Dependability of Component Based Systems

S. Anderson (Univ. of Edinburgh, GB),
R. Bloomfield (Adelard, GB),
M. Heisel (TU Ilmenau, D),
B. Krämer (FernUniv. Hagen, D)

3.11.2002 - 8.11.2002

Abstracts

Modular Specification and Analysis of Reactive Systems

Ramesh Bharadwaj

To date, the Software Cost Reduction (SCR) method has been used to specify system requirements. We propose a modular specification approach to extend the method to system design and software requirements. Our approach consists of three steps: First, the SCR method is used to specify the required system behavior, i.e., the required relation between environmental quantities that the system monitors (called monitored quantities) and the environmental quantities that the system controls (called controlled quantities.) Next, the system designers specify the Input/Output devices required to compute estimates of the monitored quantities and to set values of the controlled quantities. Finally, the required software behavior is specified as three modules: a module REQ which specifies how estimates of the monitored quantities are to be used to compute estimates of the controlled quantities, and modules D_IN and D_OUT which respectively specify how data from the input devices are to be used to compute estimates of the monitored quantities, and how the computed estimates of the controlled quantities are used to compute data to be written to the output devices. We call this the extended SCR (E-SCR) Method. To illustrate this approach, we use the method to specify the system and software requirements of a simple light controller.

A Tool for Generating Specifications from a Family of Formal Requirements

Jan Bredereke

Telephony features are related to components, and telephony users expect quite dependable systems. Feature interaction problems increasingly impair their service. We propose a formal requirements specification methodology that avoids some of the feature interaction problems from the beginning,

and that converts some more into type errors. We maintain all the variants and versions of such a system together as one family of formal specifications. For this, we define a formal feature combination mechanism. We present a tool which checks for feature interaction problems, which extracts a desired family member from the family document, and which generates documentation on the structure of the family. We also report on a quite large case study.

A Framework for the Justification of Computer Systems Important to Safety. Composability of Safety Cases: a Myth?

Pierre-Jacques Courtois

When a computer based system important to safety has to be used and approved for a given application, "claims" are made on the adequacy of the system.

These claims at the application level must be inferred from (expanded into) claims on the architecture, design and operational level. The safety case is the set of arguments and evidence components which support these application claims.

Models are needed at the architecture, design and operations levels to formulate these subclaims and their supporting evidence. The relations between these models extensions required by the proof obligations to preserve the inferences made across the models are established. They are shown to be not easily satisfied, especially in case of COTS, meta-objects, reflection mechanisms, run-time objects, fault-tolerant and self test mechanisms, and other architecture or design subsystems justified by independent sub-safety cases.

Injection of Systematically Selected Errors

Klaus Echtle

The automation of many saftety-critical systems requires fault-tolerant computing systems. Since the required redundant ressources tend to cause substantial extra cost, the designer should try to adapt the fault tolerance technique to the system and application properties as much as possible in order to reduce the expense. The design is mainly influenced by requirements concering fault coverage of the tests, realtime properties, input-output profile, transparency, safety and availability.

These and some further design criteria may lead to a rather complicated implementation of fault tolerance. Besides various verification and valida-

tion methods, error injection can be applied to reveal potential weaknesses in the countermeasurs against faults. It must be particularly guaranteed that no (injected) error can slip through the net built by the fault tolerance technique.

The most interesting issue of error injection is an appropriate selection of errors to be injected. Hand-crafted error sets may be valuable, but they highly depend on personal experience. Randomly generated error sets, on the other hand, typically lead to a small subset of all the implemented reactions to fault occurrence. Thus, a systematic and automated approach should be taken. It will exhibit some similarities to test case generation. However, there are also important differences. Instead of test data, errors are taken as input, which may occur at any time at any location. Moreover, the difficulties in generating correct test outputs do not exist. One can simply take the output of a test run without error injection (as far as the system is deterministic).

