design a set of connectors capable of tolerating typical mismatches, to extend existing components and connectors with an ability to execute exception handling, and to develop a number of handlers that are specific for mismatches of different types.

ACKNOWLEDGMENTS

Alexander Romanovsky is supported by European IST DSoS project, and Cristina Gacek by the EPSRC (UK) DIRC project.

REFERENCES

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

[2] R. de Lemos, A. Romanovsky. "Exception Handling in the Software Lifecycle". *International Journal of Computer Systems Science & Engineering 16(2)*. March 2001. pp. 167-181.

[3] A. Egyed, N. Medvidovic, C. Gacek. "Component-Based Perspective on Software Mismatch Detection and Resolution". *IEE Proceedings on Software 147(6)*. December 2000. pp. 225-236.

[4] C. Gacek, A. Abd-Allah, B. Clark, and B. Boehm, "On the Definition of Software Architecture". Proceedings of the First International Workshop on Architectures for Software Systems – In Cooperation with the 17th International Conference on Software Engineering, D. Garlan (ed.), Seattle, WA, USA, 24-25 April 1995. pp. 85-95.

[5] C. Gacek. *Detecting Architectural Mismatches during System Composition*. PhD Dissertation. Center for Software Engineering. University of Southern California. Los Angeles, CA, USA. 1998.

[6] D. Garlan, R. Allen, J. Ockerbloom, "Architectural Mismatch: Why Reuse is so Hard". *IEEE Software 12(6)*. November 1995. pp. 17-26.

[7] J.-C. Laprie. "Dependable Computing: Concepts, Limits, Challenges". *Special Issue of the 25th International Symposium On Fault-Tolerant Computing*. IEEE Computer Society Press. Pasadena, CA. June 1995. pp. 42-54

[8] P. Oberndorf, K. Wallnau, A. M. Zaremski. "Product Lines: Reusing Architectural Assets within an Organisation. *Software Architecture in Practice*. Eds. L. Bass, P. Clements, R. Kazman. Addison-Wesley. 1998. pp. 331-344.

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

	Early Cycle 1	**End of Cycle 1**	**Cycle 2**	**Cycle 3**
Definition of operational concept and system requirements	Determination of top-level concept of operations	Determination of top-level concept of operations	Determination of detailed concept of operations	Determination of IOC requirements, growth vector
Definition of system and software architecture	System scope/ boundaries/ interfaces	System scope/ boundaries/ interfaces	Top-level HW, SW, human requirements	Choice of life-cycle architecture
Elaboration of software architecture	*No explicit architectural decision*	*Small number of candidate architectures described by architectural styles*	*Provisional choice of top-level information architecture*	*Some components of above TBD (low-risk and/or deferrable)*
Binding of architectural features	*No architectural features explicitly defined*	*Fixed architectural features that are defined by architectural styles, others are unknown*	*Architectural features defined by architectural styles are fixed as are some application specific ones, others are unknown*	*Most architectural features are fixed, the few unknown ones relate to parts of the architecture still to be defined*

Table 1. Refinement of software architecture under a Spiral Model Development

Architectural Prescriptions for Dependable Systems

Manuel Brandozzi
UT - ARISE
Advanced Research In Software Engineering
The University of Texas at Austin
mbrandozzi@ece.utexas.edu

Dewayne E. Perry
UT - ARISE
Advanced Research In Software Engineering
The University of Texas at Austin
perry@ece.utexas.edu

ABSTRACT
In this paper, we advocate the enforcement of dependability requirements at the architectural design level of a software system. We illustrate how to achieve this by using our methodology, which provides a guideline of how to design an architectural prescription from a goal oriented requirements specification of a system. We distinguish between separation, additive and integral non-functional requirements, and discuss their different effects on a prescription. In particular, additive non-functional requirements provide separation of concerns by only adding to the system some new components to achieve them. Dependability requirements are a particular kind of non-functional requirements and often they are additive.

1. INTRODUCTION
Experience has shown that, for complex software systems, it's very important to take into account the non-functional requirements as early as possible during the systems' design process. This means that they should be taken into account already at the architectural design level. By doing so, it's possible to understand what the implications of this kind of requirements are on the high level components and on the high level structure of the systems.

Dependability requirements are a particular type of non-functional requirements. In this paper we adopt a broad definition of dependability and we intend it to "embrace all those aspects of behavior upon which the user of a system might need to place dependence: it thus includes reliability, safety, availability and security" [3].

Another way that an architectural prescription favors the design of dependable systems is by enabling the reuse of the high level design of systems that, having been already deployed, have been demonstrated to be dependable. A prescription allows the architect to reuse all the components and the topology that derive from particular goals (i.e. requirements), including dependability requirements. Generally, a brand new system design has a higher likelihood of failure than a well tested one.

Before illustrating our approach for dependability enforcement at the architectural level in section 4, we provide an introduction to goal oriented requirements and to architectural prescriptions in sections 2 and 3 respectively. In section 5 we illustrate our approach with an example and we summarize our contributions and discuss future work in section 6.

2. GOAL ORIENTED REQUIREMENTS SPECIFICATIONS AND KAOS

Goal oriented requirements specifications are, among all the kinds of requirements specifications, those that are closer to the way humans think and hence easier to understand by all the stakeholders in the development process. KAOS is the goal oriented specification language, introduced by A. van Lamsweerde [2], that we used in our methodology.

The KAOS' ontology is composed of objects, operations and goals. Objects can be agents (active objects), entities (passive objects), events (instantaneous objects), and relationships (objects depending on other objects). Operations are performed by an agent and change the state of one or more objects. Operations are characterized by pre-, post- and trigger- conditions.
Goals are the objectives that the system has to achieve. In general, a goal can be AND/OR refined till we obtain a set of achievable sub-goals. The goal refinement process generates a goal refinement tree. All the nodes of the tree represent goals. The leaves of the tree may also be called requisites. The requisites that are assigned to the software system are called requirements; those assigned to the interacting environment are called assumptions. Here is an example of goal declaration in KAOS:

Goal Maintain[AuthorizedAccessesOnly]
InstanceOf SecurityGoal
Concerns StockValues, BankerActor
ReducedTo
    ConfidentialityOfAccessPassword,
    ConfidentialityOfTransmittedStockValues
InformalDef
    Access passwords must remain confidential. Stock values information has to be released only to those providing the correct passwords.

Example 1: a goal specification in KAOS.

The keyword *Goal* denotes the name of the goal; *InstanceOf* declares the type of the goal; *Concerns* indicates the objects involved in the achievement of the goal; *ReducedTo* contains the names of the sub-goals into which the goal is refined. Finally, there is *InformalDef*: the informal definition of the goal. There can also be an optional attribute *FormalDef*, which contains a formal definition of the goal (that can be expressed in any formal notation such as linear temporal logic).

3. ARCHITECTURAL PRESCRIPTIONS AND PRESCRIPTOR

An architectural prescription [1] lays out the space for the system structure by selecting the architectural components (processes, data, and connectors), their relationships (interactions) and their constraints. In a prescription, the fundamental characterization of components is given by the goals they are responsible for (that are their constraints). Components are further characterized by their type: processing, data or connector. The processing components are those that provide the transformation on the data components. The data components contain the information to be used and transformed. The connector components, which may be either implemented by data components, processing components or a combination of both, are the glue that holds all the pieces of the system together. The interactions of the components among each other, together with the restriction of their possible number of instances characterize the topology of the system.

Example 2 contains the architecture prescription of a data component specified in Preskriptor, our architectural prescription language that takes KAOS requirements specifications as starting point.

Component StockValues *[1, 1]*
Type Data
Constraints Maintain[LatestStockValuesInfo], …
Composed of DB *[1,1]*, Server *[1,1]*
Uses MarketConnect *to interact with*
StockMarket

Example 2: a component's specification in Preskriptor.

The field *Component* specifies the name of the component. *Type* denotes the type of the component. *Constraints* is the most important attribute of a component. It denotes which are the requirements that the component is responsible for. Note that the semantics of any component in Preskriptor is provided by its constraints and only by them. We use here the term constraint to denote both functional and non-functional constraints (both corresponding to requirements on the system). *Composed of* identifies the subcomponents that implement the component. The last attribute, *Uses*, indicates what are the components used by the component. Since interactions can only happen through a connector, the *Uses* attribute has the additional keyword *to interact with* that indicates which components the component interacts with using a particular connector. The symbol "/" means no attribute and, since now, we will omit the fields whose value is none.

Without going into the details of how to get a prescription from the requirements [1], it's important to know that at the beginning some candidate components for the architecture are proposed, then the functional goals first, and non-functional goals afterwards, are assigned, one at a time, to a subset of the potential components. Those components who do not contribute to the achievement of any goal are discarded from the system. The next section explains in some detail how to account for non-functional requirements in an architectural prescription.

4. NON-FUNCTIONAL REQUIREMENTS IN PRESCRIPTION DESIGN

Taking into account Non-Functional Requirements (NFRs) while designing an architectural prescription has, in the most general case, three kinds of effects on the already designed prescription of a system:

1) The introduction of new components.
2) The transformation of the system's topology, i.e. a change on the relationships among the system's components.
3) The further constraining of already existing components.

Some non-functional requirements allow for separations of concerns among the architecture components; other requirements, instead, are spread throughout the code: they reach every component of the system like blood vessels reach every cell of our body.

We denote those NFR that enable separation of concerns with respect to an architecture as *Separation Non-Functional Requirements* (SNFR). SNFR are those requirements that can be achieved by further constraining, adding new components and/or by transforming the topology of only a precisely identifiable subset in strict sense of the architecture's components. By "precisely identifiable" subsystem we mean that the subsystem can be characterized by a property. By subsystem "in strict sense" we exclude the complete system, case in which we don't achieve separation of concerns. A precisely identifiable subsystem in strict sense can be, for example, a single component of the system. This happens in the case of a performance goal when a single component is the bottleneck for computation. Another example of a precisely identifiable subsystem is the set including all the connectors from a particular component, and the component itself, like in the fault tolerance example that we will illustrate in next section.

The simplest SNFRs are those that in some architectures can be achieved by only adding to the system new components and the relationships of these new components with other components, i.e. by composing some existing components with new ones without changing the constraints of any of the old components. We denote this kind of NFRs as *Additive Non-Functional Requirements* (ANFRs).

Those NFRs that are not SNFRs are denoted as Integral Non-Functional Requirements (INFRs). These requirements affect the entire system or a subset of the system for which no characterizing property can be found, i.e. the system is not precisely identifiable. A way to achieve this other kind of requirements is by making all components conform to a particular style. An example of INFR is the goal for a system is to be composed by only components that conform to CORBA. No matter what, this requirement has to be added as a constraint to all the system's components.

In general, whether an NFR is integral, separation, or additive depends on the architecture on which we want to achieve it. It also may depend on the level of refinement of the architecture. In fact, what at a finer resolution of an architecture is a clearly identifiable subset in strict sense may become the whole set of the system's components at a coarser refinement.

5. ENFORCING DEPENDABILITY AT THE ARCHITECTURAL LEVEL

Let's see, with the aid of an example, how a dependability requirement (in this case fault tolerance) can be handled by an architectural prescription.

Any computer network may have, even in absence of catastrophic events, a certain number of machines that crash or become inaccessible. Let's consider the case of a distributed system, that runs on such a network, and that contains a data component whose accessibility at any time is vital. The data component can, for example, contain the value of the stocks managed by an investment bank. It's vital that the bankers can access at any time the current value of a stock. Not being possible to do so could cost to the bank thousands of dollars, if not millions!

This kind of fault tolerance problem has been widely studied in the distributed systems community and a standard solution to it is the following. Suppose that in a network with x nodes there can be at most t (with $t < x$) nodes that can fail at the same time. We can achieve a fault-tolerant real-time access to the vital data object StockValues, by having, at least, t+1 copies of the object stored in t+1 different nodes. To guarantee this we also need some protocol that manages the access to the object from outside the network, and that updates of the copies of the object to achieve consistency among them.

Example 3. contains the prescription of a simple distributed system. This is the prescription of the system before we take into account the fault tolerance goal.

Component StockValues *[1, 1]*
Type Data
Constraints Maintain[LatestStockValuesInfo], …
Composed of DB[1,1], Server[1,1]
Uses MarketConnect *to interact with* StockMarket

Component BankerClient *[0, n]*
Type Processing
Constraints …
Uses
StockValuesAccess *to interact with* V
BankerUserInterface *to interact with*
BankerActor

Component StockValuesAccess *[0, n]*
Type Connecting
Constraints Maintain[AuthorizedAccessesOnly]

Example 3: prescription for a stock values information system

In the prescription of example 3 we have only one copy of the data component *StockValues*, the component storing the latest values of the stocks belonging to different markets that is updated by using *MarketConnect* connecting it to the stock markets. The prescription allows any number n of components *BankerClient* to be instantiated. *BankerClient* is the piece of software running on the bankers' machines.

The prescription requires the system to have communication between *BankerClient* and *StockValues* through connector *StockValuesAccess*, that has to achieve the security goal *Maintain[AuhorizesAccessesOnly]* (defined in Example 1.) together with other goals (such as mutual exclusion) not included here for simplicity. Given a particular choice of implementation of connector StockValuesAccess, the low level design may instantiate the connector only once for all the n Clients, instantiate it n times, or any other number of times between 0 and n.

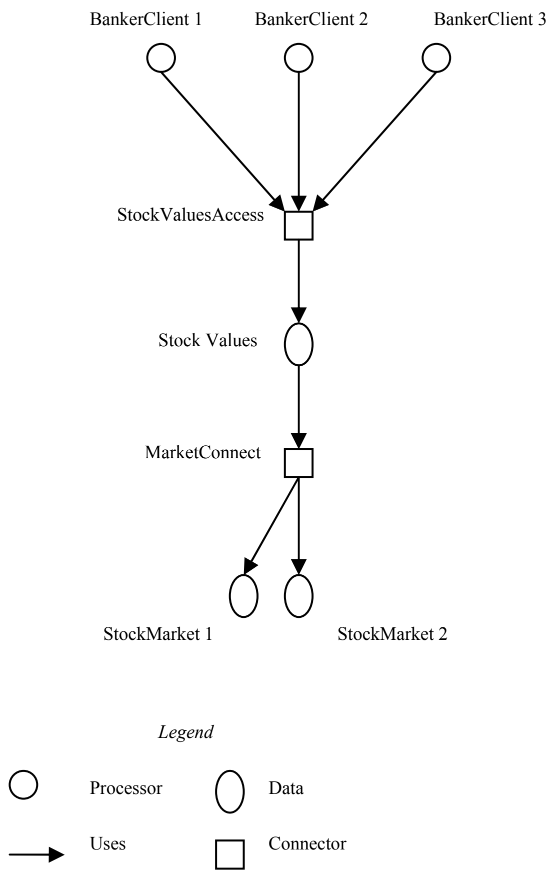


Figure 1. Topology graph of the prescription in example 3.

Figure 1. contains the graphical representation of the topology of an instantiation of the same prescription. It's the topology of an instantiation of the prescription because for each component a given number of instances has been chosen. For example, there are only three clients, rather than an indefinite number; and there is only one instance of connector *StockValuesAccess* for the interaction of all the clients with *StockValues,* rather than an indefinite number of connectors that is at most equal to the number of *BankerClient* components. Graphical representations are useful to better understand the topology that is specified by the prescription.

In any prescription graph, the arrow representing the *Uses* attribute goes from the component C, that needs some information from the interaction, to a connector CN that makes the interaction possible; and then from the connector CN to those components that provide the information needed by C.

Example 4. shows the same prescription after it has been transformed to account for the non-functional goal of fault tolerance of the component StockValues.

Component StockValues *[t+1, n]*
Type Data
Constraints Maintain[LatestStockValuesInfo], …
Composed of DB[1,1], Server[1,1]
Uses
    MarketConnect *to interact with* StockMarket
    InterCopyCoordinator *to interact with*
        StockValues

Component InterCopyCoordinator *[1, n]*
Type Connecting
Constraints Maintain[FaultTolerance]

Component StockValuesAccess *[0, n]*
Type Connecting
Constraints Maintain[AuthorizedAccessesOnly]

Component StockValueFTAccess *[1,n]*
Type Connecting
Constraints
    Maintain[FaultTolerance],
    Maintain[AuthorizedAccessesOnly]
Composed of
    IntercopyCoordinator *[1,n]*, StockValuesAccess *[0,n]*

Component BankerClient *[0, n]*
Type Processing
Constraints …
Uses
    StockValuesFTAccess *to interact with*
        StockValues
    BankerUserInterface *to interact with*
        BankerActor

Example 4: prescription for a stock values information system with fault tolerance

Like many dependability requirements, the non-functional fault tolerance requirement is an ANFR. It's an ANFR because it can be assigned as a constraint only to the new connector *InterCopyCoordinator,* which coordinates the now multiple copies of component *StockValues*. This is an example of achieving an ANFR via connectors; another such example can be found in a system developed by the DSSA group [5], case in which the NFR is performance.

The system specified by the new version of the prescription has to have at least t+1 (t being the maximum number of faults) copies of component *StockValues*, rather than only one like in its pre-fault tolerance prescription. *StockValues* is now using the newly added connector *InterCopyCoordinator*. To achieve the fault tolerance goal, among the other things, this connector will have to make it sure that, at any time, there are at least t+1 copies of *StockValues*. Also, it has to assure that the different copies are, somehow, kept consistent at least from the perspective of the rest of the software system. The *BankerClients* interacting with component *StockValues* must always get the latest value for the stocks. The access to *StockValues* by two or more clients at the same time has to abide to the same mutual exclusion policies that held when only an instance of *StockValues* was in the system. We designed the prescription so that *InterCopyCoordinator* keeps the topological transformations transparent to *BankerClient*. The only change in *BankerClient*'s specification is that now this component uses connector *StockFTValuesAccess* (resulting from the composition of *InterCopyCoordinator* and *StockValuesAccess*) to interact with *StockValues*, rather than using *StockValuesAccess*. It's the *InterCopyCoordinator*'s subcomponent of *StockFTValuesAccess* that hooks into *StockValues* to guarantee that *BankerClient* always gets the updated values of the stocks.

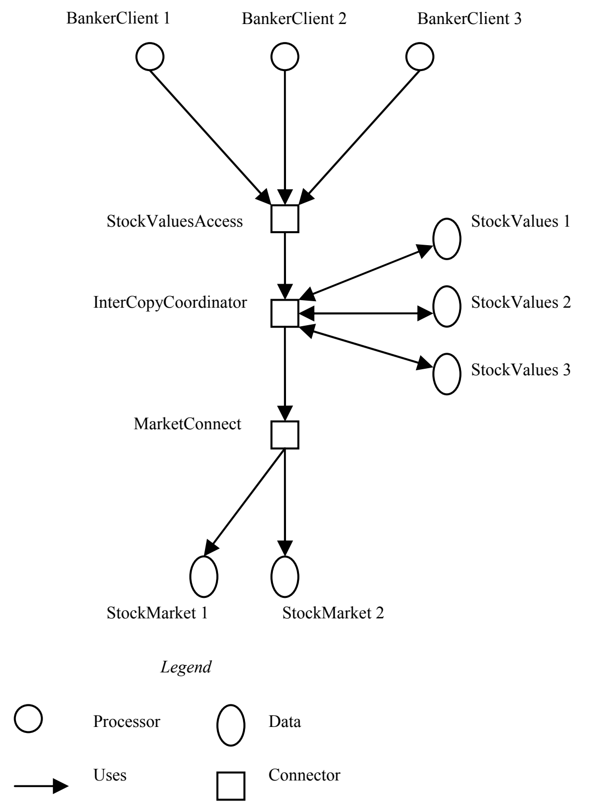


Figure 2. Topology graph of the prescription in example 4.

In a particular implementation of the prescription in a latter phase of the development process, *InterCopyCoordinator* may take care of the creation of t+1 copies at start-up, as well as creating new copies, moving the existing ones, or remove copies whenever some node fails or to save on communication costs like in illustrated in [4].

The effects of the topological transformation are evident if we have a look the topology graph of the new prescription in figure 2. Here, the graph is the same than the one of figure 1. apart from having substituted the single component *StockValues* with a more complex component that is composed by the different *StockValues* instances (three in the example) and the *IntercopyCoordinator* used by them. The double edged arrows are a syntactic shortcut to make the graph more elegant: they represent all the arrows that depart from and go to a particular component.

6. CONCLUSION

Dependability requirements are a large subset of non-functional requirements. To better achieve them and manage their changes they should be taken into account already at the architectural design level. We provided an overview of our methodology to design an architectural prescription given a set of goal oriented requirements specifications.

The requirements can either be functional or non-functional. Separation Non-Functional Requirements (SNFRs) enable separation of concerns in achieving them. Their effects are limited to a subset of the system identifiable by a property (like the set of connectors outgoing from a particular component). In particular, we illustrated with an example how a fault tolerance requirement for an object in a network (that is an ANFR, an easier case of SNFR) can be achieved by a given architecture. This was done by introducing in the architecture a new connector and modifying the topology of the system locally to one of its components. Many other dependability requirements, including security, performance and other kinds of fault tolerance can be ANFRs with respect to many architectures.

Our future work will be aimed at finding out domain independent ways to compositionally transform the prescription of a system to account for ANFRs and at developing a tool to do so automatically.

