Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 236 (2009) 21–32
www.elsevier.com/locate/entcs

On Architecting Software Fault Tolerance using Abstractions

Rogério de Lemos[1]

Computing Laboratory
University of Kent
Canterbury, UK

Abstract

In this position paper, we argue how architectural abstractions can be effective in developing fault-tolerant software systems. Depending on the fault model and the resources available, different abstractions can be employed for representing architectural issues related to fault tolerance. These architectural abstractions, and their internal views, can be instantiated into concrete components and connectors for designing fault-tolerant software architectures. Since structural and behavioural properties associated with these abstractions are formally specified, the process of verifying and validating software architectures can be automated. In this paper, we focus on two architectural abstractions: the idealised fault-tolerant architectural element (iFTE), which is based on exception handling, and the halt-on-failure architectural element (HoFE), which assumes crash failure semantics.

Keywords: software architectures, dependability, fault tolerance, scenarios, formal methods, testing

1 Introduction to Architecting Dependable Systems

One of the benefits of a well-structured system is the reduction of its overall complexity, which in turn should lead to a more dependable system. *Dependability* is defined as the ability of a system to deliver service that can justifiably be trusted [2]. The process of system structuring may occur at different stages of the development or at different levels of abstraction. Reasoning about dependability at the architectural level has lately grown in importance because of the complexity of emerging applications, and the trend of building trustworthy systems from existing untrustworthy components, such as off-the-shelf (OTS) components and legacy systems, that were not originally designed to interact with each other. One major problem when using existing components is the inability to change, or even to access, their internal designs and implementations. Moreover, since the evolution of these components might be outside the control of the system architect, solutions that

[1] Email: r.delemos@kent.ac.uk

1571-0661/© 2009 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2009.03.012

are dependent on a specific component implementation become unfeasible. Based on these limitations, the delivery of correct service, and the justification of this ability, has to be obtained from the components interfaces and their interactions rather than from their internal designs or implementations. Hence the drive for considering dependability concerns at the architectural level, rather than late in the development process. From the perspective of software engineering, which strives to build software systems that are free of faults, the architectural consideration of dependability compels the acceptance of faults, rather than their avoidance. Thus the need for novel notations, methods and techniques that provides the necessary support for reasoning about faults at the architectural level. For example, notations should be able to represent non-functional properties and failure assumptions, and techniques should be able to extract from the architectural representations the information that is relevant for evaluating the system architecture from a certain perspective.

System dependability is measured through its attributes, such as reliability, availability, confidentiality, and integrity, and there are several means for attaining these attributes, which can be grouped into four major categories [2]. *Rigorous design*, or fault prevention, which aims at preventing the introduction or the occurrence of faults. *Verification and validation*, or fault removal, which aim at reducing the number or severity of faults. *Fault tolerance*, which aims at delivering correct service despite the presence of faults. *System evaluation*, or fault forecasting, which aims at estimating the present number, the future incidence, and the likely consequences of faults. Since system structuring is relevant across all the dependability means, the ensuing discussion will focus on how fault prevention and removal can be used in conjunction with fault tolerance in order to obtain dependable software systems.

2 Architecting Fault Tolerance

Fault tolerance aims to avoid system failure via error detection and system recovery [2]. Error detection at the architectural level relies on monitoring mechanisms, or probes, for observing the system states to detect those that are erroneous at the components interfaces or in the interactions between these components. On the other hand, the aim of system recovery is twofold. First, eliminate errors that might exist at the architectural state of the system. Second, remove from the system architecture those elements or configurations that might be the cause of erroneous states. From the perspective of fault tolerance, system structuring should ensure that the extra software involved in error detection and system recovery provides effective means for error confinement, does not add to the complexity of the system, and improves the overall system dependability [8].

Since fault tolerance has a global system scope, it should be related to both architectural elements (components and connectors) and architectural configurations. However, the incorporation of fault tolerance into systems normally increases their complexity, making their analysis more difficult. One way of handling the inherent

complexity of fault-tolerant systems is to adopt architectural abstractions. These are able to hide system complexity, and provide the means for analysing how errors are propagated, detected and handled, and how faults in the system are handled.