Black box testing does not really work for error set generation, because the specification of fault tolerance just says that the system behaviour in the presence of faults should be the same as in the absence of faults. White box testing is feasible, but can be very time-consuming. For these reasons we decide for a so-called grey box approach based on a model "between the specification and the implementation". A Petri net model with time and data attributes of the faultfree system part serves for this purpose. The faulty party is simply cut out of the model. A reachability analysis derives all the possible behaviours the error injector should enforce, each defined by the (wrong) data to be injected and the injection points in time. If the number of test runs under error injection is smaller than the number of paths in the reachability graph, then three selection criteria can be applied: blind random selection (just for comparison), selection of so-called close-to-danger paths, and selection according to structural coverage of the model (precisely speaking: coverage of the place activities in the Petri net). Model-based expriments turned out that, surprisingly, the structural coverage criterion leads to the best results.

A formal description for functional dependencies in a component based system for packet level network applications

Olaf Ehlert

Current heterogeneous quality-of-service and security requirements are usually application driven and they induce a high demand for advanced and flexible classification of packets in core network systems.

In our department we have developed a fine grained component based

framework to construct networking applications like router, firewalls or VPNs, which can react highly flexible on changing requirements, even during runtime.

Within this framework naturally there exist a diversity of functional dependencies between the components, e.g. different available lookup algorithms have varying needs for information and induce requirements in the preceding extraction stage.

A pure syntactical description of the components and their interfaces is insufficient to apply formal analysis on such a network application.

By introduction of a dynamic type system for methods in an interface, based on dynamic attachments to the packet data, one get the possibility to map functional dependencies into pre- and postconditions, that can be checked and analyzed dynamically on runtime and statically before runtime.

Evolution of Computer-based Systems

Massimo Felici

Evolution is one of the major issues affecting system dependability as well as engineering activities and environments. The most common understanding considers evolution as a phenomenon that needs to be limited. By contrast this talk takes into account evolution as a necessary phenomenon for computer-based systems. This talk reviews a taxonomy of evolution identifying a conceptual space in which evolution manifests in different forms. The taxonomy provides a conceptual framework to analyse evolutionary phenomena of computer-based systems as well as models of evolution and their limits. The taxonomy of evolution points out dependability aspects of computer-based systems. The discussion of evolutionary phenomena emphasises how they differently relate to the dependability of computer-based systems.

The talk then shows an empirical investigation of a case study focusing on requirements evolution.

In conclusion, this talk takes account of evolutionary phenomena of computer-based systems and relates them to dependability. This provides a conceptual framework to analyse evolution and its influence on dependability.

Accountability of (Designer/Deployer/Executer of) Components

Andreas Pfitzmann and Elke Franz

When speaking about components, do we mean (1) self-contained subsystems, (2) software executed on hardware, or (3) software only? Components

depend on their environment in general, and lower execution layers in particular. These execution layers must not be confused with design layers common in software engineering.

To hold components accountable, besides a precise specification of their expected behaviour, we need a precise log of their manifested behaviour: (a) input, (b) environmental conditions, and (c) output. For software components, this means (a) input, including real time properties, (b) execution by underlying machine, and (c) output, including real time properties.

Accountability gets even more difficult if component-based design means no source code available for components and/or no legal entity taking responsibility for the components and their integration.

Concerning logging execution, input and output: Who is going to log? The executing machine? What if it might be the cheater? Expense of logging? Log only what cannot be calculated repeatedly.

To resolve disputes on logs, outputs should be digitally signed and the digital signature on inputs checked before processing them. But who is going to sign? The software component? This is not possible. The executing machine? This is not sufficient. Consequently, a combined signature is necessary.

Modeling and Testing with the Abstract State Machine Language

Wolfgang Grieskamp

The Foundations of Software Engineering Group at Microsoft Research developed the Abstract State Machine Language (AsmL). AsmL serves as an all-round modeling language for use in the Microsoft Environment. It incorporates concepts of Gurevich's Abstract State Machines, of specification languages like VDM or Z, of functional languages, and of modern object oriented languages. The language supports meta programming and is embedded into the .NET environment with decent tool support. One application of an AsmL model is to use it for test case derivation and runtime verification, for which techniques have been developed recently. The runtime verification is able to deal with non-determinism in the model and with matching object references between the implementation and the model. The test derivation tool is based on a parameter generator and a sequence generator and comes with a convenient graphical user interface. AsmL and its tools are being used in several pilots at Microsoft product groups.

