Reasoning about Exception Flow at the Architectural Level

Fernando Castor Filho
Patrick Henrique da S. Brito
Cecília Mary F. Rubira

Technical Report - IC-06-10 - Relatório Técnico

May - 2006 - Maio

Reasoning about Exception Flow at the Architectural Level[⋆]

Fernando Castor Filho[⋆⋆], Patrick Henrique da S. Brito[⋆⋆⋆], and Cecília Mary F. Rubira[†]

Instituto de Computação
Universidade Estadual de Campinas
Caixa Postal 6176. CEP 13083-970, Campinas, SP, Brazil.
{fernando, patrick.silva, cmrubira}@ic.unicamp.br
+55 (19) 3788-5842 (phone/fax)

Abstract. An important challenge faced by the developers of fault-tolerant systems is to build fault tolerance mechanisms that are reliable. To achieve the desired levels of reliability, mechanisms for detecting and handling errors should be designed since the early phases of software development, preferably using a rigorous or formal methodology. In recent years, many authors have been advocating the idea that exception handling-related issues should be addressed at the architectural level, as a complement to implementation-level exception handling. However, few works in the literature have addressed the problem of describing how exceptions flow amongst architectural elements. A solution to this problem would enable the early detection of mismatches between architectural elements due to exceptions. Moreover, it would make it possible to validate whether the architecture satisfies some properties of interest regarding exception flow before the system is actually built. We believe that a model for describing the flow of exceptions between architectural elements should be: (i) precise; and (ii) analyzable, preferably automatically. In this paper, we present a rigorous model for reasoning about exception flow in software architectures that satisfies these requirements.

1 Introduction

Exception handling [14] is a well-known mechanism for structuring error recovery in fault-tolerant software systems. Since exception handling is an application-specific technique, it complements other techniques for improving system reliability, such as atomic transactions [19], and promotes the implementation of very specialized and sophisticated error recovery measures. Furthermore, in applications where a rollback is not possible, such as those that interact with mechanical devices, exception handling may be the only choice available.

Usually, a large part of a system's code is devoted to error detection and handling [14, 28, 34]. However, since developers tend to focus on the normal activity of

[⋆] The authors are partially supported by FINEP/Brazil (1843/04), project CompGov, which aims at developing a Shared Library of Components for e-Government.
[⋆⋆] Supported by FAPESP/Brazil, grant 02/139960-2.
[⋆⋆⋆] Supported by CAPES/Brazil.
[†] Partially supported by CNPq/Brazil under grant 351592/97-0, and by FAPESP/Brazil, under grant 2004/10663-8.

applications and only deal with code responsible for error detection and handling at the implementation phase, in an ad hoc manner, this part of the code is usually the least understood, tested, and documented [14, 28]. To achieve the desired levels of reliability, mechanisms for detecting and handling errors should be developed systematically from the early phases of software development [15, 30], starting from requirements, and passing by analysis, architectural design, and detailed design.

The concept of software architecture [32] has been recognized in the last decade as a means to cope with the growing complexity of software systems. According to Clements and Northrop [13], software architecture is the structure of the components of a program/system, their interrelationships and principles, and guidelines governing their design and evolution over time. It is widely accepted that the architecture of a software system has a large impact on its capacity to meet its intended quality requirements, such as reliability, security, availability, and performance, amongst others [6, 12]. There are many proposals in the literature [3][18][24][25] of notations and techniques to formally describe software architectures to show how they achieve specific quality attributes, such as adherence to interaction protocols [3], and dynamism [25]. These approaches are usually supported by tools that automatically or semi-automatically verify if an architecture satisfies some previously-defined properties of interest.

An important challenge faced by developers of fault-tolerant systems is to build fault tolerance mechanisms that are reliable. If a system should be reliable and exception handling is one of the mechanisms that will be employed to achieve this goal, it may be beneficial to consider exception handling-related issues during architectural design. This idea goes hand-in-hand with the notion that exception handling should be taken into account from the early phases of software development. However, to the best of our knowledge, there are currently no approaches for the specification and analysis of exception-related information at the architectural level. As pointed out by Bass and his coleagues [5], specifying how exceptions flow between architectural components is a real problem that appears in the development of systems with strict dependability requirements, such as air-traffic control and financial.