The provision of fault tolerance relies on the existence of redundancy, which can be incorporated either implicitly or explicitly at the architectural level. An example of implicit redundancy is the usage of exception handling for supporting error recovery. If special care is not taken when structuring the system, the normal and abnormal specifications can be entangled thus increasing system complexity. Explicit redundancy is an inherent aspect of strongly-structured systems, i.e., systems in which the structuring of redundancy is part of the actual system, thus restricting the impact of faults. Examples of explicit redundancy are N-version programming and recovery blocks, which are two software fault tolerance techniques.

Architectural abstractions can be effective when developing fault-tolerant software systems, applying both implicit and explicit redundancy. The *idealised fault-tolerant architectural element* (iFTE) [5], which is an abstraction based on exception handling, provides the means for promoting error confinement and supporting fault tolerance at the architectural level. The *halt-on-failure architectural element* (HoFE) [3], which is an abstraction that assumes crash failure semantics, provides the basis for incorporating explicit redundancy when designing fault-tolerant systems. Depending on the fault model and the availability of resources, the appropriate architectural abstraction should be used. For obtaining fault-tolerant software architectures, these abstractions are instantiated into architectural components and connectors, which are then configured depending on the interaction constraints dictated by their structural and behavioural properties.

These architectural abstractions are presented in the context of a general approach for the formal specification, verification, and validation of fault-tolerant software systems. The adoption of abstractions together with the use of formal languages allows the automatic verification of high-level models for identifying and removing design faults at the initial stages of the software lifecycle. After the verification activities, the architectural models can be used as a basis for generating both the system's source code, and its architectural-based test cases. The automatic transformation from architectural models into source code tends to reduce the number of faults that are introduced into the system implementation when compared with an error prone manual programming. The architectural-based test cases are able to identify and remove implementation faults that are related to the architectural design of the system, although they are restrictive on their system test coverage. In the following, we describe how rigorous design and verification and validation techniques can be used during the development of fault-tolerant software systems.

2.1 Rigorous Design

Rigorous design is concerned with all the development activities that introduce rigor into the design and implementation of systems for preventing the introduction of faults or their occurrence during operation. Development methodologies and con-

struction techniques for preventing the introduction and occurrence of faults can be described respectively from the perspective of development faults and configuration faults (a type of interaction fault) [2]. Development faults can be prevented from being introduced during the development of software systems by using formal or rigorous notations for representing and analysing software at key stages of development. On the other hand, configuration faults can be prevented from occurring during system operation by protecting the architectural elements and their context against potential architectural mismatches (design faults) that might exist between them. These vulnerabilities can be prevented by adding to the structure of the system architectural solutions based on integrators (more commonly known as wrappers). The assumption here is that the integrators are aware of all incompatibilities that might exist between a component and its environment. The architectural abstractions presented in the following deal with both development and configuration faults.

2.1.1 Architectural Abstractions

The architectural abstractions presented below are used to structure software architectures of fault-tolerant systems. The first one implements error handling based on the exception handling mechanism, while the second one is based on crash failure. The components are described in terms of provided and required interfaces, to which operations are associated.

Idealised Fault-Tolerant Architectural Element (iFTE)

The *idealised fault-tolerant architectural element* (iFTE) is an architectural abstraction for structuring fault-tolerant systems. This abstraction enforces the principles associated with the concept of the idealised fault-tolerant component [1], and incorporates mechanisms for detecting errors, as well as propagating and handling them in a structured way. The iFTE abstraction provides an explicit separation of concerns between two types of behaviour: (i) the normal behaviour, which realises the services of the application, and (ii) the abnormal (exceptional) behaviour, which realises the detection, propagation and handling of errors. In order to provide this separation, the iFTE abstraction defines four types of interfaces, which are presented in Figure 1. While the I_iFTE_PN and I_iFTE_RN are responsible for the normal behaviour, I_iFTE_PA and I_iFTE_RA are responsible for the abnormal behaviour.