5. REFERENCES

[1] Brandozzi, M., and Perry, D. Transforming Goal Oriented requirements specifications into Architectural Prescriptions. Proceedings STRAW '01, ICSE 2001, Toronto, May 2001, 54-61

[2] Van Lamweerde, A., Darimont, R., and Massonet, P. Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt. Proceedings RE'95 – 2nd IEEE Symposium on Requirements Engineering, York, March 1995, 194-203

[3] Littlewood, B. Evaluation of software dependability. Computing Tomorrow: Future Research Directions in Computer Science, Book, I. Wand and R. Milner, 1996

[4] Johnson, G., Singh, A. Stable and fault-tolerant object allocation. Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon 259-268

[5] Tracz, W. Domain-Specific Software Architecture Pedagogical Example. ACM Software Engineering Notes, July 1995, 49-62.

Integration of Architecture Specification, Testing and Dependability Analysis

Swapna S. Gokhale, Joseph R. Horgan
Telcordia Technologies
445 South Street
Morristown, NJ, 07960, USA
swapna,jrh@research.telcordia.com

Kishor S. Trivedi
Dept. of ECE, CACC
Duke University
Durham, NC 27708, USA
kst@ee.duke.edu

ABSTRACT

Software architectural choices have a profound influence on the quality attributes supported by a system. Architecture analysis can be used to evaluate the influence of the design decisions on important quality attributes such as maintainability, performance, and dependability. As software architecture gains appreciation as a critical design level for software systems, techniques and tools to support testing, understanding, debugging, and maintaining these architectures are expected to become readily available. In addition to providing the desired support, data collected from these tools also provides a rich source of information from the point of view of performance and dependability analysis of the architecture. The present paper presents a performance and dependability analysis methodology which illustrates the use of such data. The methodology thus seeks a three way integration of distinct and important areas, namely, formal specification, specification simulation/testing and dependability/performance analysis. We illustrate the important steps in the methodology with the help of an example.

1. INTRODUCTION

As software systems [1] continue to grow in size and complexity, software architecture is increasingly appreciated as a method of understanding and analysis. The architecture of a software system defines its high level structure, exposing its gross organization as a collection of interacting components [4]. Software architecture represents the design decisions that are made in the early phases of a system. These decisions are usually difficult to change or reverse and have a profound influence on the non functional attributes that can be supported by the system. It is becoming increasingly clear that software architecture analysis is the best vehicle to assess important quality attributes such as maintainability, reliability, reusability and performance. Architectural description languages (ADLs) are formal languages that can be used to represent the architecture of a software system. They focus on the high-level structure of the overall system rather than the implementation details of any specific source module. ADLs are intended to play an important role in the development of software by composing source modules rather than by composing individual statements written in conventional programming languages. A number of architectural description languages have been proposed recently, such as Rapide [9], UniCon [13] and WRIGHT [1]. As software architecture design gains prominence, the development of techniques and tools to support understanding, testing, debugging, reengineering, and maintaining software architecture will become an important issue. Li et. al. propose a tool for understanding, testing, debugging and profiling software architectural specifications in SDL [8]. Zhao et. al. propose a technique for architecture understanding based on program dependence graphs for ACME [19]. They also propose a slicing methodology to extract reusable software architectures [20].

[1]System and application are used interchangeably in this paper.

Software architectures specified in ADLs such as SDL and LOTOS, and other modeling languages such as UML can also be used for performance and dependability analysis, by constructing quantitative models from these specifications. Wohlin et. al. develop a performance analysis methodology for specifications written in SDL [18]. Marsan et. al. present an extension of LOTOS, which enables a mapping from the extended specifications to performance models [10]. Steppler develop a tool for the simulation and emulation of formation specifications in SDL, with an eye towards analyzing these specifications for the non functional attribute of performance [14]. Heck et. al. describe a hierarchical performance evaluation approach for formally specified communication protocols in SDL [6]. Bondavalli et. al. present a dependability analysis procedure based on UML designs [2]. Wang et. al. discuss a performance and dependability analysis mechanism for specifications in Estelle [17]. The major drawback of the analysis approaches mentioned above is the lack of adequate information to parameterize the quantitative models constructed from the software specifications. As the techniques for software architecture testing become mature and tools become readily available, the data generated during testing can provide a rich source of information for subsequent model parameterization. A similar approach has been demonstrated at the source code level, where execution trace data collected from extensive testing

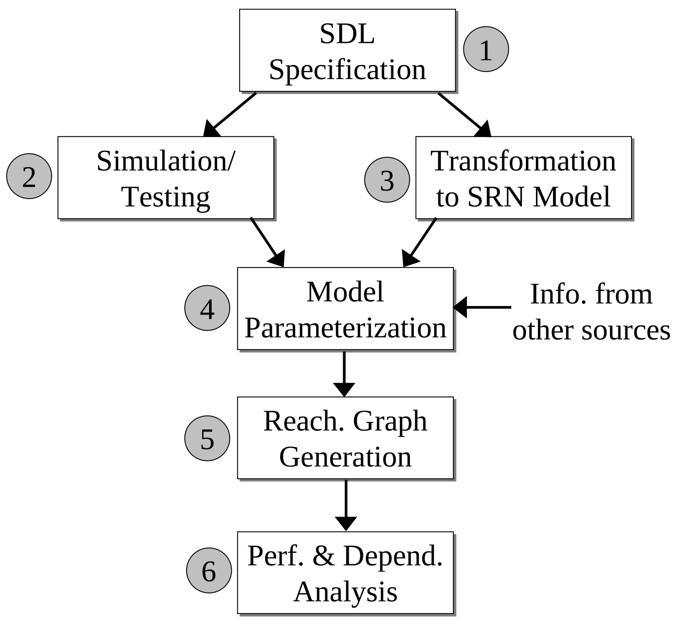


Figure 1: Steps in the methodology

of the application was used to extract and parameterize the architectural model of the system [5]. Our present paper outlines an approach which seeks a three way integration, namely, formal specification/modeling, architecture simulation/testing and performance/dependability analysis. The glue between specification, performance analysis and testing is provided by using the measurements obtained during simulation/testing to parameterize the quantitative model of the system. Our methodology is facilitated by Telcordia Software Visualization and Analysis Tool Suite (TSVAT), developed at Telcordia Technologies for architectural specifications in SDL [8].

We describe the steps involved in the analysis methodology in the extended abstract. The illustration of the methodology to assess the performance and dependability of an application specified in SDL will be presented at the workshop.

2. METHODOLOGY

The methodology centers around a system specified using the Specification and Description Language (SDL) and the quantitative modeling paradigm of Stochastic Reward Nets (SRNs). The steps involved in the methodology can be depicted pictorially as shown in Figure 1. We briefly describe the steps shown in Figure 1 in this section.

2.1 System Specification in SDL

An architecture can be specified using an informal notation (box and arrow diagrams). However, such notations are error-prone and ambiguous. An architecture specified in a formal specification language eliminates the ambiguity and provides a clear basis for analysis. We choose Specification and Description Language (SDL) as a Communicating Extended Finite State Machine (CEFSM) specification language to specify the software architecture. The choice of SDL as an architecture description language is motivated due to the following reasons: i) SDL is an International Telecommunication Union (ITU) standard [15]. Many telecom system software specifications are rendered in SDL, ii) SDL is a formal language with a well-defined semantics. Several commercial off-the-shelf tools can be used to investigate architectural models formalized in SDL, iii) SDL meets the requirements for an executable architectural specification language [9]. SDL allows dynamic creation and termination of process instances and their corresponding communication paths during execution. Hence it is capable of modeling the architectures of dynamic systems in which the number of components and connectors may vary during system execution, iv) SDL can present all four views of software architecture [7]. For example, SDL uses delay and non-delay channels to indicate the relative physical locations of components. The first step in the methodology is the specification of the application to be assessed in SDL [2]. This step is marked as "1" in Figure 1.

2.2 Specification Simulation/Testing

The next step in the methodology (marked "2" in Figure 1) is to simulate/test the system specified in SDL and collect trace data during the simulation process. We use the SDL version of the Telcordia Software Visualization and Analysis Tool Suite (TSVAT) [8], developed to support architectural specification, debugging and testing to collect the trace data. TSVAT contains a suite of tools, χSlice, χVue, χProf, χRegress, and χATAC. We primarily use χATAC in our methodology. The technique underlying this tool suite is the creation of a flow graph of the specification, thus laying out its execution structure. We use the simulator from Telelogic [21] to simulate the SDL specification of the application. The simulator is then instrumented to collect execution traces. The trace file records how many times a given part of the specification, such as a process, a transition, a decision, a state input, or a data flow, has been exercised in each simulation of the specification, or at the end of the testing process. χATAC reports coverage with respect to the following well-known criteria: function coverage, basic transition coverage and decision coverage. Function coverage simply checks that each process of the SDL specification has been executed at least once. A basic transition is simply a statement sequence of the specification that is always executed sequentially, including states and decision constructs (no internal branching constructs). Basic transition coverage checks that each basic transition has been executed at least once, which implies that each statement has been executed at least once. Decisions are conditional branches from one basic transition to another. Decision coverage checks that each such situation, decision matching or input match-

[2]It should be noted that our methodology which proposes a three way integration of architecture specification, simulation/testing and dependability analysis is not limited to the use of SDL. Any architecture specification language which provides capabilities similar to SDL can be used in this methodology.

ing is executed, so that all true and false paths have been taken as well as all input alternatives and decision alternatives.

The execution traces collected during the testing of the specification can then be used to extract branching probabilities of the various decisions in the specification. If the simulation process of the specification is guided by an operational profile [11] of the system, then these branching probabilities would be characteristic of what would be observed in the field. Regression test suites from earlier releases of the product and/or expert opinion could also be used to guide the design of test cases for simulation.

2.3 Translation from SDL Specification to a SRN Model

The SDL representation of the architecture of the system is then translated to a stochastic reward net (SRN) model for performance and dependability analysis. This step is marked "3" in Figure 1. Stochastic reward nets (SRNs) are a generalization of generalized stochastic Petri Nets (SPNs), which in turn are a generalization of stochastic Petri Nets (SPNs) [16]. Stochastic Reward Nets (SRNs) provide the same capabilities as Markov Reward Models [12] which can be used to compute various reliability, safety and performance measures of interest. We define rules to translate an SDL specification at the process level to a stochastic reward net (SRN) model.

2.4 SRN Model Parameterization

The next step in the process is to parameterize the SRN model. This step is marked "4" in Figure 1. The parameters of the SRN model can be broadly classified into five categories, depending on the sources of information used for the parameterization:

- Execution time parameters: These parameters are associated with the execution of the tasks and decisions in the SDL specification, and are heavily dependent on the implementation details. SDL specifications can also be used to generate code in a semi-automatic fashion, and the measurements obtained from the execution of this partial code can be used to determine the distributions and the values of these parameters.
- User inputs: These parameters model the inputs resulting from the actions of the user. These inputs are expected by the system at various stages of execution. The distributions and the actual values of these parameters can be derived from historical data, or can be based on the knowledge of the experts who are intimately familiar with the system and its field usage. In the event that the system is the first of its kind and not much information is available about the system, these parameters can be guestimated.
- Branching probabilities: These reflect the probabilities of occurrence of the various outcomes of a decision. These values can also be derived from historical data, or can be based on expert knowledge. The trace data collected during the simulation/testing of the SDL specification can be used to determine these branching probabilities. The SDL version of Telcordia Software Visualization and Analysis Tool Suite (TS-VAT) facilitates the collection of such trace data.
- Inputs from other components/processes: Most real life software systems are inherently distributed, and hence require interactions between the various components of the system. Since the SRN model is constructed from process level specification, it is natural for some of the parameters of the model to draw from the execution of the other processes in the system.
- Failure/repair parameters: These parameters model the failure/repair behavior of the processes and/or each task within a process and are necessary to compute various measures such as the reliability, availability and safety of an application. This information can be obtained by consulting with experts who are familiar with the application or can be guestimated.

2.5 Reachability Graph Generation

The next step in the analysis methodology is the generation of a reachability graph, and the step is marked "5" in Figure 1. The reachability graph of a Petri net is the set of states that are reachable from the other states. The reachability graph is generated using the tool SPNP (Stochastic Petri Net Package) [3], which is also used for performance and dependability analysis. The Stochastic Petri Net Package is a versatile modeling tool for the solution of stochastic Petri net (SPN) models . The SPN models are described in the input language for SPNP called CSPL (C-based SPN language). The CSPL is an extension of the C programming language with additional constructs which facilitate easy description of SPN models. The SPN models specified to SPNP are actually "SPN Reward Models" or Stochastic Reward Nets (SRNs), which are based on the "Markov Reward Models" paradigm. This provides a powerful modeling environment for dependability (reliability, availability, safety), performance and performability analysis.

2.6 Dependability Analysis

The parameterized stochastic reward net (SRN) model of the application can then be used for performance and dependability analysis. The Stochastic Petri Net Package (SPNP) can be used to compute various measures of interest such as the reliability, availability and safety [3]. The performance and dependability analysis step is marked "6" in Figure 1.

3. CASE STUDY

We illustrate the steps described in the previous section with the help of a PBX system specified in SDL. Figure 2 shows the block level specification of the PBX system in SDL.

The PBX system consists of two distributed blocks. The block *CallHandler* controls the call processing functions and the block *ResManager* involves inventory control and remote database access. The *CallHandler* block receives three signals over channel *c*1, namely, *offhook*, *dig* and *hangUp*. The first signal announces the caller's intent to place a call, the second signal is a digit of a telephone number and the hangUp signal is sent either after the call is complete, if there is a change in the caller's intention to pursue the call, due to the unavailability of the resources required to place a call, or due to the unavailability of the callee to accept the call (callee is busy). The *CallHandler* block outputs the signals *dialT*, *busyT*, *connt*, *ringT* and *ring* over the channel *c*2. Communication between the *CallHandler* block and the *ResManager* block occurs over channels *c*3 and *c*4. Channel *c*3 is used to communicate the request and release messages for resources and channel *c*4 is used to send reply messages regarding the availability of the resources to proceed with the call. Channels *c*3 and *c*4 are delaying channels which indicates that the two blocks can be implemented on different CPUs with a non-negligible delay. This reflects the reality that the database information can be stored remotely. The process level specification of the *CallHandler* and *ResManager* blocks as well as the other steps in the methodology will be presented at the workshop.

4. CONCLUSIONS AND FUTURE

In this paper we present a methodology which integrates three distinct areas in architecture design: specification, simulation/testing, and performance/dependability analysis. Our future research includes extending the methodology to analyze other non functional attributes such as maintainability and flexibility.

5. REFERENCES

[1] R. Allen. *"A formal approach to software architecture"*. PhD thesis, Dept. of Computer Science, Carneige Mellon University, Pittsburgh, NC, 1997.

[2] A. Bondavalli, I. Mura, and I. Majzik. "Automated dependability analysis of UML designs". In *Proc. of Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 1998.

[3] G. Ciardo, J. Muppala, and K. S. Trivedi. "SPNP: Stochastic Petri Net Package". In *Proceedings of the International Workshop on Petri Nets and Performance Models*, pages 142–150, Los Alamitos, CA, December 1989. IEEE Computer Society Press.

[4] D. Garlan and M. Shaw. *Advances in Software Engineering and Knowledge Engineering, Volume 1, edited by V. Ambriola and G. Torotora*, chapter An Introduction to Software Architecture. World Scientific Publishing Company, New Jersey, 1993.

[5] S. Gokhale, W. E. Wong, K. S. Trivedi, and J. R. Horgan. "An analytic approach to architecture-based software reliability prediction". In *Proc. of Intl. Performance and Dependability Symposium (IPDS '98)*, pages 13–22, Durham, NC, September 1998.

[6] E. Heck and D. Hogrefe. "Hierarchical performance evaluation based on formally specified communication protocols". *IEEE Trans. on Computers*, 40(4):500–513, April 1991.

[7] P. B. Krutchen. "The 4+1 view model of architecture". *IEEE Software*, pages 42–50, November 1995.

[8] J. J. Li and J. R. Horgan. "A toolsuite for diagnosis and testing software design specifications". In *Proc. of Dependable Systems and Networks (DSN2000, FTCS31)*, New York, NY, June 2000.

[9] D. Luckham, L. A. Augustin, J. Kenney, J. Veera, D. Bryan, and W. Mann. "Specification and analysis of system architecture using rapide". *IEEE Tran. on Software Engineering*, 21(4):336–355, April 1995.

[10] M. A. Marsan, A. Bianco, L. Ciminera, R. Sisto, and A. Valenzano. "A LOTOS extsnsion for the performance analysis of distributed systems". *IEEE/ACM Transactions on Networking*, 2(2):151–165, April 1994.

[11] J. D. Musa. "Operational profiles in software-reliability engineering". *IEEE Software*, 10(2):14–32, March 1993.

[12] A. Reibman, R. Smith, and K. S. Trivedi. "Markov and Markov Reward Model Transient Analysis: An Overview of Numerical Approaches". *European Journal of Operational Research*, 40:257–267, 1989.

[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 Trans. on Software Engineering*, 21(4):314–335, April 1995.

[14] M. Steppler and B. Walke. "Performance analysis of communications systems formally specified in SDL". In *Proc. of the Workshop on Software and Performance*, Santa Fe, NM, 1998.

[15] International Telegraph and Telephone Consulative Committee. *"SDL User Guideliness"*. International Telecommunication Union, November 1989.

[16] L. A. Tomek and K. S. Trivedi. *Software Fault Tolerance, Edited by M. R. Lyu*, chapter Analyses Using Stochastic Reward Nets, pages 139–165. John Wiley and Sons Ltd., New York, 1995.

[17] C. Y. Wang and K. S. Trivedi. "Integration of specification for modeling and specification for system design". In *Proc. of Fourteenth Intl. Conference on Applications and Theory of Petri Nets*, pages 24–31, 1993.

[18] C. Wohlin and D. Rapp. "Performance analysis in the early design of software". In *Proc. of Intl. Conference on Software Engineering for Telephone Switching Systems"*, pages 114–121, 1992.

[19] J. Zhao. *New Technologies on Computer Software, M. Li, Editor*, chapter "Using Dependence Analysis to Support Software Architecture Understanding", pages 135–142. International Academic Publishers, September 1997.

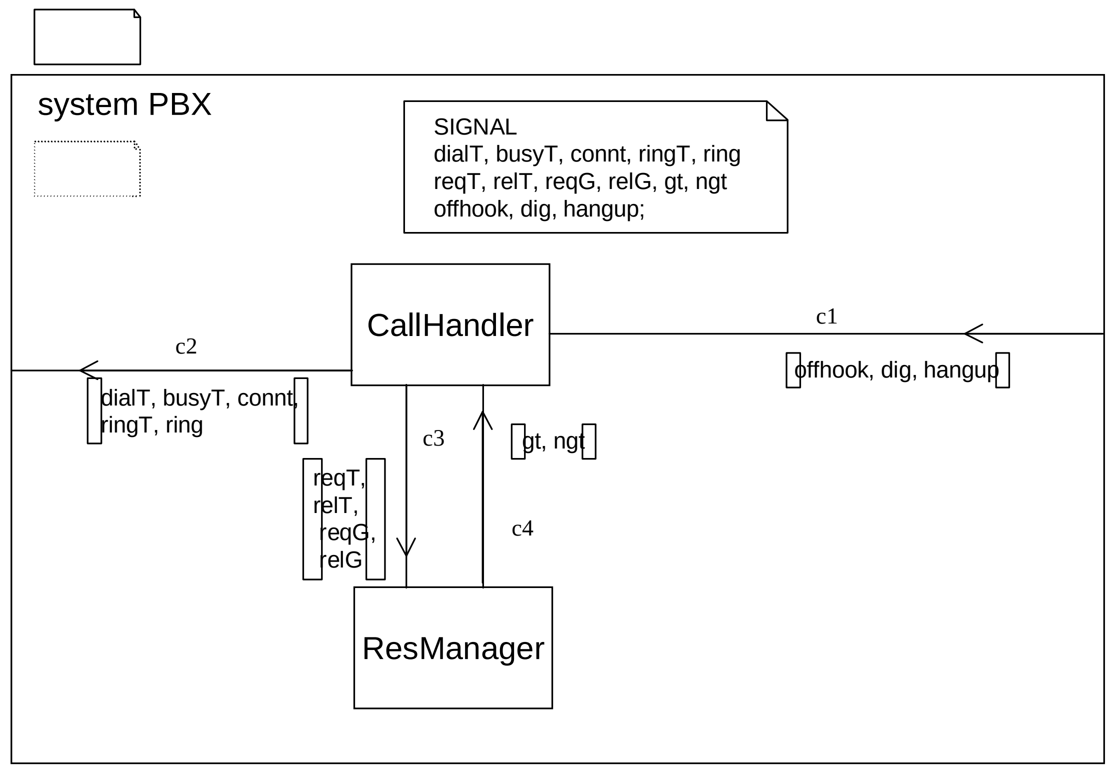


Figure 2: Block level SDL specification of a PBX system

[20] J. Zhao. "A slicing-based approach to extracting reusable software architectures". In *Proc. of 4th European Conference on Software Maintenance and Reengineering*, pages 215–233, Zurich, Switzerland, February 2000.