Concurrency Patterns - a Petri Net Perspective – work in progress –

Monika Heiner

Gamma et al. introduced in the middle of the 90's design patterns. This idea represents a landmark in software engineering, because from that moment on seasoned programmers are able to communicate own programming experience and design skills to freshmen. In the recent past, first attempts for design pattern catalogues dedicated to special application areas have been published.

We follow that line and propose a collection of concurrency patterns which are the outcome of many case studies done in model-based system development. We present the control model for the production cell, which is composed of only a few patterns: bounded producer/consumer pattern, three communication patterns for producer/consumer pipeline (independent input/output, dependent input/output, mutually exclusive input/output), mutex pattern, and the basic motion step pattern. These patterns come along with pattern properties, which establish model consistency criteria.

Afterwards, several further concurrency patterns are shortly sketched: n mutex resource pattern (pattern property to guarantee deadlock freedom: acyclic access structure), n layered client/server pattern (pattern property for deadlock freedom: acyclic communication structure), and fault-tolerant basic structures (n version programming, recovery block scheme). Finally, two challenges for future work are pointed out:

1. Is it possible to uplift Dijkstra's structured (sequential) programming approach to avoid uncontrolled use of goto's to a "structured concurrent programming" one to avoid uncontrolled use of synchronisation / communication?

2. Is it possible to develop software by step-wise pattern refinement at different abstraction levels, like problem frames, architecture styles, architecture patterns, e.g. concurrency patterns, design patterns, and finally idioms?

Tools for building and evaluating high assurance software components

Constance L. Heitmeyer

This talk describes the role that formal methods and tools can play in constructing a high quality software component, i.e., a software component for which there exists compelling evidence that the component satisfies its requirements. A number of real-world examples are presented in which formal

methods and their support tools were used to detect errors or to verify properties in software systems and software components. The examples include the U.S. Navy's A-7 Operational Flight Program, Rockwell's Flight Guidance System, and two current U.S. Navy systems, a Weapons Control Panel and a software-based cryptographic device. A brief introduction is also given to two software components that are currently under development – NASA's Fault Protection Engine and a second software-based cryptographic device – and how software tools are being used to evaluate both the components's specification and its implementation. The tools used to support the SCR (Software Cost Reduction) tabular notation are used in all of the examples.

Rely/guarantee-conditions and their Relevance to error-tolerance

Cliff B. Jones

Rely/guarantee-conditions were developed (some 20 years ago) to address the need for a compositional development method for concurrent programs. The original work addressed shared-variable programs but subsequent authors -notably Colin Stirling- have shown that the same broad ideas of documenting interference in a specification applies to communication-based (process algebraic) concurrency. The underlying link is that concurrency is all about interference. Following this line, the question can be asked whether rely-conditions can be used to handle the "interference" that comes from faults in other systems (which manifest themselves as errors in the system being specified). In common with John Rushby's contribution, I recognise the difficulties caused by failures cutting across levels of abstraction (but I have something to say here) and the need for ordered (increasingly pessimistic) assumptions on the environment.

Synthesizing Refinements from Predicate Translations

Florian Kammüller with Steffen Helke and Jeff Sanders

We suggest a method for constructing refinements of a formal specification that is based on predicate translations input by a system engineer. The main idea of the method is to infer the retrieve relation, i.e. the relation that connects abstract and concrete data, from those given translations of predicates. Classical refinement methods consider this relation as a necessary input. Building on results from Abstract Interpretation we present a technique that enables to derive the retrieve relation from translations of

predicates. Once the retrieve relation is known a concrete implementation can be calculated by standard methods.