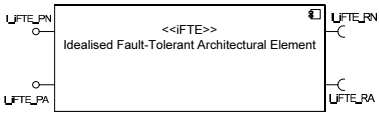

Fig. 1. iFTE Abstraction

iFTE External View. The external behaviour of the iFTE is defined through behavioural scenarios related to its external interfaces. A scenario is a sequence of events expected during the system operation. In the context of this paper, a scenario is defined as a sequence of events triggered by the request of an operation

of the I_iFTE_PN interface, including operation responses, other operation requests and signalling of exceptions. A total of nine different scenarios were identified for the iFTE. The scenarios are derived from the interaction rules existing between the interfaces of the iFTE. These rules involve requests of external services, the reception of the respective returns (normal or abnormal), raising of new exceptions, propagation of received exceptions, and masking of exceptions for tolerating software faults.

iFTE Internal View. The internal view of the iFTE (Figure 2) is composed of five architectural elements: (i) the Normal component implements the normal behaviour of the iFTE; (ii) the Abnormal component handles the exceptions raised by the Normal component, and those propagated from the environment of the iFTE; (iii) the Provided component acts like a bridge between the services provided by the iFTE and its environment, including the signal of exceptions; (iv) the Required component also acts like a bridge, but between the required services of the iFTE and its environment; and (v) the Coordinator connector coordinates the interaction between the four internal components. It is important to stress that the iFTE also supports the resolution of architectural mismatches. This high-level adaptation is carried out by the Provided and Required components present in Figure 2.

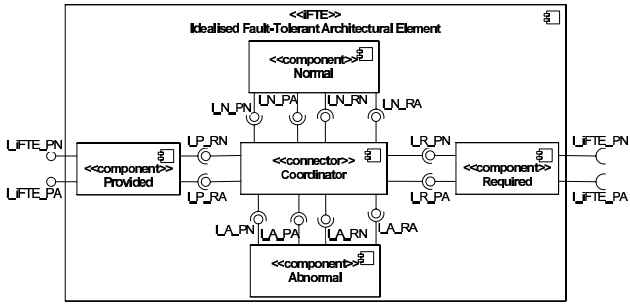

Fig. 2. Internal View of the iFTE

Analysing the internal details of the iFTE, it is possible to distinguish some interactions between the internal elements, which characterise new scenarios when compared with the external view. As a whole, we have identified four new scenarios involving the masking of internal exceptions, and the other scenarios of the external view.

Halt-on-Failure Architectural Element (HoFE)

The halt-on-failure architectural element (HoFE) is an architectural abstraction for the provision of error confinement and fault tolerance, and which enforces the principles associated with the *crash failures* fault model [9]. When an HoFE fails, it fails silently without producing any error signal. The HoFE abstraction defines two types of interfaces, which are presented in Figure 3: (i) I_HoFE_Prov defines a set of operations provided by the HoFE; and (ii) I_HoFE_Req specifies operations required by the HoFE for implementing its behaviour. It is assumed that an HoFE is able to detect failures on other architectural elements from which requests operations, e.g., by associating *time-outs* with the I_HoFE_Req interfaces.

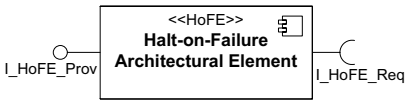

Fig. 3. HoFE Abstraction

HoFE External Behaviour. The external behaviour of the HoFE architectural abstraction is defined through five basic scenarios: (i) *internal normal execution*, when an HoFE provides the requested services without requesting external services; (ii) *internal erroneous execution*, when an HoFE fails before requesting external services; (iii) *external normal execution*, when an HoFE provides the requested services after receiving the requested external services; (iv) *external erroneous execution 1*, when an HoFE fails after receiving the requested external services; and (v) *external erroneous execution 2*, when an HoFE fails after failing to receive the requested external services. Based on these scenarios, one can describe more complex scenarios of fault-tolerant software architectures that are based on the HoFE architectural abstraction.

HoFE Internal View. The internal view of the HoFE can be implemented using different strategies for detecting errors, usually involving explicit redundancy. Figure 4 presents a possible implementation using two redundant components. In this approach, the error detection is conducted by the Decider, which evaluates the results of the two executing versions of components. If there is no consensus between them, the result is considered unreliable. Since the error detection depends on both components, this implementation of the HoFE does not provide internal fault tolerance and is not able to recover from internal errors. However, if redundant HoFEs are used, fault tolerance can be implemented at the architectural level.