[21] *http://www.telelogic.com.*

The Role of Event Description in Architecting Dependable Systems

Marcio S Dias Debra J Richardson
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
{mdias,djr}@ics.uci.edu

ABSTRACT

Software monitoring is a well-suited technique to support the development of dependable systems, and has been widely applied not only for this purpose, but also for others such as debugging, security, performance, etc. Software monitoring consists of observing the dynamic behavior of programs when executed, by detecting particular events and states of interest, and analyzing this information for specific purposes.

There is an inherent gap between the levels of abstraction the information is collected (the implementation level) and the software architecture level. Unless there is an immediate one-to-one architecture to implementation mapping, we need a specification language to describe how low-level events are related to higher-level ones. Although some event specification languages for monitoring have been proposed in the literature, they do not provide support up to the software architecture level.

In this paper, we discuss the importance of event description as an integration element for architecting dependable systems. We also present how our current work in defining an interchangeable description language for events can support the development of such complex systems.

1. INTRODUCTION

As stated in the workshop call, "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 software architecture level of abstraction helps the developer in dealing with system complexity, and is the adequate level for analysis, since components, connectors, and their configuration are better understood and intellectually tractable [16].

When building dependable systems, additional management services are required and they impose even more complexity to the system [14]. Some of these services are fault-tolerance [5] and safety, as well as security (intrusion detection) and resource management, among others. An underlying service to all these services is the software monitoring.

Software monitoring is a well-known technique for observing and understanding the dynamic behavior of programs when executed, and can provide for many different *purposes* [13][15]. Besides dependability, other purposes for applying monitoring are: testing, debugging, correctness checking, performance evaluation and enhancement, security, control, program understanding and visualization, ubiquitous user interaction and dynamic documentation.

Software monitoring consists in *collecting* information from the system execution, *detecting* particular events or states using the collected data, *analyzing* and *presenting* relevant information to the user, and possibly taking some (preventive or corrective) *actions*. As the information is collected from the execution of the program implementation, there is an inherent gap between the level of abstraction of the collected events (and states) and of the software architecture. Unless the implementation was generated from the software architectural description, or there is an easily identifiable one-to-one architecture to implementation mapping [1][10][16], we need to describe how those (primitive) events are related to higher-level (composed) events.

Many monitoring systems were developed so the user could specify composed events from primitive ones, using provided specification languages. However, in general, these specification languages are either restricted to a single monitoring system, not generic for many different purposes, or cannot associate specified events to the software architecture.

There is no monitoring system able to provide for all different purposes. One problem occurs when a user is interested in applying monitoring for more than one purpose (for instance, dependability, performance evaluation, and program visualization). In this case, he or she would probably run different monitoring systems and, consequently, need to describe the same events multiple times using different specification languages.

To put it simple, software monitoring is a well-suited technique to support the development of dependable systems, and has been widely applied for this purpose. However, monitoring systems suffer in the ability to associate collected information to software architecture level.

In this position paper, we discuss how software monitoring can be applied at the software architectural level to support dependability. In this context, we present some requirements for event description languages, and our ongoing work on xMonEve, an XML-based language for describing monitoring events.

2. EVENT MONITORING

There are basically two types of monitoring systems based on the information collection: *sampling* (time-driven) and *tracing* (event-driven). By sampling, information about the execution *state* is collected synchronously (in a specific time rate), or asynchronously (through direct request of the monitoring system). By tracing, on the other hand, information is collected when an *event* of interest occurs in the system [11].

Tracing allows a better understanding and reasoning of the system behavior than sampling. However, tracing monitoring generates a much larger volume of data than sampling. In order to reduce this data volume problem, some researchers have been working on encoding techniques [12]. A more common and straightforward way to reduce data volume is to collect interesting events only, and not all events that happen during a program execution [7][9]. This second approach may limit the analysis of events and conditions unforeseen previously to the program execution, though.

Both *state* and *event* information is important to understand and reason about the program execution [14]. Since *tracing* monitoring collects information when *events* occur, *state* information can be maintained by collecting the events associated to state changes. With a hybrid approach, the sampling monitoring can represent the action of collecting state information into an event for the tracing monitoring. Like any other event, not all events with state information should be collected, but only those events of interest. Integrating *sampling* and *tracing* monitoring and collecting the state information through events reduce the complexity of the monitoring task.

The monitoring system needs to know what are the events of interest, i.e. *what events should be collected*. Therefore, it provides an event specification language to the user. Additionally, it needs to know *what kind of analysis* it should perform over the collected information. The user may provide a specification of the *correct behavior* of the system and the monitoring checks for its correctness, showing when the system did not perform accordingly to the specification. Another approach is to have the user specifying the *conditions of interest*, and the monitoring system identifying and notifying him/her when these conditions are detected. A third approach, not frequently used by monitoring systems, is to *characterize* (build a model of) the system behavior from the program execution, mainly for program understanding and dynamic documentation.

Since analysis is so intrinsic to the monitoring activity, it became normal to have monitoring specification languages where the user describes not only the events, but also the analysis to be performed. As a consequence, monitoring specification languages are biased to the kind of analysis performed by the monitoring system. To the best of our knowledge, there is no monitoring specification language that separates the concerns of "*what are the events of the system?*" (describing the events of interest only), "*what is(are) the purpose(s) for monitoring the system?*" (performance, reliability, etc), and "*what kinds of analysis should be performed?*" (i.e. condition detection; correctness checking or comparison; or model characterization).

In the current step of our research, we are focusing on the first question for monitoring specification languages, i.e. "*what are the interesting events of the system?*". We are defining an extendable and flexible language (xMonEve) for describing monitoring events independently of the system implementation, the purpose of analysis, and the monitoring system.

2.1 Requirements for xMonEve

Initially, we identified new requirements for event description languages. Some of the requirements that guide us through the development of xMonEve are:

- *general purpose*: need to be flexible enough to accommodate event description for multiple monitoring purposes (i.e. independent of the analysis to be performed);
- *independence of monitoring system*: must allow generic description of events, both primitive and composed, not restricted to a specific monitoring system (or environment);
- *implementation independence*: need to provide mechanisms that separate the conceptual event to the implementation mapping;
- *reusable*: event description should be reusable independently of the implementation and monitoring system;
- *extensible*: extension of event description should be supported, so more specific information can be associated to the events. For instance, one extension can be the association of monitoring events to software architectural elements.

Like most monitoring specification languages, xMonEve can represent both primitive and composed events. Primitive events are events that occur in a specific moment in time, i.e. an instantaneous occurrence. Composed events are events composed of other events (primitive or composed ones), and have a specific moment of start and end. While its starting time is defined by the first event to happen, the last event determines its ending time.

Composed events provide a higher-level abstraction for the system execution. Primitive events may be filtered out and abstracted into composed events, having unneeded details thrown away.

One important advantage of event description is that it is well suited to bridge the gap between software architecture and implementation (mapping). For multiple reasons (such as reuse, maintainability, performance, fault-tolerance, security, etc), the implementation *structure* may not exactly correspond to the conceptual architectural structure. Events imply in a *functional mapping* for associating architecture and implementation, instead of a *structural mapping*. A *functional mapping* between implementation and any previous software specification document (software architecture, requirements, etc) should be always possible. If a system functionality cannot be associated to implementation actions (independently of how hard it may be for a human being to do this association), than this functionality was not implemented at first place.

Therefore, although events play an important role in the mapping between architecture and implementation, event specification languages have often ignored this importance, and not provided any mechanism to associate these different abstraction levels.

2.2 Describing Events with xMonEve

The purpose of this paper is not to provide a complete discussion about the xMonEve language, but to give an overview of its concepts and emphasize some specific details relevant for the context of architecting dependable systems.

In xMonEve, every event type has *ID*, *name*, *description*, *attributes*, and *abstraction*. The *abstraction* field is used to associate the event to a context. For instance, while a primitive event "*open*" may be associated to the "*File*" abstraction, a composed event "*open*" may be associated to the "*CheckingAccount*" abstraction. It is important to note here that *CheckingAccount* may or may not represent a structure (e.g., class or subsystem) of the system implementation. This mechanism allows multiple levels of abstraction, from the implementation level to the requirement level, passing through design and also software architecture. In the previous example, CheckingAccount may be a component abstraction at the software architectural level.

```
<event name=open type=primitive ID=#>
  <abstraction>File</abstraction>
  <description>opening file</description>
  <attributes>
    <field name=filename type=string>
    <thread_id>
    <timestamp>
  </attributes>
  <...>
</event>
```

Figure 1. Example showing common features to every event.

Additionally to the features that are common to every event, primitive and composed events have other distinct characteristics.

2.2.1 Primitive Events

A primitive event may be in more than one system, and with different implementations. In order to have a reusable definition for this event, multiple implementation mappings should be allowed. So, primitive events may have zero, one, or multiple mappings. These events will typically have no mapping until the programmer specify them, since he is the one with the right knowledge.

```
<event name=open type=primitive ID=#>
  <...>
  <mapping>
    <system ref=java_library/>
    <language name=java/>
    <class name=java.io.File/>
    <type name=operation>File(String pathname)
    </type>
    <when type=method_exit/>
    <assignments>
      <set field=filename parameter=pathname>
    </assignments>
  </mapping>
  <...>
</event>
```

Figure 2. Example mapping a primitive event to the implementation. In this example, the event *open* occurs when the "method" (actually the constructor) of *java.io.File* class returns, and the event field *filename* has its value assigned from the *pathname* parameter.

2.2.2 Composed Events

When defining composed events, no mapping is needed, since it is composed of other events. Besides the common event fields, composed events have three extra sections: *composition*, *correlation* and *conditions*. In *composition*, it is described what are the event types that compose this event. In *correlation*, the sequence or order of these events to generate the abstract event. The *condition* section describes the conditions that have to be satisfactory between these events so the composed event can be identified.

```
<event name=AccountTranfer type=composite ID=#>
<abstraction>Client</abstraction>
<composition>
  <alias name=before event=Bank.TransferRequest/>
  <alias name=withdraw event=Account.Withdraw/>
  <alias name=deposit event=Account.Deposit/>
  <alias name=after event=Bank.Tranfer/>
</composition>
<attributes>
  <field name=client value=before.client/>
  <field name=from value=withdraw.account/>
  <field name=to value=deposit.account/>
  <field name=amount value=withdraw.amount/>
  <timestamp start=before.timestamp.start
             end=after.timestamp.end/>
</attributes>
<correlation method=regexp>
  <sequence min=1 max=1>
    <event alias=before min=1 max=1/>
    <parallel min=1 max=1>
      <event alias=withdraw/>
      <event alias=deposit/>
    </parallel>
    <event alias=after min=1 max=1/>
  </sequence>
</correlation>
<condition>
  <and>
    <exp> before.client = withdraw.client =
          deposit.client = after.client </exp>
    <exp> withdraw.amount = deposit.amount </exp>
  </and>
</condition>
<...>
</event>
```

Figure 3. Example of the composed event "AccountTransfer". In this example we can see what events compose this one (*composition*), what is the *correlation* between these events, and what *conditions* should be satisfied between those events.

3. ARCHITECTING DEPENDABLE SYSTEMS

With xMonEve, events can be described in both *top-down* and *bottom-up* approaches, since the language is independent of the development process. However, in the context of architectural development of dependable software, a top-down approach would be more natural (but not the only possible approach). The architect would describe (incomplete composed) events at the architectural level, while the designer and/or programmer would decompose these events into lower-level events, until they could be completely defined in terms of primitive events only.

First, in this section, we discuss the role of events as the integration element for the development of dependable systems from software architecture to program execution. Afterwards, we briefly present a top-down approach for architecting such systems.

3.1 Event as the Integration Element

According to Hofmann et al. [4], both monitoring and modeling rely on a common abstraction of a system's dynamic behavior, *the event*, and therefore can be integrated to one comprehensive methodology for measurement, validation and evaluation.

When considering modeling and analysis techniques that have been applied for designing dependable (reliable) system, Markov models and simulations stand out [8]. It is important to note that the *event* abstraction is also common to these techniques. A Markov model has a state changed with the occurrence of an event, which time to occurrence is often modeled with a random exponential distribution. During simulation execution, event traces are generated, over which analyses are performed.

Therefore, the *event* abstraction can act as the basic element for integrating: reliability models, architecture designs, system implementation, and analyses. In order to have this integration, an interchangeable (shared and canonical) representation of events should be available during the whole software development process. In this context, xMonEve represents an important step towards this integration.

3.2 Top-Down Approach

Here, we informally and briefly describe a top-down approach for architecting dependable systems by using events as basic elements of integration.

When building Markov models for reliability analysis, architects and designers may associate information about the model to the events. In this case, the event would include the information about the state change, and also the random distribution of its occurrence. This event definition could be used for running reliability analysis prior to the system development.

```
<event name=enter_overload_state type=composite …>
  <abstraction>ComponentA</abstraction>
  <markov_model>
    <transition from="overload_state"
                to="failure_state"/>
    <distribution (...) />
    <...>
  </markov_model>
  <...>
</event>
```

Figure 4. Extension of an event description with information for the Markov model.

Independently of having or not Markov (or others) extensions to an event definition, software architects, designers and programmers may compose (or decompose) an event from (into) other events, by defining and associating these new events. Thus, multiple levels of event abstraction can be created, from requirements and software architecture abstractions to implementation primitive events.

```
<event name=overload_timeout type=composite …>
  <abstraction>ComponentA</abstraction>
  <markov_model>...</markov_model>
  <composition>
    <alias name=eos event=enter_overload_state />
    <alias name=avg event=loadAverageSampling.../>
  </composition>
  <attributes>
    <field name=status .../>
    <field name=loadaverage value=avg.la .../>
    <...>
  </attributes>
  <...>
  <condition>
    <and>
      <exp>status = "running"</exp>
      <exp>loadaverage > 10</exp>
      <exp>ellapsedtime(eos.timestamp.end)>5</exp>
    </and>
  </condition>
</event>
```

Figure 5. Event definition of Figure 4 with the information added by software architects, designers and/or programmers.

After the implementation of the application, with the event description represented in xMonEve, a monitoring system can observe the application execution and analyze its behavior at multiple abstraction levels, depending on the purpose and interest of the user. For instance, analysis can happen at the implementation level for debugging, performance, testing etc, as well as at the architectural level for dependability, performance, validation etc.

4. RELATED WORK

Many specification languages have been proposed in the literature for describing events (and states) for monitoring technique. The definition of xMonEve is influenced by characteristics present in most of them.

Some specification languages were developed based upon extended regular expressions, such as EBBA [1]. These languages put more emphasis in temporal ordering, and, in general, have limited capability to specify states, and events are assumed to occur instantaneously. These languages influenced xMonEve in the specification of the correlation of composed events, although in xMonEve we also consider non-instantaneous events.

Snodgrass [15] developed a query language for a history database, using it to specify events and states. Although this work has a large influence in monitoring techniques, the language has a limited set of operators from relational algebra with a limited representation power. One important influence of this work in ours is that, in this work, with relational algebra, the language expresses *what* derived information is desired, and not *how* it is derived.

PMMS [9] uses a specification language based on relational calculus to for description of events and user questions. A big contribution of this work is in providing an automatic technique for instrumenting the program code to collect only the events needed to answer explicit user questions. This technique removes the burden of code instrumentation from the programmer. This specification language has limitations to specify events, and this is linked to the fact that PMMS supports tracing monitoring only, and no sampling.

Shim et al. [14] proposed a language based on classical temporal logic for specifying event and states. This work influenced us in considering non-instantaneous events. However, they do not provide any mechanism to create different levels of abstraction (to associate, for instance, events to software architecture elements), neither an extensible way to associate more semantics to the event specification.

With FLEA [3], user expresses his/her requirements and assumptions for monitoring. The main idea behind it is to be able to monitor programs that were not developed with monitoring in mind, and to check software requirements though events. In a similar way, xMonEve is meant to be independent of implementation, and this also includes its structure. Additionally, we also think it is important to bridge different abstractions, such as requirements and implementation, and any other possible abstraction.

Another kind of related work is the application of software monitoring at the architecture level[1][16]. It is worth to mention that both works consider the instrumentation of connectors for collecting the information, instead of the components, and the basic element for analysis is the event at the architectural level. In these works, the mapping problem between software architecture and implementation is simplified since the implementation and software architecture design presents a one-to-one structural correspondence.

5. CONCLUSIONS AND CURRENT WORK

The event and its definition play a major role in the integration of development techniques for architecting dependable systems, since it is a common abstraction to multiples techniques. However, to have an effective integration, events also have to be described in a common way. xMonEve is an event description language for this integration purpose. We are currently working on xMonEve definition and refinement. xMonEve does not describe *how* the event is going to be collected, but *what* that event is or represents. xMonEve is not intended to be a substitute for other event specification languages, but to promote integration of techniques by providing an interchangeable description for events.

In this position paper, we present the problem of mapping implementation to software architecture; discuss the importance of the event description in the context of developing complex and reliable systems; present requirements for event description languages; presented our current work in xMonEve; show how xMonEve can support integration of reliability techniques and software architectures; propose a top-down approach for reliability; and compare our work with other specification languages from the literature.

Inside this paper, in many occasions we say "*the developer would describe the event*", or similar. However, this is a hard task by itself and should be supported by tools. Event definitions could and should be generated from other system documents, such as requirement specifications, architectural and design models, testing documents, etc. This type of tool support is also an important step towards the usefulness and success of monitoring techniques, as well as such event specification languages.

At this step, we have not gotten yet to analysis description, i.e., how to describe what types of analyses a monitoring system should perform, and for what purpose. Now, it is important for us to understand better how each different purpose may affect monitoring systems. Since a major part of the functionality of monitoring systems is the same in multiple occasions, we probably need a family of monitoring systems with customizable components, so the configuration of a monitoring systems could go one step forward. Instead of configuring sensors and probes, configuration would represent the tailoring of the whole monitoring system to attend specific developer needs.

6. REFERENCES

[1] R. Balzer, "Instrumenting, Monitoring, & Debugging Software Architectures", 1997. [http://citeseer.nj.nec.com/411425.html]

[2] P. C. Bates, "Debugging heterogeneous distributed systems using event-based models of behavior", ACM Trans Computer System, vol. 13, n. 1, Feb. 1995, pp. 1 – 31

[3] D. Cohen, M. Feather, K. Narayanaswamy, and S. Fickas, "Automatic Monitoring of Software Requirements", Proc Int'l Conf Software Engineering (ICSE) 1997, pp. 602-603.

[4] R. Hofmann, R. Klar, B.Mohr, A. Quick, and M. Siegle, "Distributed Performance Monitoring: Methods, Tools, and Applications", IEEE Trans. Parallel and Distributed Systems, vol. 5, n. 6, June 1994, pp.585-598.

[5] Y. Huang and C. Kintala, "Software Implemented Fault Tolerance: Technologies and Experience", Proc. 23rd Int'l Symp on Fault Tolerance Computing, 1993, pp. 2-9.

[6] C. Jeffery, "Program Monitoring and Visualization: An Exploratory Approach", Springer-Verlag, 1999.

[7] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring Distributed Systems. ACM Transactions on Computer Systems, vol. 5, no. 2, May 1987.

[8] J. F. Kitchin, "Practical Markov Modeling for Reliability Analysis", Proc Annual Reliability and Maintainability Symposium, 1988, Jan. 1988, pp. 290-296.

[9] Y. Liao and D. Cohen, "A Specification Approach to High Level Program Monitoring and Measuring", IEEE Trans. Software Engineering, vol. 18, n. 11, Nov. 1992.

[10] N. Medvidovic, D. Rosenblum, and R. Taylor, "A Language and Environment for Architecture-Based Software Development and Evolution", Proc Int'l Conf on Software Engineering, May 1999, pp. 44 -53

[11] D. M. Ogle, K. Schwan, and R. Snodgrass, "Application-Dependent Dynamic Monitoring of Distributed and Parallel Systems", IEEE Trans. Parallel and Distributed Systems, vol. 4, n. 7, July 1993, pp. 762-778.

[12] S. Reiss, and M. Renieris, "Encoding Program Executions", Proc Int'l Conf Software Engineering, May 2001.

[13] B. Schroeder, "On-Line Monitoring: A Tutorial", IEEE Computer, vol. 28, n. 6, June 1995, pp.72-77.

[14] Y. C. Shim and C.V. Ramamoorthy, "Monitoring and Control of Distributed Systems", Proc. 1st Int'l Conf on System Integration, Apr. 1990, pp. 672-681.

[15] R. Snodgrass, "A Relational Approach to Monitoring Complex Systems", ACM Trans. Computer Systems, vol. 6, n. 2, May 1988, pp.156-196.

[16] M. Vieira, M. Dias, D. Richardson, "Analyzing Software Architecture with Argus-I", Proc Int'l Conf on Software Engineering, June 2000, pp. 758 –761.

Using Architectural Properties to Model and Measure System-Wide Graceful Degradation

Charles P. Shelton
ECE Department
Carnegie Mellon University
Pittsburgh, PA, USA
cshelton@cmu.edu

Philip Koopman
ECE Department
Carnegie Mellon University
Pittsburgh, PA, USA
koopman@cmu.edu

ABSTRACT

System-wide graceful degradation may be a viable approach to improving dependability in computer systems. In order to evaluate and improve system-wide graceful degradation we present initial work on a component-based model that will explicitly define graceful degradation as a system property, and measure how well a system gracefully degrades in the presence of multiple combinations of component failures. The system's software architecture plays a major role in this model, because the interface and component specifications embody the architecture's abstraction principle. We use the architecture to group components into subsystems that enable reasoning about overall system utility. We apply this model to an example distributed embedded control system and report on initial results.