Parameterised Contracts for Software Components – Contractual Use, Adaptation and Reliability Prediction of Software Components

Ralf H. Reussner

In this talk we introduced the concept of "parameterised contracts" for software components and demonstrated their application on automated component adaptation and reliability prediction.

After a review of current challenges in the field of software components, B. Meyer's design-by-contract principle was formulated for software components. The contractual use of software components (a term sometimes used loosely – or even inconsistently – in current literature) at design-time refers to the requires-interface as a kind of pre-condition (because it states the expectations of the component against its environment necessary for the component to operate). Consequently, the provides-interface acts as a post-condition of the component, since here it is stated what the user can expect to component to fulfil, if the component is deployed according to its pre-condition.

We discuss that this translation acts as a starting point for a generalisation of the design-by-contract principle. In practice, it proves to be very hard for the software component developer to state a single pre- and post-condition of the component without making to many assumptions on the component environment. A parameterised contracts is a reversible function, mapping a pre-condition to a related post-condition.

For example, if a user only uses a subset of a component's functionality the parameterised contract will compute the weakest pre-condition (i.e., require-interface) allowing the component to fulfil the requested services. In a different case, a component may find that its environment does not deliver all functionality requested in the component's require-interface. Consequently, the parameterised contract computes the strongest post-condition (i.e., provides-interface) possible in this specific environment. These scenarios demonstrate parameterised contracts as a mean to increase the reusability of software components.

Besides these functional adaptations also the concrete realisation of parameterised contracts for protocol-modelling component interfaces were demonstrated.

The last part of the talk dealt with parameterised contracts and component interfaces specifying extra-functional properties of components. Here, we concentrated on the component's reliability.

Our evaluation confirms that software component reliability prediction necessitates modelling the behaviour of binary components. Signature-level reliability (associating reliability measures to interface names) is not sufficient. Fortunately, our measurements also show that an abstract protocol view of that behaviour is sufficient to achieve reliability model accuracy. Furthermore, it our measurements clearly demonstrate that the reliability of a component strongly depends on its environment. Therefore, we advocate the use of parameterised contracts for context-dependent reliability computation, rather than using a fixed value.

On Problem Analysis, Test Automation and Reliability Testing

Thomas Rottke

Especially in the area of safety critical systems testing consumes a high fraction of project budgets.

A frame oriented approach to problem analysis and test automation yields to precise requirements and early acceptance criteria by means of system environment models and system models as well.

Therefore conformance testing could be done in an early stage of the system development.

Although conformance testing and reliability testing have the same basic concepts, reliability testing needs additional stress test profiles and stress test interfaces.

Modular Certification

John Rushby

Safety is a system property, so its assurance requires examination of the whole system. Thus, airplanes, for example, are certified as a whole: there is no established basis for separately certifying some components independently of their specific application in a given airplane.

But if a notion of modular certification could be developed, we could envisage development of components that could be largely "precertified" and used in several different contexts within a single system, or across many different systems.

I examine the issues in modular certification of software components and propose an approach based on assume-guarantee reasoning. I extend the assume-guarantee method from verification to certification by considering behavior in the presence of failures. This exposes the need for partitioning, and stratification of assumptions and guarantees into normal and abnormal

cases. I identify three classes of property that must be verified within this framework: safe function, true guarantees, and controlled failure.

Pre-Developed Software to be re-used in Safety-Relevant Component-based Systems

Francesca Saglietti

The problem of assessing the suitability of off-the-shelf (OTS-) software components for new development projects represents a serous challenge for the software engineering community. The strategy suggested consists of 5 decision phases:

1. identification of safety demands at system level
2. analysis of OTS-role within new system (safety relevance & sensitivity)
3. qualitative (subjective) assessment of development process and product quality
4. quantitative (objective) assessment of past (testing or operating) experience
5. validation of component interfaces within the integrated system

With respect to point 4. an approach to evaluate past operational experience in terms of reliability to be expected with respect to a new usage profile was presented.