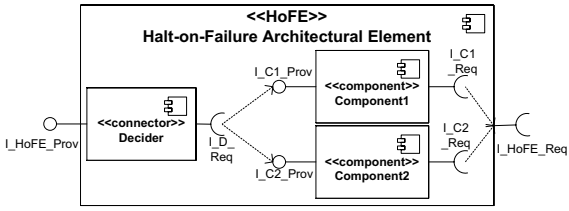

Fig. 4. Internal view of the HoFE using Redundant Components

2.1.2 Architectural Representation

An abstraction-based fault-tolerant software architecture is obtained by following these two steps: (i) instantiation of the architectural abstraction into architectural elements; and (ii) integration of the architectural elements into an architectural configuration.

For the formal specification of the architectural elements, the B-Method machine explicitly represents the provided and required interfaces, as well as the respective operations. In addition, the events related to requests and responses of operations are explicitly represented as B-Method operations. Complementary to the B-Method representation, CSP is employed for describing the sequence of events

that define the scenarios of an architectural abstraction. For the architectural configuration, a B-Method machine represents the structural information about the software architecture, e.g., architectural elements, including their interfaces and operations, and how these are interconnected for obtaining an architectural configuration. On the other hand, the CSP specification defines the sequence of events associated with each architectural element, and how these events are related at the architectural level.

2.1.3 Development Method

In our approach, abstractions are first-level units, guiding the development since the specification, until the verification and validation of the software architecture. An overview of the development method is shown in Figure 5.

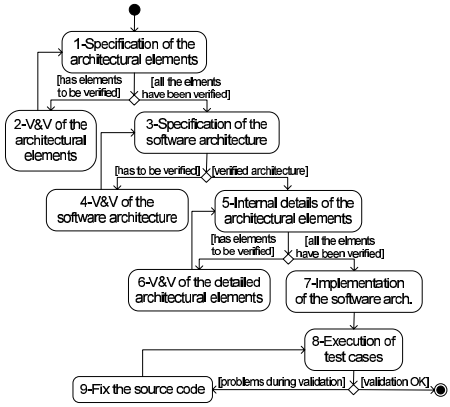

Fig. 5. A Rigorous Development Method

Figure 6 details the execution of Activities 2, 4, and 6 of Figure 5, which refer to the verification and validation (V&V) of the software architecture. A main feature of our proposed approach is that the scenario concept is used in all the V&V activities.

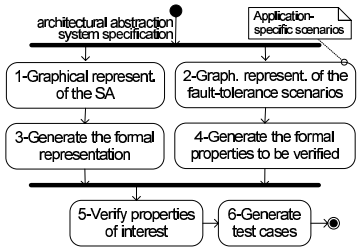

Fig. 6. Steps of the V&V Activities

2.2 Verification and Validation

Verification and validation, also known as fault removal, is concerned with all development and post-deployment activities that aim at reducing the number or the severity of faults [2]. The role of architectural representations in the removal of

faults during development is twofold: first, it allows faults to be identified and removed early in the development process; and second, it also provides the basis for removing faults late in the process. The early removal of faults entails checking whether the architectural description adheres to given properties associated with a particular architectural style, and whether the architectural description is an accurate representation of the requirements specifications. The late removal of faults entails checking whether the implementation fulfils the architectural specification. While early fault removal is essentially obtained through static analysis, late fault removal is gained through dynamic analysis.

2.2.1 Verification Process

The verification process of software architectures that are based on architectural abstractions consists of verifying properties associated with the architectural elements, and the architectural configuration of these elements. For the architectural elements, we should check whether they are consistent against an architectural abstraction. For the architectural configuration, we should check whether it follows the composition rules dictated by the architectural elements. The properties to be verified address four verification goals: (i) *integrity consistency*, which comprises the syntactical analysis of the B-Method machines, as well as the integrity of their variables; (ii) *architectural abstraction scenarios violations*, which amounts to identifying possible violations on the specified scenarios; (iii) *architectural scenarios violations*, which involves analysing specific architectural scenarios originated from the composition of architectural elements; and (iv) *quality requirements scenarios*, which assess the satisfaction of specific quality requirements through application-specific scenarios.

The properties to be verified can be automatically checked using the ProB model checker [6]. Properties that address verification goals (i) and (ii) are expressed as *assertions*, i.e., rules to which the model should comply. If one of these rules is violated, an error message and a counter-example are presented by the model checker. Properties that address verification goals (iii) and (iv) are associated to the model as *definitions*, i.e., state patterns that the model checker tries to find ("goals" to be achieved). If there is a state that satisfies this rule, a warning message and an example are presented by the ProB model checker. Since they are derived from the behavioural scenarios, the properties regarding verification goals (ii) to (iv) are considered *scenario-based* properties.