1. INTRODUCTION

Dependability is a term that covers many system properties such as reliability, availability, safety, maintainability, and security [4]. System dependability is especially important for embedded computer control systems, which pervade everyday life and can have severe consequences for failure. These systems increasingly implement a significant portion of their functionality in software, making software dependability a major issue.

Graceful degradation may be a viable approach to achieving better software dependability. If a software system can gracefully degrade automatically when faults are detected, then individual software component failures will not cause complete system failure. Rather, component failures will remove the functionality derived from that component, while still preserving the operation of the rest of the system. Specifying and achieving system-wide graceful degradation is a difficult research problem. Current approaches require specifying every system failure mode ahead of time, and designing a specific response for each such mode (e.g., [2]). This is impractical for a complex software system, especially a fine grained distributed embedded system with hundreds or thousands of software and hardware components.

In order to evaluate and improve system-wide graceful degradation, we present a component-based system model that provides a means for evaluating and predicting how well a system should gracefully degrade, as well as how graceful degradation influences dependability properties. We base the model on using the system's interface definitions and component connections to group the system's components into subsystems. We hypothesize that the software architecture, responsible for the overall organization of and connections among components, can facilitate the system's ability to implicitly provide the property of graceful degradation, without specifying a response to each possible failure mode at design time. We define a failure mode to be a set of system components failing concurrently. By using the model to measure how gracefully a system degrades, we predict that we can identify what architectural properties facilitate and impede system-wide graceful degradation.

Related to our concept of graceful degradation is the term survivability. Survivability is another property of dependability that has been proposed to explicitly define how systems will degrade functionality in the presence of failures [3]. Our work differs from survivability specifications in that we are interested in building implicit graceful degradation into systems without specifying failure scenarios *a priori*, and having the system "do the right thing" in the presence of component failures. Also, we are focusing on distributed embedded systems rather than on large-scale critical infrastructure information systems.

The remainder of this paper is organized as follows. Section 2 describes our initial system model and key assumptions. Section 3 describes our representative distributed embedded system and its architecture, and applies our model to this architecture. Section 4 includes discussion about the model's predictions, and how they compare to initial fault injection tests we ran with a simulated version of the control system. Section 5 wraps up with conclusions and future work.

2. SYSTEM MODEL

As a first step, we are concentrating on software architecture at a high level of abstraction. Our system model initially focuses on the "functionality" components of the system: software, sensors, and actuators. We make the initial assumptions that individual software components each have their own processing elements, that there is enough network bandwidth to transmit all needed sensor values, and that there are enough system resources to satisfy real-time requirements. These system aspects will all influence system-wide graceful degradation, but we are planning to include them in the model at later stages.

We consider a system as a set of software, sensor, and actuator components. We use the interfaces among components to define a set of *system variables* through which all components communicate. These variables can represent any communication structure in the software implementation. Actuators receive input variables and output to the environment, while sensors receive input from the environment and output system variables. We assume that components can either be in one of two states: working or failed. Working means that the component has enough resources to output its specified system variables. Failed means the component cannot produce its specified outputs.

The fault model for our system uses the traditional fail-fast, fail silent assumption. All faults are manifested as the loss of system variable communication among components. Components either provide their output variables or do not. Thus, failures can be detected when components do not provide their outputs when specified. This does not account for more complex types of failures such as providing invalid but syntactically correct information, and assumes component failures can be quickly detected. Fault detection and propagation issues are challenging research areas in and of themselves, and are outside the scope of this work. Additionally, since software component failure rates are difficult to identify, we make an initial assumption that all components have approximately equal failure rates.

A key concept in our model is the notion of *utility*. Utility is a measure of how much benefit can be gained from the entire system, a certain subsystem, or an individual component. For the entire system, the overall utility is determined by a nonlinear function of its individual subsystem utilities. Each subsystem's utility is determined by a nonlinear function of its individual component utilities. For individual components, we define a component's utility to be 1 when working and 0 when failed. We assume that if all components are working the system will be at its maximum utility, and if all components are failed, the system will have an overall utility of zero. Thus, a system gracefully degrades if individual component and subsystem failures reduce system utility gradually.

In our model, we initially concentrate on measuring whether the system has zero or positive overall utility by identifying how resistant critical subsystems are to component failures. Determining the functions that quantitatively measure how working components improve subsystem and system utility values is a challenging problem. However, without knowing these functions we can initially make a distinction between a system that is working and has positive but not necessarily maximum utility, and a failed system that has zero utility. In order to make this distinction we must have a clear definition of what "working" means for the entire system. In other words, we must specify what features of the system are necessary for the system to complete its primary functions. In most cases, this is not all the features available in the system. For example, the primary function of a car is to provide transportation. Critical features necessary for the car to continue working include engine and transmission control, brakes, and steering. The power windows, emissions control, air conditioning, and radio provide auxiliary functionality not necessary for the car to complete its primary task, and can be lost without causing a catastrophic failure.

The system can have many different component configurations based on which components are working or failed. If n is the number of components in the system, then there are 2^n different configurations that can be considered. The system's component configuration determines the utility of all its subsystems, and thus the utility of the entire system. An ideal gracefully degrading system is one where a large fraction of these 2^n configurations result in a system with overall positive utility; *i.e.*, the system can tolerate multiple combinations of component failures and still provide useful functionality.

Our first metric for graceful degradation is the system's resistance to complete failure (zero system utility). We determine this value by looking at how many configurations result in a system with positive utility. The ratio of log_2 *[number of valid component configurations] / n* gives a measure of how many configurations will provide utility relative to the total number of system configurations. This value is 0 (only one valid configuration) for a brittle system, and 1 for a perfect system where any component configuration can provide some utility (ignoring the trivial configuration of zero components that results in no system at all).

Clearly, if we had to consider the utility of every possible component configuration individually, then specifying graceful degradation becomes exponentially difficult as the number of components increases. However, we can use the system's software architecture, which defines system software components, input and output interfaces, and connections among components, to group components into subsystems according to the system variables they provide, and thus reduce complexity.

We define these subsystems in our component model as *feature subsets*. A feature subset is a set of components (software components, sensors, actuators, and possibly other feature subsets) that work together to provide a set of output variables. Feature subsets may or may not be disjoint and can share components across different subsets. Feature subsets have utility values based on which of their components are working, and contribute to overall system utility. A feature subset is critical if its functionality is required by the system; *i.e.*, the total system utility is zero whenever any critical feature subset has zero utility. Thus, the system will have positive utility if and only if all of its critical feature subsets have positive utility. If we view the system as a set of feature subsets rather than individual components, then we should only need to consider valid component configurations of critical feature subsets rather than configurations of all system components to determine how well the system gracefully degrades.

In addition to grouping components into feature subsets, we define a set of dependency relationships between feature subsets and their components. A feature subset may have strong dependence on some of its components, weak dependence on others, and some of its components may be completely optional. A feature subset strongly depends on one of its components if the loss of that component results in the feature subset's having zero utility. A feature subset weakly depends on one of its components if the loss of that component reduces the feature subset's utility to zero in some, but not all, configurations in which that component was working. For example, if there are two components that output a required system variable, loss of both will result in the feature subset having zero utility, but loss of only one or the other will not. If a component is optional to a feature subset, then it may provide enhancements to the feature subset's utility, but is not critical to the operation of the feature subset. Every valid component configuration of the feature subset where that component is working still provides positive (but possibly lower) utility when that component is broken. These dependency relationships can also exist among individual components as well, based on their input and output interfaces. A component that requires a certain system variable as an input will depend on the components that provide it as an output.

We can use this model to develop a space of systems with varying degrees of graceful degradation. At one end of the spectrum, we have extremely "brittle" systems that are not capable of any graceful degradation at all. In these systems, any one component failure will result in a complete system failure. In our model, this would be a system where every component is within a critical feature subset,

Table 1. Sensors, Actuators, and Software Components in the Elevator Architecture

Sensor Type	#	Output Variable	Actuator Type	#	Input Variable	Software Component	#	Output Variable
DriveSpeed	1	DriveSpeed	Drive Motor	1	DriveMotor	Drive Control	1	DriveMotor
CarPosition	1	CarPosition	Door Motor	1	DoorMotor	Door Control	1	DoorMotor
AtFloor	**f**	AtFloor[f]	Emergency Brake	1	EmergencyBrake	Safety	1	EmergencyBrake
HoistwayLimit	2	HoistwayLimit[d]	Car Lanterns	2	CarLantern[d]	Dispatcher	1	DesiredFloor
DoorClosed	1	DoorClosed	Car Position Indicator	1	CarPositionIndicator	VirtualAtFloor	**f**	AtFloor
DoorOpened	1	DoorOpened	Car Button Lights	**f**	CarLight[f]	Lantern Control	2	CarLantern[d]
DoorReversal	1	DoorReversal	Hall Button Lights	**2f-2**	HallLight[f,d]	Car Position Indicator Control	1	CarPositionIndicator
Car Buttons	**f**	CarCall[f]				Car Button Control	**f**	CarLight[f]
Hall Buttons	**2f-2**	HallCall[f,d]				Hall Button Control	**2f-2**	HallLight[f,d]

and each feature subset strongly depends on all of its components. Therefore, every component must be functioning to have positive system utility.

Similarly, any modular redundant system can be represented as a collection of several critical feature subsets, where each feature subset contains multiple copies of a component plus a voter. The valid configurations that provide positive utility for each feature subset are those that contain the voter plus one or more component copies. This redundant system can tolerate multiple failures across many feature subsets, but cannot tolerate the failure of any one voter or all the component copies in any one feature subset.

At the other end of the spectrum, an ideal gracefully degrading system is one where any combination of component failures will still leave a system with positive utility. In our model, this system would be one where none of its feature subsets would be labeled as critical, and every component would be completely optional to each feature subset in which it was a member. The system would continue to have positive utility until every component failed.

3. EXAMPLE SYSTEM: A DISTRIBUTED ELEVATOR CONTROL SYSTEM

To illustrate how we can apply our system model to a control system, we will use a model of a relatively complex distributed elevator control system. The complete details of the model have been published in [6], but we will describe a portion of the system and the software architecture here for clarity.

The general requirement for an elevator is that it must safely transport people among floors in a building. The control system has a set of sensors (door opened/closed, elevator speed, button sensors, etc.) for determining the current environment and passenger requests, a set of actuators (door motor, drive motor, emergency brake, lights, etc.) for performing tasks and informing passengers about system state, and a set of software objects (door controller, drive controller, dispatcher, etc.) that implement the control logic to perform the elevator's functions.

Table 1 summarizes the list of sensors, actuators, and software components in the elevator control system. In the table, *f* represents the number of floors in the elevator's building, and *d* represents a choice of two directions, up or down. For example, there are *f* floor sensors and *f* car button sensors (one for each floor), two hoistway limit sensors (the "up" sensor is at the top of the hoistway, and the "down" sensor is at the bottom), and $2f - 2$ hall button sensors (two per floor in each direction, except for the top and bottom floors, which only have one button). In the table each sensor has a specified output variable, and each actuator has a specified input variable. The software components have several inputs and a few outputs. There are a total of $14 + 11f$ components in the system.

The system's software architecture defines each component's input and output interface, as well as connections among components, which can be used to construct the system's feature subsets. Figure 1 shows the critical feature subsets of the system, and the dependencies between the feature subsets and components. Each arrow in the figure represents a system variable being communicated between components. In an elevator control system, the only critical functions of the elevator are that it must be able to service all floors, open and close the doors, and ensure the safety of the passengers. All other functionality, such as responding to passenger requests, providing passenger feedback, and minimizing wait time and travel time, are enhancements over the basic elevator requirements. Therefore, the critical feature subsets for this system are only the feature subsets that are required to operate the drive motor, door motor, and emergency brake actuators.

The software components are designed to have a default behavior based on their required inputs, and to treat optional inputs as "advice" to improve functionality when those inputs are available. For example, the Door Control and Drive Control components can listen to each other's command output variables in addition to the Drive Speed and Door Closed sensors to synchronize their behavior (open the doors more quickly after the car stops), but only the sensor values are necessary for correct behavior. Likewise, the Drive Control component has a default behavior that stops the elevator at every floor, but if the DesiredFloor variable is available from the Dispatcher component, then it can use that value to skip floors that do not have any pending requests. Also, the Door Control component normally opens the door for a specified dwell time, but can respond to button presses to reopen the doors if a passenger arrives. We also enumerated the other non-critical feature subsets in the elevator system such as the various passenger feedback lights in the elevator, but we omit them here for the sake of brevity.

4. ANALYSIS

In order to derive the graceful degradation metric for our elevator control system, we need only consider the critical feature subsets and the components upon which they depend. Therefore, all configurations containing enough components to provide working Drive

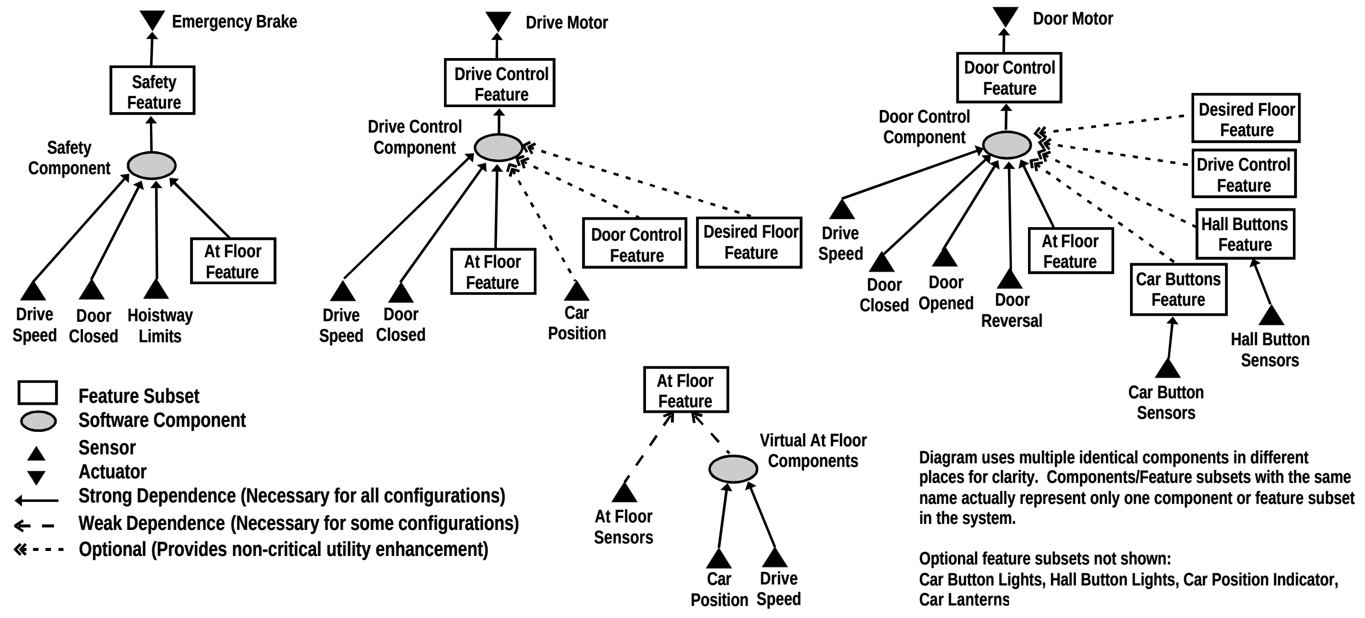


Figure 1. Critical Feature Subsets in the Elevator Control System

Control, Door Control, Safety, and AtFloor feature subsets are valid, and can contain any arbitrary combination of other optional system components. There are $1 + 9f$ optional system components (which can be arranged in $2^{1 + 9f}$ different arbitrary combinations), leaving $13 + 2f$ critical components (components within critical feature subsets) that have configurations that require examining (and $2^{13 + 2f}$ component combinations left to consider individually).

By examining the critical feature subsets, we can see that they are strongly dependent on the Drive Speed, Door Closed, Door Opened, Door Reversal, and Hoistway Limits sensors, the Drive Control, Door Control, and Safety software components, and the Drive Motor, Door Motor, and Emergency Brake actuators. Any valid configuration must have all of these twelve components present. Therefore, we can restrict the number of configurations we calculate by not considering any configurations in which these components are broken.

This leaves $1 + 2f$ components (the Car Position sensor, the AtFloor sensors, and the VirtualAtFloor software components) in the AtFloor feature subset to be considered. By examining the critical feature subsets, we have systematically reduced the graceful degradation calculation from considering $2^{14 + 11f}$ combinations to $2^{1 + 2f}$ combinations. Now we can determine the number of valid configurations for the AtFloor feature by noting that all floors must be serviced by the elevator. Therefore, on each floor there must be a working AtFloor sensor or a working VirtualAtFloor component with a working Car Position sensor. If the Car Position sensor breaks, then all AtFloor sensors must work. Since all the AtFloor sensors must work in this situation, they are fixed and have one configuration. However, the VirtualAtFloor components can either work or not work since their failure will not affect the availability of the AtFloor system variables, making 2^f valid combinations for the various VirtualAtFloor components. If the Car Position sensor works, then one or both AtFloor sensor and VirtualAtFloor component must work for each floor, so the only invalid combinations are when both have failed for at least one floor. This means there are 3 valid combinations per floor, making 3^f valid combinations out of the possible 2^{2f}. Thus there are $2^f + 3^f$ valid combinations of components in the AtFloor feature subset.

Multiplying this with the number of combinations of optional components results in a total of $(2^f + 3^f)(2^{1 + 9f})$ valid component configurations. Taking the base 2 log of this and dividing it by the total number of system components $(14 + 11f)$ gives us our graceful degradation metric. If we calculate this value for an elevator that serves seven floors, we get 0.83.

For comparison, we also consider an elevator system that does not contain any VirtualAtFloor software components. The VirtualAtFloor components improved the system's ability to gracefully degrade because they provided a way to compensate for AtFloor sensor failures by using information provided by other system sensors to synthesize AtFloor sensor values. Therefore, if we remove the VirtualAtFloor components, the resultant system should also receive a lower graceful degradation value.

In our model, the removal of the VirtualAtFloor components reduces the AtFloor feature subset to being strongly dependent on all of the AtFloor sensors. Therefore there is only one valid configuration for the AtFloor feature subset in which every AtFloor sensor must work. Since this is a critical feature subset, all valid system configurations must contain a working AtFloor feature subset. Additionally, the Car Position sensor becomes an optional system component because the AtFloor feature subset no longer depends on it. This results in there being only $2^{2 + 9f}$ valid system configurations since most of the components in the critical feature subsets must work and only the optional components can have multiple valid configurations. The total number of system components is also reduced by the removal of the f VirtualAtFloor components, leaving $14 + 10f$ total system components. For a seven-floor elevator, this results in a graceful degradation score of 0.77.

The graceful degradation metric provides a concrete comparison among similar systems. We can quantitatively assess how adding or subtracting components to the system affects its ability to gracefully degrade. However, this metric may be misleading when comparing two systems that are substantially different in terms of functionality and number and type of system components.

We have developed a discrete event simulator that implements our elevator architecture, and have run some initial fault injection experiments to evaluate whether the implemented system actually gracefully degrades. So far, every test we have run with one of the possible valid configurations was able to successfully deliver all passengers to their destination floors, including a test that failed all components but the critical ones and the AtFloor sensors.

5. CONCLUSIONS AND FUTURE WORK

We have demonstrated a component-based system model that can provide insight into how well a system will perform graceful degradation in the presence of multiple component failures. We developed an initial metric for graceful degradation that indicates how many combinations of component failures can be tolerated by examining critical subsystem configurations rather than considering every possible system component configuration. In some initial experiments on a simulated implementation of the example control system studied, we found that the architecture described was resistant to certain combinations of component failures, as predicted by the model.

We did not incorporate failure recovery scenarios for every possible combination of component failures, but rather built the software components to the architectural specification. The individual components were designed to ignore optional input variables when they were not available and follow a default behavior. This is a fundamentally different approach to system-wide graceful degradation than specifying all possible failure combinations to be handled ahead of time.

Properties of the software architecture such as the component interfaces and the identification and partitioning of critical system functionality from the rest of the system seem to be key to achieving system-wide graceful degradation. The model we developed illustrates how well a system can gracefully degrade by using the software architecture's component connections to decompose the system. We are also exploring how to use an architectural description language such as Acme [1] to provide rigorous component interface specifications and facilitate development of our system model.

For this particular system, it is relatively easy to calculate the possible valid configurations by examining the software architecture without the model. However, the model provides a systematic framework for partitioning the system based on its software architecture, and we hypothesize that it will be useful in evaluating any architectural specification that has a well-defined component interface. This framework allows us to measure the graceful degradation properties of individual feature subsets with respect to their components as well. The architecture and model also identify a set of critical system components within the critical feature subsets that must continue to operate to provide any system functionality. This can be used to determine on which system components to spend effort ensuring component reliability through redundancy and other fault tolerance measures.