With respect to point 5. a classification of component interface inconsistencies was proposed, including a.o. also inconsistency classes related to syntactical problems and semantical problems in the logical domain, semantical problems on the physical domain, temporal constraints, deviations between representable data and physical context for the purpose of supporting testing and wrapping techniques.

Confidentiality-Preserving Refinement – A Step to Integrate Confidentiality in Component-Based Design

Thomas Santen with Maritta Heisel, Andreas Pfitzmann, Elke Franz and Florian Kammüller

Integrating security concerns, in particular confidentiality, in component-based design means to find the right places where to address security issues in the development process of components and component-based systems:

one must find ways to distribute security requirements to component specifications, to ensure that implemented components satisfy these specifications while relying on security mechanisms that a component framework may provide as part of container and component server implementations. One must take into account the effect that component deployment has on security issues. To consider the run-time environment of a component-based system, e.g. the server the system runs on, is also important: it must ensure that security is indeed enforced and security flaws can be logged and evaluated after the fact.

A fundamental problem in this setting is that security properties, most notably confidentiality properties, in general are not compositional. This is true when decomposing and composing security properties at one level of abstraction, and it is also true when considering refinement, i.e. the systematic transition from one level of abstraction to a more detailed one.

Confidentiality-preserving refinement describes a relation between a specification and an implementation that ensures that all confidentiality properties required in the specification are preserved by the implementation in a probabilistic setting. It is interesting to investigate conditions under which that notion of refinement is *compositional*, i.e. the condition under which refining a subsystem of a larger system yields a confidentiality-preserving refinement of the larger system. It turns out that the condition for compositionality can be stated in a way that builds on the analysis of subsystems thus aiding system designers in analyzing the confidentiality properties of a system whose components are implemented independently of each other.

Contracts versus compositionality: Reasoning about extra-functional system properties

Heinz Schmidt

The software architect is concerned with both functional and extra-functional design. In component-based software engineering, an important task in functional design is the adaptation of a component interface for use by other components. In extra-functional analysis the focus is rather on the prediction and reasoning about performance, reliability, usability and other "ilities". These are often system properties that depend on other components in the environment and the architecture or framework that the component is deployed in.

We present a concurrent trace-based model and method for component composition and automatic adaptation called called dependent finite state machines (DFSMs). DFSMs are based on finite state machines and Petri nets, permit model checking, execution-based verification and extend the

notion of design-by-contract from precondition, postcondition and invariant assertions on objects to dynamic models for components that cater for parameters actualised at deployment time. While DFSMs capture only limited aspects of component behaviour, they are capable of carrying various extra-functional properties as add-on attributes.

This seminar presents work in progress. We focus on key ideas for a partially compositional approach and on problems with compositionality for extra-functional models.

Process-Oriented, Consistent Integration of Software Components

Sebastian Thöne

The integration of software components becomes a more and more important issue in software engineering. Process-oriented approaches should provide automated information processes. To support dependability properties, the software components have to be integrated in a consistent way, i.e., their export interfaces have to be respected by the importing components. Furthermore, the type system of component interfaces has to support a tunable degree of freedom. This allows the insertion of components with interfaces of restricted but sufficient degree of compatibility. In this talk, we introduce a concept for consistent and exible integration of components. We present a process modeling language that combines UML and XML in order to support consistent, exible, and executable processes. Finally, we outline a formalization of the proposed component type system including typed transitions for bridging incompatible interfaces.

BALES

Willem-Jan van den Heuvel

In this presentation a methodology was outlined, called binding Business Applications to Legacy systems (*BALES*), that allows to blend modern business objects and processes with objectified legacy data and functionality (hence legacy objects) in order to construct flexible, configurable applications, that are able torespond to business changes in a consistent and reliable way.

Legacy objects serve as conceptual repositories of extracted (wrapped) legacy data and functionality. These objects are, just like business objects,described by means of their interfaces rather than theirimplementation. Business objects in the *BALES* methodology are configured so that part of their implementation is supplied by legacyobjects. This means that their interfaces are parameterizable (or self-describing) to allow these objects to

evolve by accommodating upgrades or adjustments in their structure and behavior.