2.2.2 Validation Process

There are several techniques that can be used for validating a software system against its architectural representation. In the following, we focus on two of those techniques, integration testing and robustness testing.

Architectural-Based Integration Testing

One of the reasons to focus on integration testing is that the identification of mismatches between architectural elements is critical during system integration.

First, we show how to build a dependency matrix for determining the integration testing order, and then we present how the integration test cases are identified from the specification of the software architecture.

The existence of dependencies between architectural elements, in terms of their provided and required services, imposes an order on how architectural elements are integrated, thus facilitating fault localisation. The basis for establishing this order is obtained from the dependency analysis between architectural elements. For the dependence analysis, we use the chaining approach [10], which considers the elements defined at the architectural description, as for example, components, connectors, and interfaces. The links connect architectural elements that are directly related, and they represent dependencies between them.

The features considered during the dependency analysis depend on the notation used to specify the architecture. In the B-Method and CSP specifications, the architecture is described in terms of architectural elements, interfaces, operations, exceptions and events. These features are used to construct a dependency matrix (DM) that represents the relationships among architectural elements. The columns of a DM represent the dependency in the relationship, and its rows represent the depended-on element. Both structural and behavioural dependencies are indicated in the matrix, and this information could be obtained from the architectural representation. For example, in our case, since we are using B-Method and CSP to describe the architecture, we can obtain structural dependencies from B-Method notation, and the behavioural ones from CSP. The dependencies among architectural elements at the architectural level are established by the interactions between elements, and the constraints on these interactions. These interactions usually involve structural dependencies that are complemented with behavioural dependencies [10] since both aspects are important for a precise analysis of the architecture. Structural dependencies involve mechanisms that are used for creating new architectural elements from other elements, or composing existing architectural elements. In the B-Method, for example, information on structural dependencies are obtained from clauses, like, *includes*, *sees*, *uses*, *extends* and *promotes*. Behavioural dependencies, on the other hand, involves dynamic interactions, such as: temporal (the behaviour of one component precedes or follows the behaviour of other components), causal (the behaviour of one component implies the behaviour of another component), state-based (the behaviour cannot happen unless the system or some part of it is in a specified state) or input/output (a component requires/provides information to another component). For example, the CSP definition of an execution sequence characterises causal dependencies among events.

The objective of integration testing is to exercise the interactions between the implementation of architectural elements, through the operations at their interfaces, to determine whether they implement the valid scenarios. In particular, the objective is to identify mismatches related to the flow of exceptions between architectural elements. First of all, we have to identify the invocation sequence of each operation

in the different interfaces. This can be obtained from the dependence matrix, since the behavioural relationships, derived from the CSP specification, gives the synchronization sequence of internal and architectural events. From this information we can construct a graph from which test cases can be derived for each integration step. In this graph, each architectural interface is represented as a node. Operations that refer to these interfaces (requests and responses) are represented as edges, which either points to the interface that it refers (service request), or points to the source of the requesting (service response). The graph also has a start node, which requests the first service that starts the interaction, and many final nodes, which represent the possible returns of the first service request. From the CSP specification of the software architecture, it is necessary to construct a graph of the execution sequence of the B-Method operations. In this graph, each operation that refers to a request of an operation is represented as an edge. The respective response is also represented as an edge departing from the operation to the returned value (node). In our approach, which is based on the MDCE+ method [4], each path (from the root to a leaf) is considered a test case for integration testing.

Architectural-Based Robustness Testing

In order to validate the consistency between the software architecture and the system implementation, architectural-based robustness testing is being promoted. This approach relies on the formal description of the software architecture, and corresponding architectural behavioural scenarios. Robustness testing is a necessary step to complement functional testing, and its role is to guarantee that the software behaviour is acceptable in the presence of internal or external failures, or stressful environmental conditions.

The formal representation of a software architecture is used as an input to the validation process. From the architecture specification we can: (i) define the types of faults to be injected; (ii) establish the points of injection, i.e, the operations of the architectural interfaces where faults should be injected; (iii) generate abstract test cases, which are the sequences of operations that should be activated in each test execution.