Our next step is to extend this model to incorporate the allocation of the software components to hardware units. In a distributed system, components that communicate via the network are strongly dependent on the network for their required input variables, making the network a single point of failure. Also, software components are strongly dependent on the hardware node on which they are hosted. These constraints will surely influence a system's ability to gracefully degrade (hardware failures might remove multiple components simultaneously), but may be ameliorated by system-wide reconfiguration as proposed in [5]. Additionally, we want to further develop the concept of system utility to not only distinguish between when the system is “broken” or “not broken,” but also different levels of functionality available in different system configurations. We have identified which configurations result in systems with positive utility, but we also need to quantitatively determine which of those configurations have higher utility than others. This will be based on determining which features are more useful than others based on measures such as performance and functionality.

6. ACKNOWLEDGMENTS

This work was supported by the General Motors Satellite Research Lab at Carnegie Mellon University, and Lucent Technologies. Thanks to Beth Latronico, Bill Nace, Orna Raz, and Yang Wang for their help in developing the ideas presented.

7. REFERENCES

[1] Garlan, D., Monroe, R.T., Wile, D., “Acme: Architectural Description of Component-Based Systems,” *Foundations of Component-Based Systems*, Leavens, G.T., Sitaraman, M. (eds), Cambridge University Press, 2000, pp. 47-68.

[2] Herlihy, M. P., Wing, J. M., “Specifying Graceful Degradation,” *IEEE Transactions on Parallel and Distributed Systems*, vol.2, no.1, pp. 93-104, 1991.

[3] Knight, J.C., Sullivan, K.J., "On the Definition of Survivability," University of Virginia, Department of Computer Science, Technical Report CS-TR-33-00, 2000.

[4] Laprie, J.-C., "Dependability of Computer Systems: Concepts, Limits, Improvements", *Proceedings of the Sixth International Symposium on Software Reliability Engineering*, Toulouse, France, Oct. 1995, pp. 2-11.

[5] Nace, W., Koopman, P., “A Product Family Approach to Graceful Degradation,” *Distributed and Parallel Embedded Systems (DIPES)*, October 2000.

[6] Shelton, C., Koopman, P., “Developing a Software Architecture for Graceful Degradation in an Elevator Control System,” *Workshop on Reliability in Embedded Systems* (in conjunction with Symposium on Reliable Distributed Systems/SRDS-2001), October 2001, New Orleans, LA.

Specification-Driven Prototyping for Architecting Dependability

Dennis B. Mulcare, ACM member
Consultant
cefsm@ellijay.com

Abstract - This paper describes a major part of an architecting methodology developed for safety-critical fault-tolerant software systems. The methodology coverage centers on specification-driven prototyping. This approach to prototyping is seen to be superior to the customary approaches of throwaway and evolutionary prototyping. A still developmental form of representation, higher-level statecharts, provides a suitably expressive prototype specification language.

Dependability is held to rely crucially on the rigor and specificity of the architecting process, as well as on the propagatability of its products. The subject four-step prototyping approach can subserve such needs, especially with regard to conceptualization insights, complexity management, dynamic analysis, and dependability assurances. Such efforts primarily address the underlying architecture or infrastructure of a nascent software system. In particular, the advocated prototyping approach focuses on absolute time-based concurrency, with accommodation of arbitrary scalability, non-ideal timing, and stochastic effects.

1 INTRODUCTION

Dependability refers to an encompassing qualitative judgment regarding the degree to which a software system merits or elicits the confidence of customers and users. Mainly, dependability resides in the extra-functional properties exhibited by a deployed system, and ultimately, in the system's underlying architecture or infrastructure. The underlying architecture includes most of the redundancy elements for fault tolerance, along with system-wide management logic. The definition and organization of such features during architecting present major leverage for ensuring high dependability from the outset of development.

Driven primarily by the demands of safety-critical systems, the development of associated methods has proceeded, with moderate success, for some three decades now. The routine use of these methods, however, has largely been restricted to embedded applications that prompt their use because of the severity of inherent hazards. Nonetheless, the adequacy of dependability methods and practices remains a continuing challenge because of the ever-increasing degree of sophistication and integration sought in embedded software systems.

1.1 Critical System Background

A dependable development process with reliable methods is essential to the assured development of dependable software systems. Safety-critical embedded systems, such as fly-by-wire flight control systems, motivated extensive research in associated development methods from the 1970s well into the 1980s. More or less independently, software-based prototyping methods evolved during the same period.

As evident by deployed aerospace systems, critical system development practice has been rather successful from about 1980 forward. Such success has in general been achieved because the associated software systems were:

- Dedicated to the critical function(s);
- Kept as simple as possible;
- Constrained in by limited computing resources;
- Supported by specialized hardware;
- Developed by staff with *a priori* understanding of the intended system needs and capabilities;
- Driven by considerations of safety and reliability.

In short, dependability was explicitly a major driver in architecting, implementing, and deploying such systems. While safety criticality was the paramount concern, other properties engendering dependability were necessarily and consciously incorporated into such systems. For example, various self-test and fault diagnosis features essential to safety also enhanced reliability and maintainability. Also, the criticality of real-time constraints levied stringent demands on infrastructure performance.

1.2 Current Dependability Challenges

Then, as processing power and functional integration became compelling realities, new issues regarding dependability arose because typically the physical segregation of critical functions was no longer programmatically viable. In turn, the requirements placed on the infrastructure became increasingly more diverse, demanding, and complex.

Regarding both research and practice, probably the greatest opportunity for dependability technology improvement exists at the architecting stage. Apart from the more customary concerns over functionality or applications, assured dependability mainly necessitates a rigorous approach to the formulation, analytical assessment, and verification of a nascent software system's underlying architecture. Typically, safety-critical systems involve redundancy management and hard real-time constraints that compel a focus on the dynamic analysis of concurrency logic and absolute timing in order to ensure requisite levels of dependability.

Such architecting activities can be facilitated through specification-driven prototyping, as depicted in Figure 1. Here, all prototype definition and development is first performed at the specification level. Then, the prototype is implemented or modified accordingly. Its execution is used in problem exploration, architecture development, dynamic analysis, and concept validation. Ultimately, the verified and optimized results, especially quantitative parameters as for timing, are propagated from the prototype specification to that of development product.

For the identified class of problems, however, two major problems persist regarding readily usable prototyping methods. First, there is the problem of a specification language that can express absolute time-based concurrency in an arbitrarily scalable manner. Such scalability should apply to both process types and data types needed in prototype specification. Second, specifications rendered in such a language should be directly and precisely translatable into an executable form. Such capacity is vital to methods automation. These issues, together with the inclusion of stochastic effects, appear to be neglected areas of research. These issues, moreover, would seem to be relevant to many classes of systems with significant dependability demands.

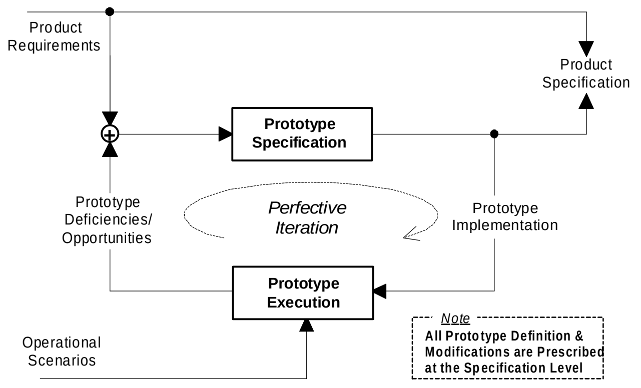


Figure 1. Specification-Driven Prototyping Concept

2 ARCHITECTING METHODOLOGY

A dependable architecting process, supported by reliable and effective methods, is necessary to ensure a dependable development product. To stabilize subsequent development and manage complexity throughout system development, the architecting process should capture all the *essential* problem complexity at the outset. Where demands on the architecture are stringent, the methodology should also be suitably rigorous and tailored to the particular system characteristics. For safety-critical embedded software systems then, the architecting methodology should preferably concentrate on system-wide control logic and address the underlying architecture largely separate from that of the applications architecture. This separation is generally quite tractable, and is beneficial for both static and dynamic analysis.

Table 1 summarizes salient attributes of the applications versus the infrastructure architectures, which together compose the software system architecture. Dependability associates mainly with the extra-functional requirements, which for safety-critical systems tend to have Go/No-Go acceptability criteria. Thus, the proper operation of the infrastructure must be sustained under both faulted and fault-free conditions, generally under very stringent absolute timing constraints. Moreover, the operation of the applications architecture is totally dependent on that of the infrastructure. Further, the acceptability criteria for the operation of the applications architecture are usually relatively tolerant, with some latitude for limited functional performance degradation.

From an architecting standpoint, the distinction in Table 1 between System STATE and Operational MODE is pivotal. System state denotes the state of the overall system, and hence the support available for functionality. On the other hand, operational mode effects the activation of particular functionality, conditional upon its availability per system state. System state is based on absolute time based concurrency logic that manages the system and its hardware elements. As such, the real-time determination of system is state is both critical and intricate, especially where diverse hardware operability and fault tolerance are involved.

As a consequence, the overall methodology that is context for this paper is called control state decomposition[1]. Practitioners in safety-critical systems have employed this kind of methodology, at least implicitly, to some degree. Otherwise, the linkage from dependability-related analyses to the development product is tenuous or obscure, and hence potentially misleading. Control state decomposition is relevant here for several reasons:

- Infrastructure-applications partitioning;
- Dependability-related product specification content;
- Focus of dynamic modeling and analysis.

APPLICATIONS ARCHITECTURE	INFRASTRUCTURE ARCHITECTURE
Functional Requirements	**Extra-functional Requirements**
System Services	**System Properties ("Ilities")**
What Kind(s) of Service	**How Well Service is Supported**
Operational MODE	**System STATE**
Functional Performance	**Infrastructure Performance**
Shades of Grey Criteria	**GO/NO-GO Criteria**

Table 1. Applications versus Infrastructure Architecture

2.1 Applications-Infrastructure Dichotomy

As seen in Figure 2, the architecting accomplished under the control state decomposition methodology begins with a delineation of software system requirements into functional and extra-functional ones. On separate but interdependent trajectories, the functional requirements drive the architecting of the applications architecture, while the extra-functional requirements drive the development of the underlying architecture. It serves as the system-wide platform used by the applications architecture.

The extra-functional requirements are typically rather high-level statements, usually quantitative ones. Hence, these requirements are subject to considerable interpretation and elaboration. Their interpretation largely dictates the degree of rigor and kinds assurance methods that need to be used in development. Their elaboration derives from the analysis and architecting associated with the functional requirements. As seen in Figure 2, the applications architecture furnishes its quantitative processing needs for infrastructure development. These needs are identified for all operational modes, with emphasis on worst-case demands.

The structure *per se* of the underlying architecture, such as redundancy management schemes and self-test mechanisms, derive largely from the extra-functional requirements. The suitability of that structure, however, is subject to the requirements derived through applications architecting. Further, the parameterization of that structure, such as capacity or timing quantities, is determined by the needs of the applications architecture. As suggested earlier, the interplay between the underlying and the applications architectures is mediated by the system state and operational mode logic. Associated information flow involves sensor, controller, display, and effector signals as essential to the intended system functionalities.

The system architecture integration shown in Figure 2 is a logical one that centers on dynamic analysis of various applications mode demands. The focus is on overall quantitative adequacy under worst-case demands. Cases of particular concern are the transient workload for handling faults and potential computational delays due to dependencies in the interleaving of distributed processes. At this stage, the associated software exists only in logical or specification forms, so the associated dynamic analysis is truly at an architectural level.

Such analysis involves prototype execution, or absolute time-based simulation, using representative and worst-case scenarios. Simulation provides tangible insights and quantitative calibrations of architectural acceptability, including information on behaviors pertaining to dependability. Such information can be vital to associated static analysis in terms of data or assumptions.

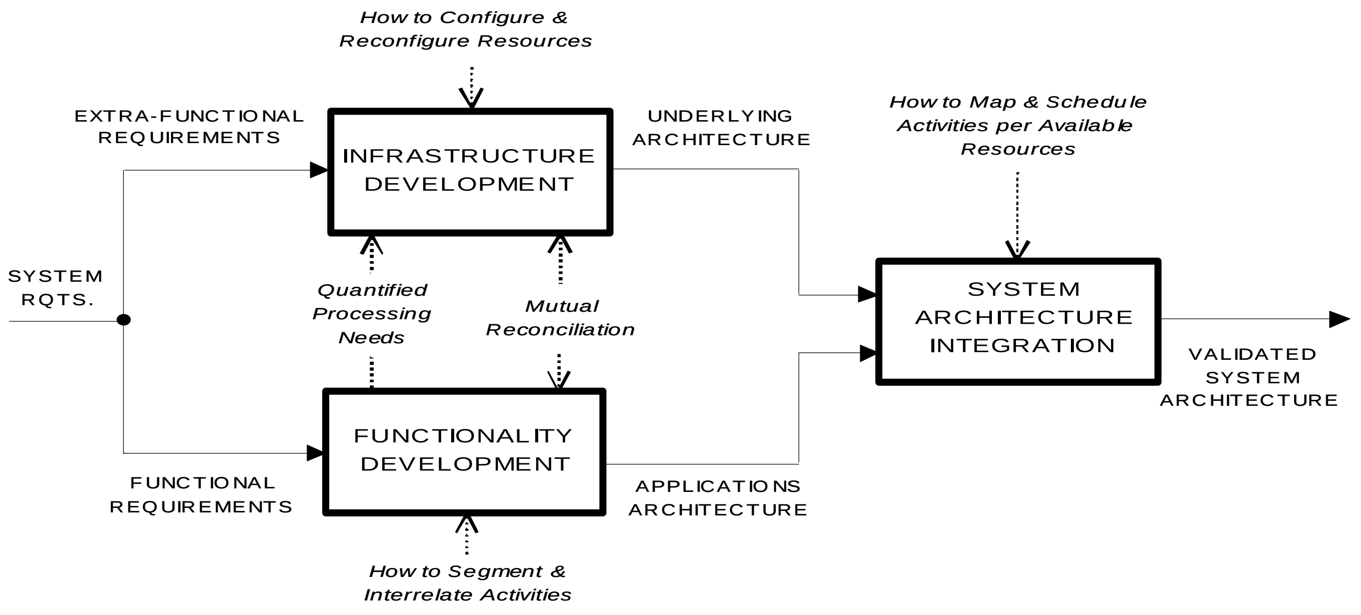

How to Configure & Reconfigure Resources
EXTRA-FUNCTIONAL REQUIREMENTS
INFRASTRUCTURE DEVELOPMENT
UNDERLYING ARCHITECTURE
How to Map & Schedule Activities per Available Resources
SYSTEM RQTS.
Quantified Processing Needs
Mutual Reconciliation
SYSTEM ARCHITECTURE INTEGRATION
VALIDATED SYSTEM ARCHITECTURE
FUNCTIONALITY DEVELOPMENT
FUNCTIONAL REQUIREMENTS
APPLICATIONS ARCHITECTURE
How to Segment & Interrelate Activities

Figure 2. Architecting Methodology Dichotomy

2.2 The Architecting of Dependability

During software system architecting, dependability is pursued indirectly through ensuring the aforementioned extra-functional requirements. For example, the fulfillment of safety, and reliability may be explicitly sought in composing a software system architecture. Pending architecture commitments then have to be analyzed to predict or confirm compliance with the respective requirements. All such assessments are not quantitative, as safety assurance also depends crucially on qualitative analysis.

In confirming extra-functional properties, it is vital to employ complementary simulation and analysis techniques. Here, simulation entails dynamic analysis, as contrasted with the static nature of analysis *per* se. Their mutually reinforcing attributes are outlined in Table 2. Necessarily, all assessments must use precise and consistent architectural representations. Then, the corroboration results obtained from static analysis and simulation must be established. Finally, the analytical models must be consistent with the finalized architecture definition.

.	ANALYSIS	SIMULATION
SCOPE	General Conclusions	Particular Conclusions
ORIENTATION	Equivalence Classes (Breadth)	Problematic Scenarios (Depth)
DOMAIN	Encompassing Properties	Selective Subset of Behaviors
KEY	Tractable yet Admissable Model Simplifications	Representative Scenario Selections
MECHANISM	Reasoning/Consequences	Stimulation/Observations
MODE	Static/Detached	Dynamic/Tangible
CLOSURE	Deductive	Inductive

Table 2 - Attributes of Analysis and Simulation

2.3 Specification-Driven Prototyping

The importance of the dynamic analysis of a developmental infrastructure architecture has been noted. This refers especially to the capacity to examine complex behaviors and models with selective high fidelity. Aside from detailed models, higher fidelity may involve an absolute timebase, the modeling of concurrency, or realistic scenarios. As noted in Table 2, problematic scenarios like the transient behavior exhibited during fault handling can be investigated through simulation, or prototype execution. Comparable static analysis is in general intractable. Since capabilities like timely and assured fault handling are essential to safety-critical systems, prototype execution is a vital architecting tool for ensuring reliability and safety from the outset.

The customary forms of software-implemented prototypes are the throwaway and the evolutionary approaches. Both approaches suffer from a lack of inherent discipline and focus. This renders their use for architecting critical systems problematic. The throwaway prototype has to be reversed engineered to recover the implicit semantics from code to propagate into actual product; this is a nebulous and error-prone task. Evolutionary prototypes tend to diverge from strictly relevant features and to embody dubious structure on which to base a development product.

Specification-driven prototyping was developed expressly to aid in the architecting of safety-critical systems. It overcomes problems associated with both throwaway and evolutionary prototyping, and provides a stable and rigorous basis for overall software system development. In specification-driven prototyping, neither the prototype nor its specification is thrown away. They are useful throughout system development, provided they are kept consistent with related aspects of the developmental software system. Further, the prototype is not itself evolved; only its specification is. As depicted in Figure 1, all prototype definition and development occurs at the specification level. The prototype is merely kept consistent with its specification. Hence, the prototype is not apt to diverge from the system requirements, and its complete semantics are always available in a version that has been partially verified through prototype execution.

Property	Realization	Role
Communicating	External events	Message passing Notification Request Timeout
Extended	Tokens	Local state data Message parameters
	Timing	Process duration Scheduling times Transmission delays
	Stochastics	Timing variation Demand variability Stochastic decisions
Finite-State Machine	Statechart subgraph	Active objects

Table 3. Higher-Level Statechart Expressiveness

2.3.1 Payoff from Specification-Driven Prototyping - Ultimately, appropriate information from the prototype specification is propagated to product specifications. Such information includes global concurrency logic, exact timing parameters with tolerances, and quantitative parameters such as sizing values. Unfortunately, this kind of information is often conservatively estimated in early-on product specifications, or acknowledged only by "TBDs." If such information is not cogently stipulated, the basis for subsequent development is questionable. Ill-informed or defaulted specification entries tend to yield component/sizing mismatches, performance deficiencies, trial-and-error development, and undue system complexity.

In summary, the motivation for specification-driven prototyping is to establish global concurrency logic and quantitative parameter values for the infrastructure architecture. The intent is not only to eliminate uncertainty, but also to ensure a more balanced, robust, and economical design. These attributes translate into a system implementation that has: assured real-time performance under worst-case conditions; minimal spurious fault alarms and service outages; freedom from disparate bottlenecks and surplus capacity; and in general stable parameter values.

2.3.2 Higher-Level Statecharts (HLSs) - HLSs were developed expressly to support specification-driven prototyping[2]. They were needed because of the lack of any other language with suitable expressiveness at that time. Previously, a kind of higher-level Petri net, a predicate-transition network, had been used, but it did not provide desired modeling construct modularity. Since a subgraph in a basic statechart constitutes a finite-state machine (FSM), statecharts possess the relevant modularity property. They model communication among FSMs through events issued across subgraph boundaries. These circumstances prompted innovations to extend the expressiveness of basic statecharts to that equivalent to predicate-transition networks, or the development of HLSs. So features like circulating tokens and complex transition rule syntax were added to basic statecharts to define HLSs.

HLSs may be thought of as *scalable* communicating extended finite-state machines (CEFSMs), with correspondences as seen in Table 2. The inherent scalability applies to tokens as well as to subgraphs. Tokens are based on (passive) data types, and subgraphs are defined as (active) process types. Prefix-dot notation is used to denote scalability in both cases. The result is arbitrary scalability with no changes at all to transition rules.

While of the same form as used for basic statecharts, the syntax for HLS transition rules is appreciably more complex. Augmented first-order logic is used to express HLS transition rules, with additional constructs to represent absolute timing and stochastic effects. Some other aspects of HLS transition rule notation are:

- Distinction between Logical- and Operational-ANDs;
- Distinction between Logical Exclusive-OR and Operational Exclusive-OR;
- Notation for Token Migration between subgraph Nodes;
- Compound Action-Parts;
- In-line Comments in Action-Parts.

Overall, HLSs enable a coherent form of nested abstractions, where the HLS itself represents an encompassing concurrency model. HLS subgraphs denote interacting process abstractions, which in turn operate on instances of various token types, or data abstractions. Consequently, state data is captured at three different levels: global, process, and data token levels. This multi-level state characterization coincides well with the aforementioned control state decomposition methodology, with its emphasis on system state and operational modes. And the top-level states and modes correspond explicitly with the models used for static analysis and product specification provisions.