The definition of a representative fault model starts from an abstract classification until it reaches a concrete list of faults. First, it is essential to clearly identify the type of faults that can be injected in the system under test. Second, this classification must consider both the origin of the faults (internal, external, human, and non-human), and their nature (data value, and event sequence). Finally, these types of faults will be used to identify a list of faults for the system under test.

The points in which to inject faults are derived from the B-Method and CSP specifications. For identifying the places in which faults can occur, the proposed approach relies on both structural and behavioural dependencies between architectural elements. In brief, the architecture specification is used to determine the dependencies among architecture elements. The idea is to select the points of injection (elements to inject) so that, if the objective is to determine the impact of failures in the rest of the system by a given architectural element, we can select to inject those

components and connectors on which the architectural element depends on. On the other hand, if the aim is to analyze the impact of faults of a given architectural element in the rest of the system, the points of injection can be those components and connectors that depend on that architectural element. In this way we can reduce the size of a fault injection test sequence, by prioritizing the tests based on dependence analysis. The information for that is obtained from the architectural configuration of the B-Method machine, and the architectural scenarios of the CSP specification.

An abstract test case is directly derived from an architectural scenario specified in CSP, according to the sequence of operations executed in that scenario. For example, for the halt-on-failure property of HoFE, the specified test case requires the execution of two scenarios: a failure scenario followed by a scenario with no failures. For satisfying the halt-on-failure property, the HoFE should remain halted during the execution of the second part of the test case. How to identify the points in the architecture where faults should be injected can be found elsewhere [7].

3 Conclusions

The development of fault-tolerant software system can be more effective if architectural abstractions are employed. These are able to abstract away from system details while providing the means for analysing how errors are propagated, detected and handled, and how faults are handled. Associated with these abstractions, we have defined a general rigorous development approach for the formal specification, verification and validation of software architectures that are based on these abstractions. In this paper, we have presented two distinct architectural abstractions from which fault-tolerant software systems can be built: the idealised fault-tolerant architectural element (iFTE), and the halt-on-failure architectural element (HoFE).

Acknowledgement

The author would like to thank Patrick H. S. Brito, Cecília Mary F. Rubira, Eliane Martins, and Regina Moraes for their contributions to the work reported in this paper.

References

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

[2] Avizienis, A., J.-C. Laprie, B. Randell and C. Landwehr, *Basic Concepts and Taxonomy of Dependable and Secure Computing*, IEEE Transactions on Dependable and Secure Computing **1** (2004), pp. 11–33.

[3] Brito, P. H. S., R. de Lemos and C. M. F. Rubira, *Development of fault-tolerant software systems based on architectural abstractions*, in: *European Conference on Software Architectures (ECSA 2008)*, LNCS **5292**, 2008, pp. 131–147.

[4] Brito, P. H. S., C. R. Rocha, F. Castor Filho, E. Martins and C. M. F. Rubira, *A Method for Modeling and Testing Exceptions in Component-Based Software Development*, in: *Proc. of the 2nd Latin American Symposium on Dependable Computing (LADC 2005)*, LNCS **3747**, 2005, pp. 61–79.

[5] de Lemos, R., *Architectural Fault Tolerance using Exception Handling*, in: *Architecting Dependable Systems IV*, LNCS **4615**, 2007, pp. 142–162.

[6] Leuschel, M. and M. J. Butler, *ProB: A Model Checker for B*, in: *Proc. of FME 2003*, LNCS **2805**, 2004, pp. 855–874.

[7] Moraes, R. and E. Martins, *Fault Injection Approach Based on Architectural Dependencies*, in: *Architecting Dependable Systems III*, LNCS **3549**, 2005, pp. 300–321.

[8] Randell, B., *Turing Memorial Lecture Facing Up to Faults*, Comput. J. **43** (2000), pp. 95–106.

[9] Schneider, F. B., *Byzantine generals in action: implementing fail-stop processors*, ACM Transactions on Computer Systems **2** (1984), pp. 145–154.

[10] Stafford, J. A. and A. L. Wolf, *Architecture-Level Dependence Analysis for Software Systems*, International Journal of Software Engineering and Knowledge Engineering **11** (2001), pp. 431–451. URL http://citeseer.ist.psu.edu/article/stafford98architecturelevel.html