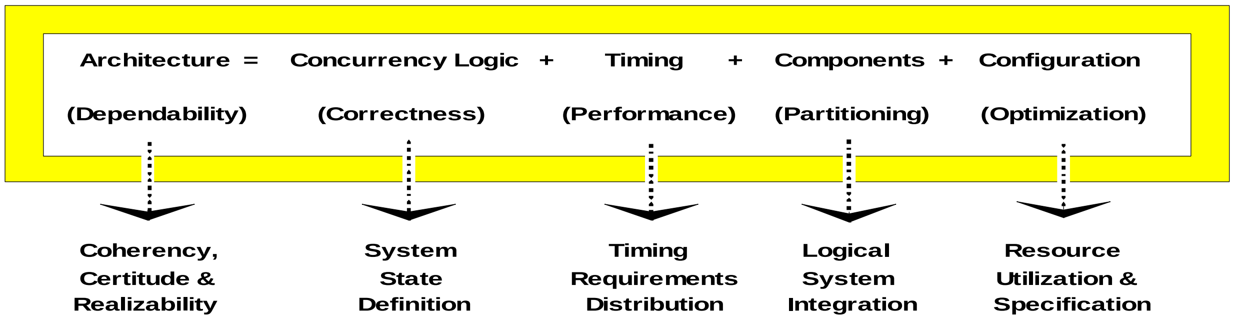
While HLSs have been applied extensively on a manual basis, their precise definition has not been completed. Accordingly, their precise mapping to an executable prototype has not been pursued. Consequently, such mappings have also been performed on a manual basis. Nevertheless, in numerous instances, successful prototypes have been developed, and the HLSs have been very incisive in informing prototype refinements.

3 PROTOTYPING METHODOLOGY

The prototyping methodology is summarized in Figure 3, where the same infrastructure architecture model, and hence prototype specification, are evolved over four stages of elaboration and assessment. Overall, the intent is to rigorously define and verify an infrastructure architecture with confirmed dependability properties like safety. The initial prototyping stage investigates the correctness of concurrency logic for the global management of the software system. Here, HLS process types correspond to logical elements in the nascent architecture. This activity serves to reveal many HLS specification deficiencies, and ultimately, to establish precise definition and management of overall system state(s).

The second stage of prototyping dynamically examines the absolute timing of interleaved processes in a distributed system, based on applied stimuli and the affected HLS transition rules. Here, the transition rules are expanded to include timing terms and possibly stochastic variations in prototype operation. The emphasis is on confirming real-time response and performance.

The third stage focuses on physical component partitioning and interfaces, which tend to associate with process type boundaries. Distribution of timing requirements and tolerances are allocated among components as well. Refined estimates for the applications architecture needs are introduced at this stage, which corresponds to the System Architecture Integration block in Figure 2. The application demands are merely simulated as dummy loads.


Architecture = Concurrency Logic + Timing + Components + Configuration
(Dependability) (Correctness) (Performance) (Partitioning) (Optimization)
Coherency, Certitude & Realizability
System State Definition
Timing Requirements Distribution
Logical System Integration
Resource Utilization & Specification

- Same Architecture Model(s) Evolved over the 4 Stages
- Outputs of Logic & Quantitative Parameters for Specifications

Figure 3 - Infrastructure Architecting Progression

The final stage is an optimization effort where the specific numbers of various component types and their respective quantitative parameters can be set to best overall advantage. This involves the use of stochastic simulation and objective functions. For example, a genetic algorithm has been used with stochastic simulation to optimize component counts and parameters[3].

This evolution of the specification-based prototype imparts precision and certitude, and hence dependability, to crucial aspects of the architecting process. Prototype execution also stimulates conceptual and usage insights. When appropriate, the first stage of prototyping, concurrency logic correctness, can be skipped. This does not affect the prototype specification, but it greatly simplifies its implementation. Even though the process remains a manual one, the benefits for crucial aspects of critical systems are seen as well worthwhile for orderly development and economical design.

3.1 Prototyping Experience

Two categories of experience with specification-driven prototyping have accrued: that derived during architecting activities, and that obtained in supporting physical system development. Architecting experience has in part accrued in the modeling of rate monotonic scheduling, Ada virtual nodes and remote rendezvous, the Pilot's Associate processing concept, and the optimization of an on-line transaction processing system[3].

Figure 4 depicts some results of prototype execution for the Ada virtual nodes concept. Here, the failure of one of the physical nodes has been simulated. Detection of the hardware fault has then prompted processor reconfiguration and application restart. The restart is based on given applications priorities and the availability of standby virtual nodes at operable physical nodes. The diagram also shows the interplay between the underlying and the applications architectures in the prioritized restoration of services. HLS events are shown in bold font on the interconnection arcs.

Two cases of prototyping support for system development are notable. First, a subtle quad channel synchronization problem in a physical system was corrected with about three hours of prototype experimentation. Previously, a week of physical system testing had failed to even diagnose the problem. Second, an intermittent synchronization dropout by one particular computational channel was quickly diagnosed with the use of the prototype. Previously, two weeks of system testing had failed to disclose the source of the problem. Derivatives of a prototype have also been used successfully for real-time system execution monitors.

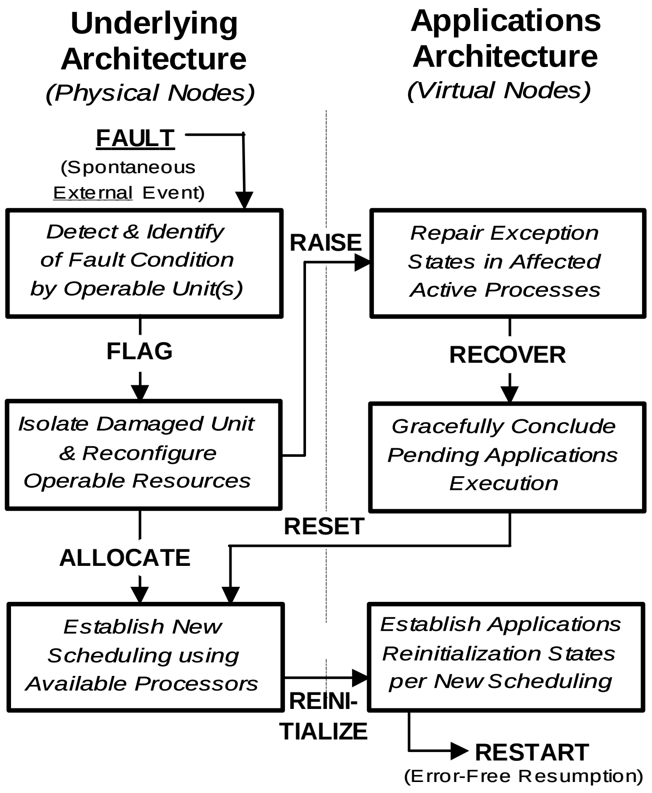

Underlying Architecture (Physical Nodes)
Applications Architecture (Virtual Nodes)
FAULT
(Spontaneous External Event)
Detect & Identify of Fault Condition by Operable Unit(s)
RAISE
Repair Exception States in Affected Active Processes
FLAG
RECOVER
Isolate Damaged Unit & Reconfigure Operable Resources
Gracefully Conclude Pending Applications Execution
RESET
ALLOCATE
Establish New Scheduling using Available Processors
Establish Applications Reinitialization States per New Scheduling
REINI-TIALIZE
RESTART
(Error-Free Resumption)

Figure 4. Prototype Execution of Fault Handling

REFERENCES

1. MULCARE, D.B. *et al.* 1984. Analytical design and assurance of digital flight control system structure. *AIAA Journal of Guidance, Control, and Dynamics*, May-June 1984.
2. MULCARE, D.B. 1993. Ada multitasking prototyping using higher-level statecharts, Tutorial at TRI-Ada '93.
3. MULCARE, D.B. 1996. System-level optimization of architectural performance under varying service demands. *9th International IEEE Symposium and Workshop on Engineering of Computer-Based Systems.*

Evaluation of Dependable Layered Systems with Fault Management Architecture

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

The need for a separate fault-management system, that is able to carry out both failure detection and reconfiguration, is becoming imperative due to the increasing complexity of fault-tolerant distributed applications. Such practice would eliminate the intricacies of the failure detection mechanisms from the application and would avoid repeating them in every program. The dependability of such an application depends on the interconnection of components in the fault-management system, management subsystem failures, delays incurred due to system reconfiguration and failure information propagation in the management architecture, as well as on the structure of the application itself. This position paper describes avenues for evaluating the dependability of a multi-layered service system that uses a separate fault-management architecture.

1. INTRODUCTION

Distributed software systems are usually structured in layers with some kind of user-interface tasks as the topmost layer, making requests to various layers of servers. Client server systems and Open distributed processing systems such as DCE, ANSA and CORBA are structured this way. [2, 1, 12] introduced an approach to express the layered failure and repair dependencies in these systems. However, the work done there is limited by the assumption of instantaneous perfect detection and reconfiguration, and independent failures and repairs.

This position paper describes avenues to incorporate the effect of fault management architecture in the dependability evaluation of layered systems. The fault management architecture influences the dependability in the following ways:

- management component failures and the interconnections among the management components affects the successful system recovery.
- delays for system reconfiguration and detection propagation in the management architecture increases the system downtime.

Our earlier work in [3] considered the delays for detection and reconfiguration by a separate detection architecture for layered systems. However, it was restricted to a particular detection architecture that would support full coverage of the failures by the system. If arbitrary connections among fault management components are considered, then it is possible that due to the loss of connectivity in the management architecture, the system may not be able to detect a failure and therefore would fail even if adequate redundancy exists. This issue has been addressed in this work that extends the work in [3] by taking into account arbitrary fault management architectures.

Other work analyzes the effect of software architecture (and not the management architecture) on reliability and is given by Trivedi *et. al*. [4, 5].

As in [3], this work considers only crash-stop failures, in which an entity becomes inactive after failure, and not to the other more complex failure modes such as Byzantine failure [9].

2. LAYERED MODELS CAPTURING FAILURE OCCURRENCE AND REPAIR BEHAVIOR

Figure 1 shows an example of a layered model using a notation proposed in [1, 2] with two groups of users (50 UserA users and 100 UserB users, which may be people at terminals or at PC workstations) accessing applications which in turn access back-end servers. The rectangles in this figure represent tasks (i.e. operating system processes) such as AppA or Server1 with entries, which are service handlers embedded in the task. For instance, eA-1, eB-1 are entries of task Server1. An arrow represents a request-reply interaction, such as an RPC. Processors are represented by ovals. The numbering #1, #2 on the request arcs indicate primary/backup choice for a service. Server1 is the primary server while Server2 is the backup, implying that if Server1 fails, both "serviceA" and "serviceB" would use Server2 until Server1 is working again. Failure and repair rates are provided for each component (either a task or a processor).

The special property of multi-layered client-server systems is that a failure of a task or a processor in one layer can cause many tasks, requiring its services, to fail, unless they have a backup. The model in Figure 1 captures such cascaded service operational dependencies.

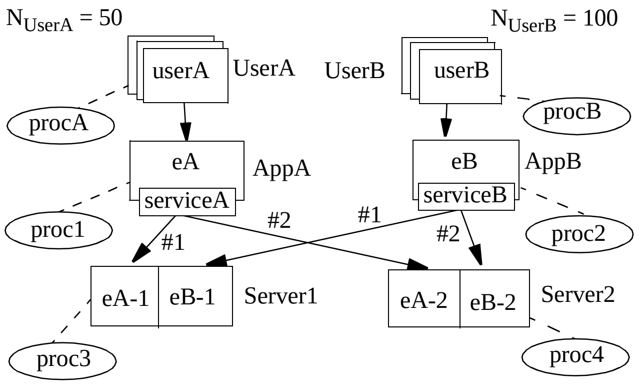


Figure 1. A layered model of a client-server system with two groups of users. Server2 is the backup of Server1.

In order to capture the effect of fault management architectures, the first step would be to describe the architecture in some relevant way. The next section introduces an architectural model to describe various fault management architectures.

3. FAULT MANAGEMENT ARCHITECTURE

The generic management components and their relationships can be depicted as in Figure 2, following [7]. Applications have embedded modules (Subagents) which may be configured to send heartbeat messages in response to timer interrupts (indicating they are alive) to a local Agent, or to a manager directly. A node may have an Agent task which monitors the operating system health status and all the processes in the node, and there may be one or more Manager tasks which collect status information from agents, make decisions, and issue notifications to reconfigure. Reconfiguration can be handled by a subagent (to cause a task or an ORB to retarget its requests) or an agent (to restart a task, or reboot a node altogether).

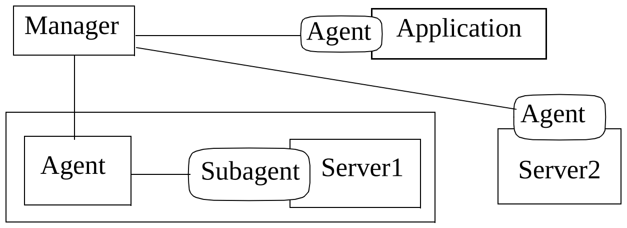


Figure 2. Management components and relationships

The agents and managers are described in this paper as if they are free-standing processes, even though in practice some of these components may be combined with other components in a dependability ORB [8], or an application management system [11].

Failures of system entities are detected by mechanisms such as heartbeats, timeouts on periodic polls, and timeouts on requests between application tasks. Heartbeat messages from an application task can be generated by a special heartbeat interrupt service routine which sends a message to a local agent or to a manager, every time an interrupt occurs, as long as the task has not crashed. Heartbeat messages for an entire node can be generated by an agent configured similarly, to show that the node is functioning; the agent could query the operating system health status before sending its message. Heartbeat information once collected can be propagated among the agents and managers to act as a basis for decisions, made by reconfiguration modules.

An entity that cannot initiate heartbeat messages may be able to respond to messages from an agent or manager; we can think of these as status polls. The responses give the same information as heartbeat messages. Polls to a node could be implemented as pings, for instance.

3.1. Reconfiguration

In this paper, we considered primary-backup replication for achieving fault-tolerance, i.e. the requests are routed to the backup server when the primary server fails, for masking the failure. This alternative targeting of requests is indicated in Figure 1 by showing an abstraction called "serviceA" and "serviceB" for the data access service required by the applications. This service has alternative request arrows attached to it, with labels “#n” showing the priority of the target. A request goes to the highest-priority available server, which is determined by a *reconfiguration decision*. In this work, the reconfiguration decision will be made by the management system, and will be conditioned by its knowledge of status of system components. It can respond not only to processor failures but also to software failures (task crashes and operating system crashes). Network components can be included in the model as well. A reconfiguration strategy different from the alternative targeting of requests to the highest-priority available server can also be analyzed. For instance, a strategy which involves distributing the workload equally among the available servers can also be considered.

3.2. Management Architecture

The architecture model described here will be called MAMA, ***M***odel for ***A***vailability ***M***anagement ***A***rchitectures. The model has four types of components: application tasks (which may include subagent modules), agent tasks, manager tasks, and the processors they all run on (network failures are for the time being ignored). There are three types of connectors: *alive-watch*, *status-watch* and *notify*. These connectors are typed according to the information they convey, in a way which supports the analysis of knowledge of the system status at different points in the management system.

Components have *ports* which are attached to connectors in certain *roles*. The roles are defined as part of the connector type. The connector types and the roles they support are:

- *Alive-watch* connectors, with roles *monitor* and *monitored*. They only convey data to detect crash failure of the component in the *monitored* role, to the component in the *monitor* role. A typical example is a connector to a single heartbeat source.

- *Status-watch* connectors, also with roles *monitor* and *monitored*. They may convey the same data about the *monitored* component, but also propagate data about the status of other components to the component in the *monitor* role. A typical example is a connector to a node agent, conveying full information on the node status, including its own status.

- *Notify* connectors, with roles *subscriber* and *notifier.* The component in the *notifier* role propagates status data that it has received to a component in a *subscriber* role, however it does not include data on its own status.

Manager and Agent tasks can be connected in any role; an Application task can be connected in the roles *monitored*, or *subscriber.* A Processor is a composite component that contains a cluster of tasks that execute there. If the processor fails, all its enclosed tasks fail. The Processor can only be connected in the *monitored* role to an *alive-watch* connector (which might convey a ping, for example).

Upon occurrence of a failure or repair of a task or a processor, the occurrence is first captured via *alive-watch* or *status-watch* connections and the information propagates through *status-watch* and *notify* connections, to managers which initiate system reconfiguration. Reconfiguration commands are sent by *notify* connections. Cycles may occur in the architecture; we assume that the information flow is managed so as to not cycle. In this work, we note that if a task watches a remote task, then it also has to watch the processor executing the remote task, in order to distinguish between the processor failure and the task failure.

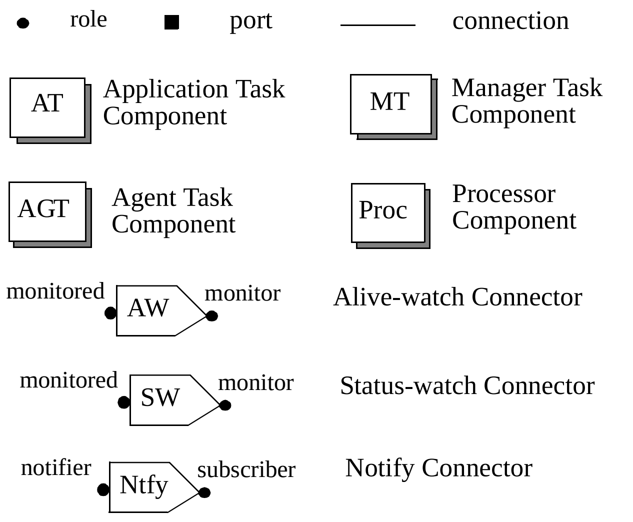


Figure 3. MAMA notations. The graphical notation of components, ports, connectors and roles are taken from [6].

Figure 3 shows a graphical notation for various types of components, ports, connectors and roles based on the customized UML notation for conceptual architecture as defined in [6]. The component types and connector types will be shown as classes in this work. In order to avoid cluttering in the MAMA diagrams, the role names such as *monitor*, *monitored*, *notifier* and *subscriber* have been omitted from them.

Figure 4 shows a centralized management architecture, in MAMA notation, for the system of Figure 1. Manager1 is introduced here as the central manager task that collects status information from the agents ag1-ag4 running on the processors proc1-proc4. The application tasks AppA and AppB are also subscribers for the notifications from Manager1, which control retargeting of requests to the Servers.

Other management architectures (such as “distributed”, “hierarchical”, “general network” architectures as described in [10]) containing several managers and agents with multiple detection paths can be modeled and analyzed.

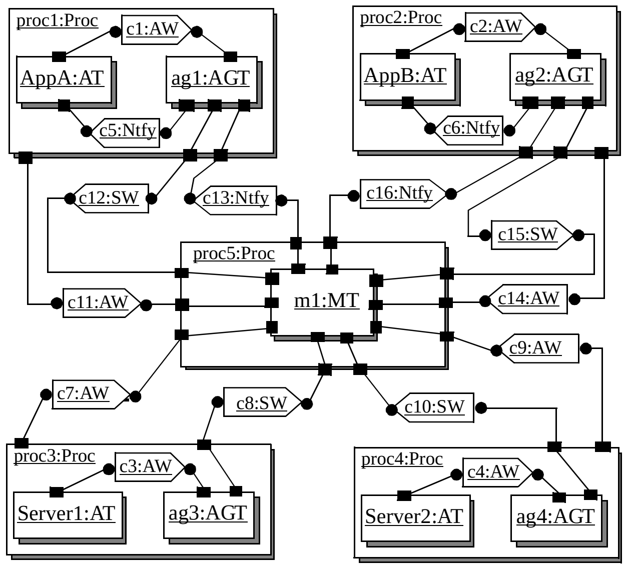


Figure 4. MAMA Model of a centralized management architecture for the system in Figure 1.

4. DETECTION AND RECONFIGURATION

The detection and reconfiguration parameters to be provided in the model are as follows:

- delay of detection propagation from one component to another in the fault management architecture, i.e. a delay parameter associated to each connector in the management architecture. It can be computed from the heartbeat or polling interval for alive-watch and status-watch connectors or from the notification delay for notify connectors.
- delay required by a management component for analyzing and forwarding data.
- restart delay of each application task.
- reconfiguration delay for each service request that has alternative targets.
- probability of successful local recovery of an application task, within a given time interval.

5. MODEL SOLUTION

Let us define a *system state* to be a vector of the states of the fault management components and the components in the software architecture.

The dependability measures for the layered model are then obtained as follows:

1. Construct a continuous-time Markov chain that describes the system changes due to failure and repair and includes the reachable set of system states. It incorporates the detection and recovery behavior of the system in between every two system states.
2. Associate the reward rate equal to 1 to each state of the resulting Markov chain that represents a “working” configuration of the system. Otherwise associate a reward rate of zero with the state. The importance of the fault management architecture is that its failures can modify the system’s ability to reach “working” states.
3. Solve the resulting Markov reward model to obtain the desired measures. For example, we can obtain the steady-state availability of the system by summing up the probabilities of all the states that has reward rate equal to 1.

Other interesting measures might be the mean throughput of the system, mean response time for a client, mean outage time for a client of the system etc.

Solvers for these (and more general) measures are presently being developed.

6. CONCLUSION

An approach to incorporate the effect of fault management architectures, that does both failure detection and reconfiguration, in the dependability evaluation of layered systems has been considered. The value of including the management architecture in the analysis is first to account for failures and repairs of managers and agents, and second to evaluate limitations in the fault management architecture.

Current work is to develop a model for capturing the effect of failures and repairs of the management subnet on system dependability measures. The key question to be answered is the complexity of the solution to determine the state probabilities.

7. REFERENCES

[1] Das, O., and Woodside, C.M. The Fault-tolerant layered queueing network model for performability of

distributed systems. IEEE International Computer Performance and Dependability Symposium (IPDS'98), Sept. 1998, pp. 132-141.

[2] Das, O., and Woodside, C.M. Evaluating layered distributed software systems with fault-tolerant features. Performance Evaluation, 45 (1), May 2001, pp. 57-76.

[3] Das, O., and Woodside, C.M. Failure detection and recovery modelling for multi-layered service systems. Fifth International Workshop on Performability Modeling of Computer and Communication Systems, Erlangen, Germany, Sept. 2001, pp. 131-135.

[4] Gokhale, S.S., Wong, W. E., Trivedi, K. S. and Horgan, J. R. An analytical approach to architecture-based software reliability prediction. IEEE International Computer Performance and Dependability Symposium (IPDS'98), Sept. 1998, pp. 13-22.

[5] Goseva-Popstojanova, K. and Trivedi, K. S. Architecture-based approach to reliability assessment of software systems. Performance Evaluation, 45 (2-3), 2001, pp. 179-204.

[6] Hofmeister, C., Nord, R., and Soni, D. Applied Software Architecture. Chapter 4, Addison-Wesley, 2000.

[7] Kreger, H. Java management extensions for application management. IBM Systems Journal, 40(1), 2001, pp. 104-129.

[8] Moser, L.E., Melliar-Smith, P.M., and Narasimhan, P. A fault tolerance framework for CORBA. Proc. of 29th Annual Int. Symposium on Fault-Tolerant Computing, 1998, pp. 150-157.

[9] Schneider, F.B. What good are models and what models are good. Sape Mullender, Editor, Distributed Systems, ACM Press, 1993.

[10] Stamatelopoulos, F., Roussopoulos, N. and Maglaris, B. Using a DBMS for hierarchical network management. Engineer Conference, NETWORLD+INTEROP'95, March 1995.

[11] Tivoli Systems Inc., 9442 Capital of Texas Highway North, Arboretum Plaza One, Austin, Texas. See http://www.tivoli.com.

[12] Woodside, C.M. Performability modelling for multi-layered service systems. Third International Workshop on Performability Modeling of Computer and Communication Systems, Bloomingdale, Illinois, Sept. 1996.

A Conflict Resolution Control Architecture For Self-Adaptive Software

N. Badr
School of Computing and Mathematical Science,
Liverpool John Moores University,
Byrom Street, Liverpool
L3 3AF, UK
cmsnbadr@livjm.ac.uk

D. Reilly
School of Computing and Mathematical Science,
Liverpool John Moores University,
Byrom Street, Liverpool
L3 3AF, UK
d.reilly@livjm.ac.uk

A.TalebBendiab
School of Computing and Mathematical Science,
Liverpool John Moores University,
Byrom Street, Liverpool
L3 3AF, UK
A.Talebbendiab@livjm.ac.uk

ABSTRACT

An essential feature of dependable software is its adaptive capability to respond to changes that occur in its operating environment through the dynamic transformation and reconfiguration of its components and/or services. Such adaptive capability is often a design aspect derived from the software architecture model, which describes the software components and their interactions, the properties and policies that regulate the composition of the components and norms that limit the allowable systems adaptation operations. Research in reflective middleware architectures and policy-based distributed systems management has focused on the use of managerial or meta-level protocols to attain reactive adaptive behaviour. However, reflective and policy-based management approaches alone cannot address all of the needs of self-adaptive software due to their inability to maintain a faithful runtime model of the system. This paper considers the development of control architecture for self-adaptive software, which combines conflict resolution and control strategies to resolve runtime conflicts. In particular, the paper describes a prototype service-based architecture, implemented using Java and Jini technologies, which provides runtime monitoring and conflict resolution to support software self-adaptation.
.

1.Introduction

Self-adaptive software can be seen as a new architecture style, which extends the controller concepts to adapt the structural configuration and dynamic behaviour of a system. Structural components can evaluate their behaviour and environment against their specified goals with capabilities to revise their structure and behaviour accordingly. *Laddaga* [7] defines self-adaptive software as:

"*Software that evaluates and changes its own behaviour when the evaluation indicates that it has not accomplishing what it is intended to do, or when better functionality or performance is possible*".

Such a software architecture style presents an attractive concept to developing self-governing software, which fully or partially accommodates its own management and adaptation activities. Research in this area has adopted control engineering concepts, as typified by *Osterweil* [9] who presents an architecture, which uses a controller with a well-specified control function with *feedforward* and *feedback* loops to enable a target system to be monitored to regulate its operation in accordance with its given control model. *Osterweil* [9] describes the delegation of the responsibility for testing and evaluation of software applications from humans onto automated tools and processes, advocating the automation of the continuous self-evaluation processes.

However, there are further issues to be addressed in order to achieve self-adaptation, such as: reasoning, control and decision-making to assess the gap between a given software operational model and its requirements, and the use of appropriate strategies for conflict resolution. This paper argues, that during any software self-adaptation process, it is likely that autonomous changes may lead to execution errors and software integrity conflicts. Thus the self-adaptation of distributed software requires control and decision-making to support the monitoring, detection and resolution of conflicts, which may occur at runtime.

The remainder of the paper provides an overview of our "work in progress" concerning the development of a service-based architecture that uses conflict resolution to achieve self-adaptation. The paper is structured as follows: section 2 provides a brief review of self-adaptive software and conflict resolution strategies. Section 3 describes the prototype conflict resolution control architecture and its constituent services. Section 4 briefly describes a case study, which illustrates how the architecture achieves runtime self-adaptation. Finally, Section 5 draws conclusions and mentions future work.

2. Background

The control theory based paradigm provides a framework for designing software that supports self-control during the operation of the software. The self-controlling software model supports three levels of control: feedback, adaptation, and reconfiguration [6]. *Meng* [8] proposed a control system for self-adaptive software based on a descriptive model of a self-adaptive control system, which employs the control system concepts of feedforward and feedback. For example, if a self-adaptive software system consists of two components the feedforward process can provide specifications of the software and its predictability and the feedback process can gather and measure the software's environmental attributes.

Central to this paradigm are the decision-making and delegation strategies that are used to resolve conflicts, as considered by *Barber et al* [2] who discuss different decision-making strategies required in conflict resolution. *Barber et al* describe *negotiation* as the most popular strategy, but also consider *arbitration*, *mediation* and/or *voting* as viable strategies in agent-based systems. *Adler et al.* [1] describe the "Independence" strategy, which regards *self-modification* as a simple and effective resolution strategy for use in agent-based systems, which is used when an agent detects a conflict but does not wish to interact with other agents to solve the conflict, as the agent would rather resolve the conflict itself. Other approaches by *Williams and Taleb-Bendiab* [13] illustrate the use of meta-languages to support software agent composition and the runtime reconfiguration of middleware services.

Recently there has been an increasing research trend in the development of *self-healing* software, facilitated through innovations in operating systems [5] and *reflective middleware* architectures through which structural and/or behavioural models can be modified at runtime. *Robertson et al* [12] indicate that whilst reflective architectures share similar aims with self-adaptation architectures, they differ in that self-adaptive architectures generate runtime evaluators to check the deviation of the state of the program against some measure. A control regime is then used to compute the distance between the current state and the goal state in order to maintain stability and robustness. The control regime makes use of sensor and actuator concepts to feedforward and feedback the systems states enabling the software management layer to reconfigure and switch the control regime itself to suite the systems requirements.

The general model of self-adaptive software can be viewed from many different aspects, which take the architectural model as a parameter in the monitoring and repairing framework to allow the monitoring mechanism to match both properties of interest and adaptation operators at runtime [3]. Previous work by *Cooper and Taleb-Bendiab* [4], based on the same theme as the current research described herein, focused on a heuristic-based approach to support software agent self-adaptation. The current research extends this previous work by concentrating on automated self-adaptation that can be applied at runtime.

3. Service-Oriented Conflict Resolution Control Architecture

The development of dependable distributed system is hindered by the conflicts and faults that may occur at runtime. With being the case, our approach is based on a control mechanism which monitors behaviour, detects and identifies conflicts and formulates remedial action in the form of a resolution strategy. A *service-oriented* approach was adopted to develop the control mechanism and overall service-based architecture, as shown in Figure 1 overleaf.

The service-based architecture achieves self-adaptation by detecting, identifying, and resolving conflicts that occur at runtime, After detection, a conflict is identified and categorized according to its type before a resolution strategy is used to *minimize* the conflict. A monitor element then provides feedback to guide the conflict resolution tasks, which are used to implement the conflict resolution process. The prototype architecture, used to implement the conflict resolution and control element (figure 1) is based on Java and Jini middleware[1] technologies to provide the following services.

❖ Monitor: makes use of a set of control rules against which behaviour is monitored to detect conflicts.

❖ Diagnostic: the execution of a control rule implies a conflict, which activates the diagnosis services that results in:

- Identification of the part of the control rule that raised the conflict.
- Identification of the cause of the conflict through the examination of service attributes and method invocations using Java's reflection API (`java.lang.reflect`).
- Classification of the type of conflict, which provides the basis for the selection of a resolution solution strategy.

❖ Notification: makes use of Jini's remote event mechanism to notify clients when conflict resolution solutions become available.

❖ Control Rules: serve as the basis for the previous monitoring and diagnose services. They consist of a number of rules and gates that execute when a conflict is detected, which in practice execute when a service method is invoked. This in turn may result in the firing of a remote event to notify of the availability of a resolution solution strategy.

❖ Exception Handling: makes use of Java's exception handling facilities to catch exceptions, which are thrown when a control rule executes due to some kind of fault/attack that cannot be solved. Exceptions are dealt with according to priority, which may be low, intermediate or high to accommodate varying degrees of fault tolerance.

[1] Jini is a Java based middleware technology developed by Sun Microsystems Inc.<http://www.sun.com/jini/specs>

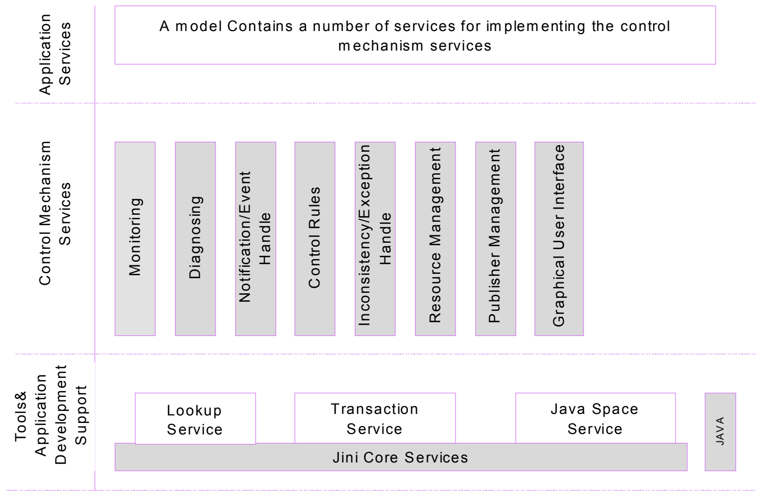

Application Services
A model Contains a number of services for implementing the control mechanism services
Control Mechanism Services
Monitoring
Diagnosing
Notification/Event Handle
Control Rules
Inconsistency/Exception Handle
Resource Management
Publisher Management
Graphical User Interface
Tools& Application Development Support
Lookup Service
Transaction Service
Java Space Service
JAVA
Jini Core Services

Figure 1: Self-Adaptive Control Architecture

Figure 2 illustrates a flow chart of the reasoning that takes place in a typical conflict resolution solution strategy. The flowchart begins with a client request for an application service and if the request succeeds, then no conflict has occurred so the client responds. If a conflict does occur an attempt is made to provide the client with an alternative service, which performs the same function. If this also results in a conflict then the client may choose a renewable notify option, whereby the client is notified when a suitable service becomes available or the client may renew the notification period when its lease expired.

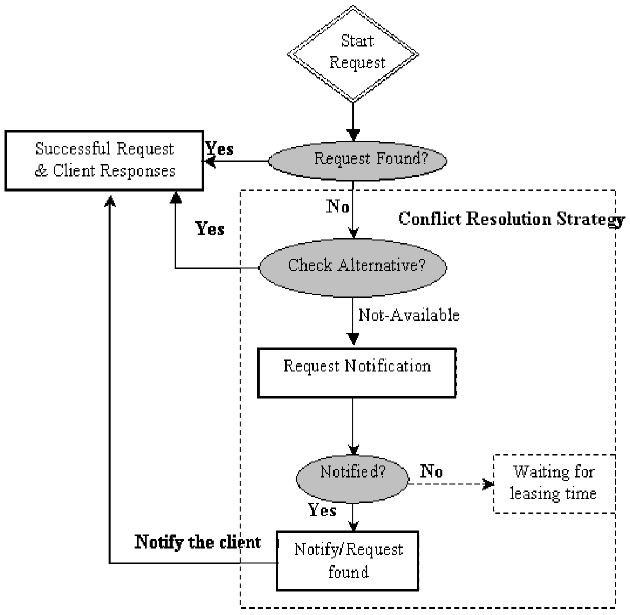


Figure 2: Example of Control Resolution Strategy

4. Case Study

We illustrate the architectures capabilities to accommodate self-adaptation through an industrial case-study, which involves a dependable software system developed out of the EmergeITS project. EmergeITS is concerned with the development of an adaptable software architecture to provide In-Vehicle Telematics Systems (IVTS) capabilities to emergency service response teams [11]. Figure 3, provides an overview of the EmergeITS services, which use Jini services to provide: remote hardware control, remote database access (through Java Servlets and XML documents) and mobile communication management capabilities. The IVTS Manager oversees the overall operation of the system and provides capabilities to add additional services, as well as limited control and adaptation facilities. IVTS Clients (typically remote vehicles) may request the use of any service that the IVTS Manager has to offer.

One crucial service for the IVTS architecture is the *3-in-1 phone* service (*palowireless2001*) [10], which allows a portable wireless phone or PDA to be used as a cellular phone, WAP device or walkie-talkie. An IVTS Client may request the use of a 3-in-1-phone service whenever a physical device such as a mobile phone or PDA is to be used from a moving or stationary vehicle. The client may request to use the device in one of the 3 different modes for which the 3-in-1 phone service must adapt accordingly. The 3-in-1 phone service essentially implements the conflict resolution and control strategies by examining: 1) user operation mode requested, 2) client location and 3) bearer service location and availability (e.g. BTCellNet, , GSM or Tetra[3]) and each of these three parameters manifest as remote method invocations made on the 3-in-1 phone service. The outcome of the initial resolution strategy may result in IVTS client notification that the request was successful, or else that a further conflict occurred, leading to further control rules

[2] EmergeITS is a collaborative project involving our own research group within the School of Computing and Mathematical Sciences at Liverpool John Moores University and the Merseyside Fire Service <http://www.cms.livjm.ac.uk/emereits>

[3] Tetra is a digital network under consideration by Police and Fire emergency services in the UK

executing for which a conflict resolution must be sought, such as the initiation of a more comprehensive search for a bearer service to resolve the conflict. Eventually a situation is reached whereby the system has attained a configuration that best meets the needs of the initial request.

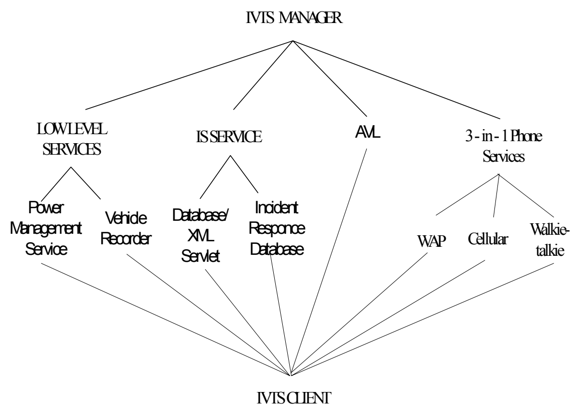


Figure 3: IVTS Services

5. Conclusion and Future Work

In this paper we have described a prototype service-based architecture, based on Java and Jini technologies, that uses conflict resolution and control strategies to detect, identify and resolve conflicts that occur at runtime. We intend to extend our architecture, in future related research, by considering the negotiation aspects in the control decision making of self-adaptation. It is anticipated that this will increase the flexibility of our architecture, which we intend to evaluate through further case studies from our complementary group research area of intelligent-networked-vehicles

References

[1] Adler, M., *et al. Conflict- Resolution Strategies for Nonhierarchical Distributed Agents*. in *In Distributed Artificial Intelligent* ||,. 1989. London.

[2] Barber, K.S., T.H.liu, and D.C.Han. *Strategic Decision-Making for Conflict resolution in Dynamic Organized Multi-Agent Systems*. in *GDN 2000 PROGRAM*. 2000.

[3] Cheng, S.W., *et al.*, *Using Architectural Style as a Basis for System Self-repair*, . 2002.

[4] Cooper, S. and A.Taleb-bendiab. *A High Level ControlMechanism For Managing Conflict Resolution In Concurrent Product Design*. *In Proceedings of the fourth ISPE International conference on Concurrent Engineering :Research and Application (CE97)*. 1997

[5] eLizaProject, http://www-1.ibm.com/servers/introducing/eLiza .

[6] Kokar, M., K. Baslawski, and Y. Eracar, *Control Theory-Based Foundation of Self- Controlling Software.* IEEE Intelligent Systems, 1999: p. 37-45.

[7] Laddaga R. *Active Software*. in *First International Workshop on Self-Adaptive Software, (IWSAS2000)*. 2000.

[8] Meng, A.C. *On Evaluation Self-Adaptive Software*. in *First International Workshop on Self-Adaptive Software, (IWSAS2000), **April 2000.*** 2000.

[9] Osterweil, L.J. and Clarke, L.A. *Continuous Self-Evaluation for the Self-Improvement of Software*. in " *First International Workshop on Self-Adaptive Software, (IWSAS2000)*. 2000.

[10] palowireless:Bluetooth Resource Center

[11] Reilly, D.and A. Taleb-Bendaib, *A Service Based architecture for in-Vehicle Telematics Systems* Submitted in "A Special Issue of CERA Journal: A. Complex Systems Perspective on Concurrent Engineering", 2002

[12] Robertson, P., Laddaga, R., and Shrobe H.. *Introduction: the First International Workshop On Self-Adaptive Software*. in *First International Workshop on Self-Adaptive Software, (IWSAS2000)*. 2000.

[13] Williams, M. and A.Taleb-Bendiab. *A Toolset for Architecture Independent, Reconfigurable Multi-Agent systems*. in *First International Workshop on Mobile Agents*. 1998.

Improving the availability of web services [*]

D. Cotroneo[1,2], M. Gargiulo[2], S. Russo[1,2], G. Ventre[1,2]
[1]Universitá degli Studi di Napoli “Federico II"
Dipartimento di Informatica e Sistemistica
Via Claudio 21, 80125 - Napoli, ITALY
[2]Consorzio C.I.N.I.
Laboratorio ITEM
Via Diocleziano 328, 80124 - Napoli, ITALY
{cotroneo, sterusso, giorgio}@unina.it
mauro.gargiulo@napoli.consorzio-cini.it

ABSTRACT

In order to maintain the popularity and reputation of a web site, the quality of service perceived by users, especially the service availability, is a success factor. A service that is frequently unavailable may have negative effects on the reputation of the service provider, or result in loss of business opportunities. From the user's perspective, a service that exhibits poor quality is virtually equivalent to an unavailable service. In this work, we present the overall architecture and the evaluation of a middleware infrastructure which provides quality-of-service differentiation among classes of communication-bound processes. By communication-bound processes we mean processes whose activity is typically dominated by network communication, e.g. a video server. The proposed architecture supports different classes of service, each with different quality attributes concerning the network data delivery performance. In particular, the architecture is able to provide a class of service, namely *guaranteed service class*, which is suitable for increasing the service availability for a group of premium users, especially in overloaded servers (in absence of external faults).

Keywords

Class-of-Service, Web-service, Availability, Real-time Operating Systems

1. INTRODUCTION

The Internet world is moving toward a scenario where users and applications have very diverse service expectations, making the current best-effort model inadequate and limiting. In fact, new web applications demand for delivery of multimedia data in real-time (e.g. streaming stored video and audio), and the information transfer via the Internet is becoming one of the principal paradigm for business: electronic sales, banking, finance, collaborative work, are examples of this. In this scenario, in order to maintain the popularity and reputation of a web site, the quality of service perceived by users, especially the service availability, is a success factor [5]. The principal QoS attributes that users perceive include those related to the service availability and timeliness: a service that is frequently unavailable may have negative effects on the reputation of the service provider, resulting in loss of business opportunities. From the user's perspective, a service that exhibits poor quality is virtually equivalent to an unavailable service.

The performance perceived by the users of a Web service depends on the network infrastructure (possibly QoS-enabled) but especially on the management of the servers' resources [13]. It is thus desirable that network servers (e.g., Web, Video on Demand, and FTP servers) should be able to differentiate their services in a variety of classes, replacing the current simple best-effort paradigm. This leads to a model in which applications and users are treated differently, in a way that best meets their quality and pricing constrains. This paper presents the overall architecture and the experimental evaluation of an operating system extension for service differentiation of communication-bound processes. The architecture provides server application developers with a communication library (similar to the standard socket), named cosSocket (*class of service-enabled Socket*), which is able to realize different classes of service using features of a real-time operating system. A service differentiation scheme can be applied to different applications, or to different users in the context of the same application. Service differentiation is obtained by assigning different CPU time-slices to applications I/O tasks. The underlying idea is that it is possible to decrease time-slice assigned to a given process in order to reduce its communication throughput, freeing server's resources in favor of processes with a better class of service. Real-time features are achieved through a Rate Monotonic Algorithm for CPU scheduling, included

[*]This work has been performed under the financial support of Italian Ministry of Education, University and Research (MIUR) under grant “LABNET2”. The work of D. Cotroneo and S. Russo has also been supported by MIUR within the project “MUSIQUE: Infrastructure for QoS in Web Multimedia Services with Heterogeneous Access”.

.

in the TimeSys Linux/RT kernel [*www.timesys.com*], which also offers a complete support for POSIX threads.

The paper is organized as follows. In section 2 we introduce the web service correctness and related problems. The failure mode assumptions we adopted is presented in section 3. In section 4 we describe the overall architecture; experimental results are provided in section 5. Section 6 discusses related work in this field. Finally, section 7 provides some concluding remarks related to the obtained results, along with information on future work.

2. WEB SERVICE CORRECTNESS

Before illustrating the design and the implementation of the proposed middleware infrastructure, it is worth clarifying the definition of the web service availability, starting from the position stated in [14] by D. Powell. Availability deals with the readiness for correct service. In particular, it is a function $A(t)$, which is the probability that the system is operational (i.e., delivers the correct service) at instant of time t. This function quantifies the alternation between deliveries of correct service and incorrect service. A system can fail to deliver a correct service due to the following reasons:

- the presence of faults, caused by system errors;
- the presence of overloading condition, i.e. the server is so much busy that it is not able to deliver a correct service.

Throughout this paper we focus only on system failures, stemming from overload conditions. This kind of failures is strictly tied with the delivered quality of service because, if the QoS falls down under a certain threshold, the service can be considered unavailable. The architecture we propose provides an efficient and flexible resources management strategy, which aims at improving the quality of the delivered service, reducing the QoS degradation perceived by some premium users. The result is a higher probability of delivering to these users a correct service, improving the system availability. The architecture does not prevent system from hardware/software faults, hence it does not guarantee the service availability. In order to achieve this further goal, a redundant scheme has to be also implemented, as described in section 7.

As stated in [14] and in [4], in order to analyze the availability of a system it is essential to clarify what does correct service mean. Starting from definition given by Powell, the service delivered by a system can be defined in terms of a sequence of service items $s_i, i = 1, 2....$, each characterized by a tuple $< vs_i, ts_i >$, where vs_i is the value or content of service item s_i and ts_i is the time or instant of observation of service item s_i. Assuming the presence of an omniscient observer that has a complete knowledge of the specified sequence of service items the system should deliver, a service s_i is defined correct if:

$$(vs_i \in SV_i)\, and\, (ts_i \in ST_i)$$

where SV_i and ST_i are respectively the specified sets of values and times for service item s_i. For a general system, SV_i and ST_i are functions of the sequence of system inputs. As far as modern web-based systems with QoS constraints are concerned, this definition is certainly suitable, but sets SV_i and ST_i have to be extended.

Indeed, modern web-based systems are implemented over a QoS-enabled network. In this context, the term QoS is related to the quality of communication service, such as a certain value of packet loss, latency, jitter, and assured bandwidth, appearing at the communication endpoints like a point-to-point connection or a virtual “leased line” with the requested quality attributes. In this scenario, it should be possible to provide the same service with different quality attributes to several classes of users. In this way, the correctness of a service also depends on the specified class of users which request it. We can thus define the correctness of a web service, for a certain class j of users, as:

$$(vs_i \in SV^*_{i,j})\, and\, (ts_i \in ST^*_{i,j})$$

where:

$$SV^*_{i,j} = f(SV_i, CU_j)$$
$$ST^*_{i,j} = f(ST_i, CU_j)$$

and where CU_j represents the class j to which the user belongs. It is thus desirable to have an architecture in which applications and users are treated differently, in a way that best meets their quality and pricing constrains. The proposed middleware architecture, using features of a real-time operating system, is able to realize different classes of service inside the web server too. A service differentiation scheme is provided to different applications, or to different users in the context of the same application.

3. FAILURE MODE ASSUMPTIONS

This section gives a formal definition of the failure modes of a web server we adopted. By web server it is meant a server, such as multimedia or HTTP server, that provides its services via a web infrastructure. According to [14], a failure mode is defined in terms of an assertion on the sequence of value-time tuples that a server is supposed to deliver. Let us assume the following class of users exist: C_n(normal user), C_m (medium user), C_p (premium users). Assertions may be defined in the value domain and in the time domain. Effects of value errors are not considered afterward. As already mentioned, we investigate failures caused by server overload conditions, i.e. the server process or host is too busy for delivering the correct service to a certain class of user.

3.1 Timing errors assertion

These assertions are the most important in the considered context.

- No timing errors can occur:
  $\tau_{none} := \forall i, \forall j\; ts_i \in ST_{i,j}$
- Omission errors can occur:
  $\tau_O := \forall i, \forall j\; (ts_i \in ST_{i,j})\, or\, (ts_i = \infty)$
- Late timing errors can occur:
  $\tau_L := \forall i\; (\forall j,\;\; ts_i \in ST_{i,j})\, or\, [\exists j \in \{m,p\} : (ts_i > maxTime_j)]$

The omission error assertion depicts a fail silent behaviour: in such a situation if the system does not supply the service s_i in ST_{ij} it will never supply it at all. Similary, a late timing error can occur if a service item s_i is delivered at a medium or premium user after a threshold, named $maxTime$, depending on user requirements.

It should be emphasized that late timing errors can occur only when medium (C_m) and premium users (C_p) request the service. This means that a service delivered to users belonging to class C_n, is correct even though it is delayed for a time greater than $maxTime$.

4. OVERALL SYSTEM ARCHITECTURE

As mentioned, we focus on Internet-based data delivery servers (Web, FTP, Video on Demand servers). For these kinds of servers, controlling I/O activities is essential to achieve a pre-determined behaviour. We propose an architecture which provides differentiated communication services according to a number of service classes. The real-time scheduler assigns to each service class a CPU amount depending on its service level. By doing so, it is possible to schedule processes in a deterministic way. However, assigning a service level to the entire process does not ensure real-time communication. In fact, the performance of a communication-bound process mainly depends on the scheduling of its I/O tasks, as indicated in [7, 8]. The architecture we propose is in charge of managing I/O activities of all processes residing on the end-system. In fact, process I/O tasks consist of a sequence of system call invocations which require the execution of an operating system thread serving the request. Our strategy relies on the capability of controlling the number of system calls issued for requesting I/O tasks.

The proposed architecture is able to completely separate I/O from the CPU activities, by providing application developers with a new communication library (cosSocket) similar to the standard TCP/IP socket library. Once separated these activities, I/O tasks can be scheduled by the real-time kernel. As for the design methodology, we adopted an object-oriented approach, namely *Concurrent Object Modeling and Architectural Design Method* (COMET), particularly suited for designing concurrent and real-time distributed systems [9]. The static model for the overall architecture is depicted in figure 1.

As shown in figure 1, user applications can create one or more instances of class cosSocket. From user point of view, such a class is able to perform I/O operations with a specified quality of service. The cosManager, which inherits from a POSIX thread class, is in charge of handling an instance of cosSocket. In other words, by means of cosSocket each application delegates, or defers, all the I/O activities to an instance of cosManager. I/O operations of cosManager are then performed using standard socket libraries. The activities of cosManager, which are mainly I/O calls, are optimized and then scheduled by the real-time kernel by means of the cosDaemon. This daemon is the only architecture's entity, capable of using real-time features in order to control all cosManagers present on the end-system. The cosDaemon is also in charge of implement admission control policies for guaranteed services.

We defined a class service model which consists of two kinds of service classes: Adaptive and Guaranteed. They are presented in the following subsections.

4.1 Adaptive class service

By adaptive we mean a service class that can be requested without any admission control mechanism [2]. According to this class definition, we allocate CPU shares in a weighted way. This means that we set preliminary n weights, $W_1 < W_2 < ... < W_n$, associated to each of n classes (class n has the highest priority). The adaptive service does not provide hard guarantee on the effective throughput of each process; however, it allows to define several classes, providing a guaranteed differentiation between them on the basis of assigned weight.

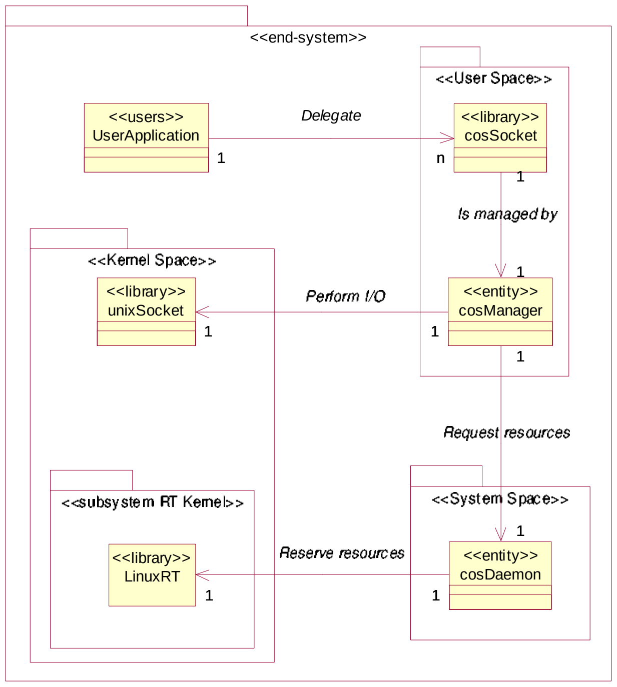


Figure 1: COMET Static Diagram of the proposed architecture.

4.2 Guaranteed service class

By guaranteed service class we mean a class subject to an admission control policy. In this case, each instance requires a specific throughput. The request can be accepted or rejected according to the specific policies implemented in the admission control module. If accepted, the service has to be guaranteed by the system during all its life cycle. This kind of service is particularly suitable for applications that require a constant throughput (e.g. multimedia applications) or for satisfying a group of premium users, leaving the service always available to them independently of the system workload (in absence of external faults).

The mechanism for providing Guaranteed services class is implemented by a self-regulating utilization control loop. The throughput control loop determines the CPU amount necessary for obtaining a Constant Bit Rate (CBR). Let y_0 the desired throughput, and y the current throughput obtained by the cosManager. For the sake of simplicity, the cosManager was modeled like a "black box". The value e is the "throughput error", $e = y_0 - y$. The cosDaemon, which acts as the controller, samples the current throughput y, and computes the corresponding error e at fixed time intervals, then produces an output, u, that regulates the CPU

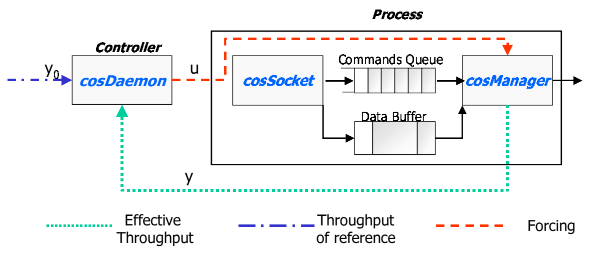


Figure 2: Self-regulating control loop for providing Guaranteed service.

to be assigned to the cosManager. We used a proportional-integral (PI) controller in our loop. The controller produces an output that is proportional to the last error and to the sum of the previous m errors. At each sampling time the controller performs the following computation, diminishing the medium quadratic error: $u = u + k * e$ where k is a constant. The adopted scheme is illustrated in figure 2.

4.3 Implementation models

As far as implementation is concerned, two main issues have been addressed: the synchronization mechanism between cosSocket and cosManager, and data buffer management. The solution of these problems resulted in three different implementation models, as shown in [6]:

- *Synchronous model*;
- *Asynchronous model*;
- *Asynchronous Aggregated model*.

We implemented all the three models and evaluated them in order to investigate which of ones is best suited for the considered applications. Experimental results, reported in [6], show how best performance are obtained with the last one. Therefore we adopted the Asynchronous Aggregated model, and results discussed in section 5 refer to this model.

5. AVAILABILITY EXPERIMENTS

In this section we present measurements which aim to demonstrate how the proposed middleware architecture is able to improve availability of service delivered to the premium user class.

The testbed used is composed of three different kinds of COTS PC, as described in the following:

- One HTTP server (named Tigri): is a Pentium III at 600 Mhz with 256 Mb of RAM and Linux/RT by Timesys installed.
- One HTTP client (named Eufrate): is a Pentium III at 600 Mhz with 256 Mb of RAM and Linux/RT by Timesys installed.
- Five HTTP clients: Celeron 700 Mhz with 128 Mb of RAM and Windows 2000 Professional installed.

All PCs are on the same LAN at 100 Mbit/s full switched.

On the Server Tigri we installed our architecture. We also used a simple HTTP-server application which replies to the client requests on different port numbers, one for each class

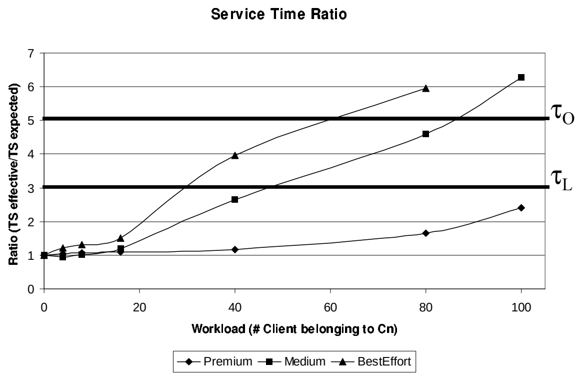


Figure 3: Performance measurements in presence of variable workload.

of service, as previously described: C_n(normal user), C_m (medium user), C_p (premium users).

The Windows clients are used only to generate the workload on the target server. To this aim, each Client run one or more instances of a http-client application requiring a file transfer service belonging to the C_n class. The same HTTP-client was also used on the Linux client machine, where measures are taken, in order to require a new service belonging each time to one of the three possible classes. We used a Linux client with the real-time extension in order to obtain a high-precision timer. All requests issued concern the same service: the transfer of a file of 10 Mbyte. The Windows clients repeat such a request in a cyclic way, to obtain a constant workload for all tests. Each test was repeated five times and the final result is the average value.

Tests were performed by evaluating the bandwidth obtained by the Linux client and the correspondent service time, with different workload due to the variable number of C_n connections. The C_n and the C_m classes are handled in a adaptive way, so in these cases the architecture do not provide a guaranteed quality of the service delivered, but only a service differentiation. Effective performance, for users belonging to C_n and C_m classes, depend on the total workload. Instead, at each premium user (C_p) is assigned an instance of the Guaranteed service class with a throughput of 5 Mbit/s.

We were interested to effective service time and bandwidth, establishing for both an expected value in absence of workload. Then we made test and compared real value with those expected. The results, for each service class, are depicted in figure 3 as the ratio between effective and expected service time.

As the figure shows, in presence of operating system trashing conditions (i.e., increasing the number of connection requests) there are differences between expected and effective values (ratio greater than 1). This difference increases more rapidly for users belonging to C_n and C_m classes than for users belonging to C_p one.

In particular, we assumed to have a Late Timing Error, τ_L, when the effective service time is three time greater then the expected one, and to have an Omission Error, τ_O, when this ratio is greater than five. Althought, as explained in section 3, these tresholds are defined only for medium and premium users, they can be useful to compare the service degradation between the different classes.

According to these rules, the C_m service experiences a Late Timing Error with a workload of about 48 C_n requests instead of only 25 needed for the C_n service. Similarly, an Omission Error occurs with 87 requests for C_m and only 59 for C_n.

On the contrary, a workload of 100 requests is not enough to cause a LateTiming Error for the Premium user class C_p.

It is clear from the figure that the proposed architecture is able to prevent the class C_p from overload conditions by guaranteeing the availability of the service even in the case of server overload.

6. RELATED WORK

Quality of Service provisioning for data delivery and real-time applications have received considerable attentions in [1] and [2]. There are been appreciable progresses in QoS support separately for Web Server [1]. Many works like [13] as well as our previous experience in quality of service support [7], highlights needs to service differentiation even in the end system. Different architectures have been proposed and implemented in order to support QoS guarantees in the end-system. For example, in [12] are proposed some architectural mechanisms to manage communication resources for guaranteed-QoS connections, and in [10, 15] has been addressed the problem of scheduling real-time applications on general-purpose Operating System in order to provide different classes of communication services. Both architectures did not address the implementation issues of a mechanism to control the bandwidth assigned to different class of service. In all revised works, resources control was used to increase performance or to provide class differentiation, without considering the lack of availability due to a poor control of communication QoS.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we focused on Internet-based data delivery services (e.g., services provided by Web, FTP, and video-on-demand servers). These services are run by processes whose activity is typically dominated by network communication; we called them communication-bound processes. We presented an overall description of an operating system extension for quality-of-service differentiation among classes of communication-bound processes. Our strategy relies on the capability of controlling the I/O activity performed by applications. We defined three Class of Service corresponding to different priorities in I/O-resources utilization. An extended evaluation demonstrated that such an architecture is useful to improve the satisfaction of premium users, increasing the effective availabilty in presence of system overload. Finally, we are currently investigating the behaviour of the cosSocket architecture with respect to other performance parameters as response time and jitter. We are also evaluating the influence of architecture setup on this parameters.

8. REFERENCES

[1] T.F. Abdelzaher, N. Batti, “Web Server QoS Management by Adaptive Control Delivery”, in *International Workshop on Quality of Service (IWQOS’99)*, London, UK, June 1999.

[2] T.F. Abdelzaher, K.G. Shin, “End-host architecture for qos-adaptive communication”, in *Proc. of IEEE Real Time Technology and Applications Symposium*, Denver, Colorado, June 1998.

[3] C. Aurrecoechea, A. Campbell, L. Hauw, “A survey of QoS architecture”, in *4th IFIP International Conference on Quality of service*, Paris, France, March 1996.

[4] A. Bondavalli, L. Simoncini, “Failure Classification with respect to Detection”, in *Esprit Project N.3092 (PDCS), 1st Year Report*, IEEE-CS, Los Alamitos, CA (USA), May 1990.

[5] S. Chandra, C. Ellis, A. Vahdat, “Differentiated Multimedia Web Services Using Quality Aware Transcoding”, in *Proc. 19th Annual Joint Conference Of the IEEE Computer And Communications Societies (INFOCOM’00)*,December 2000.

[6] D. Cotroneo, M. Ficco, M. Gargiulo, S. Russo, G. Ventre, “Service Differentiation of Communication-bound Processes in a real-time Operating System”, in *Proc. of 7th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2002)*, San Diego, CA, Jenuary 2002.

[7] D. Cotroneo, M. Ficco, S. Romano, G. Ventre, “Bringing Service Differentiation to the End System”, in *Proc. of the IEEE International Conference on Networks (ICON’00)*, Singapore, Oct. 2000.

[8] D. Cotroneo, M. Ficco, G. Ventre, “Bringing Service Differentiation to the End System”, in *Proc. of the 8th International Workshop on Interactive Distributed Multimedia Systems*, Lancaster, September 2001.

[9] Hassan Gomaa, “Design Concurrent, Distributed, and Real-Time Applications with UML ”, Addison Wesley, Object Technology Series, 2000.

[10] D. Ingram, “Soft Real-Time Scheduling for general Purpose Client-Server Systems”, in *Proc. of the IEEE Workshop in Operating System*, Aug. 1999.

[11] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, J. Hansen, “A scalable solution to Multi-Resource QoS problem”, in *the 20th IEEE Real-Time Systems Symposium*, Aug. 2000.

[12] A. Mehra, Kang G. Shin,“Structuring Communication Software for Quality-of-Service”, in *IEEE Transactions on Software Engineering,* (Vol. 23, No. 10), pp. 616-634, October 1997.

[13] K. Nahrstedt, J. Smith, “The QoS Broker”, in *Proc. of the IEEE Multimedia Spring* 1995,Vol.2, No.1, pp. 53-67.

[14] D. Powell, “Failure Mode Assumptions and Assumption Coverage”, in *Proc. of the 22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, IEEE-CS, Los Alamitos, CA (USA), 1992.

[15] I. Stoica, “A Proportional Share Resource Allocation Algorithm For Real-Time, Time-Shared Systems”, in *Proc. Of IEEE Real-Time Systems Symposium*, December 1996.