We believe an approach for describing software architectures extended with information about exception flow should be based on a formal model that supports reasoning about exception flow-related properties of interest. Furthermore, it should make it possible to automatically verify if an architecture satisfies these properties of interest. The ability to formally specify and verify the flow of exceptions in a system can help in the discovery of ambiguities, mistakes, and incompletenesses, thus improving the system's overall reliablity. In this paper we present a model for reasoning about exception flow in software architectures. This model specifies the structuring of an architecture in terms of components (loci of computation and data stores) and connectors (loci of interaction), as well as information relative to exception flow amongst these elements. We show how systems adhering to this model can be automatically verified using the Alloy [20] specification language and its associated tool set.

This work is organized as follows. Section 2 provides some background on exception handling and the Alloy design language. Section 3 describes the overall approach that we propose for extending architecture descriptions we exception flow-related information. Section 4 presents the proposed model for reasoning about the flow of exceptions at the architectural level. A mix of informal explanations and set theory notation is employed. Section 5 describes how the proposed model can be used to verify exception flow in architecture descriptions. Section 6 compares the proposed model with some related research. The last section rounds the paper and points directions for future works.

2 Background

2.1 Exception Handling

Exception handling [14] is a mechanism for structuring error recovery in software systems so that errors can be more easily detected, signaled, and handled. It is implemented by many mainstream programming languages, such as Java, Ada, C++, and C#. These languages allow the definition of exceptions and the corresponding handlers. The set of exceptions and exception handlers in a system define its abnormal, or exceptional, activity.

When an error is detected, an exception is generated, or *raised*. If the same exception may be raised in different parts of a program, it is possible that different handlers are executed. The choice of the handler that is executed depends on the exception handling context (EHC) where the exception was raised. An EHC is a region of a program where the same exceptions are handled in the same manner. Each context has an associated set of handlers that are executed when the corresponding exceptions are raised. Typical examples of EHCs in object-oriented languages are blocks, methods, and classes [17]. At the architectural level, contexts are usually defined by components and connectors [9].

The concept of *idealized fault-tolerant component* (IFTC) [4] defines a conceptual framework for structuring exception handling in software systems. An IFTC is a component (in a broader sense; an object, a software component, a whole system, etc.) in which the parts responsible for the normal and abnormal activities are separated and well-defined, within its internal structure. The goal of the IFTC approach is to provide means to structure systems so that the impact of fault tolerance mechanisms in the overall system complexity is minimized. This eases the detection and handling of errors. Figure 1 presents the internal structure of an IFTC and the types of messages it exchanges with other components in a system.

When an IFTC receives a service request, it produces a *normal response* if the request is successfully processed. If an IFTC receives an invalid service request, it

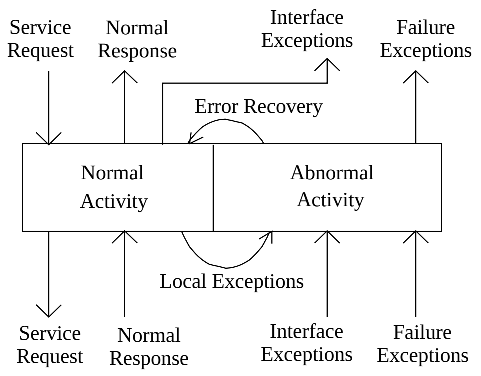


Fig. 1. Idealized Fault-Tolerant Component.

signals an *interface exception*. If an error is detected during the processing of a valid request, the normal activity part of the IFTC *raises* an *internal exception*, which is received by the exceptional activity part of the IFTC. If the IFTC is capable of handling an internal exception properly, normal activity is resumed. If the IFTC has no handlers for an internal exception or is unable to handle an exception for which it has a handler, it *signals* a failure exception. Interface and failure exceptions are collectively called *external exceptions*. An IFTC might also *catch* external exceptions signaled by other IFTCs and attempt to handle them. In this work, it is assumed that architectural elements behave like IFTCs. Hence, only external exceptions are taken into account, as strictly internal exceptions are not visible architecturally.

2.2 Alloy

Alloy [20] is a lightweight modeling language for software design. It is amenable to a fully automatic analysis, using the Alloy Analyzer (AA) [21], and provides a visualizer for making sense of solutions and counterexamples it finds. Similarly to other specification languages, such as Z and B [2], Alloy supports complex data structures and declarative models.

In Alloy, models are analyzed within a given scope, or size. The analysis performed by the AA is sound, since it never returns false positives, but incomplete, since the AA only checks things up to a certain scope. However, it is complete up to scope; the AA never misses a counterexample which is smaller than the specified scope. As pointed out by the Alloy tutorial [21], small scope checks are still very useful for finding errors.

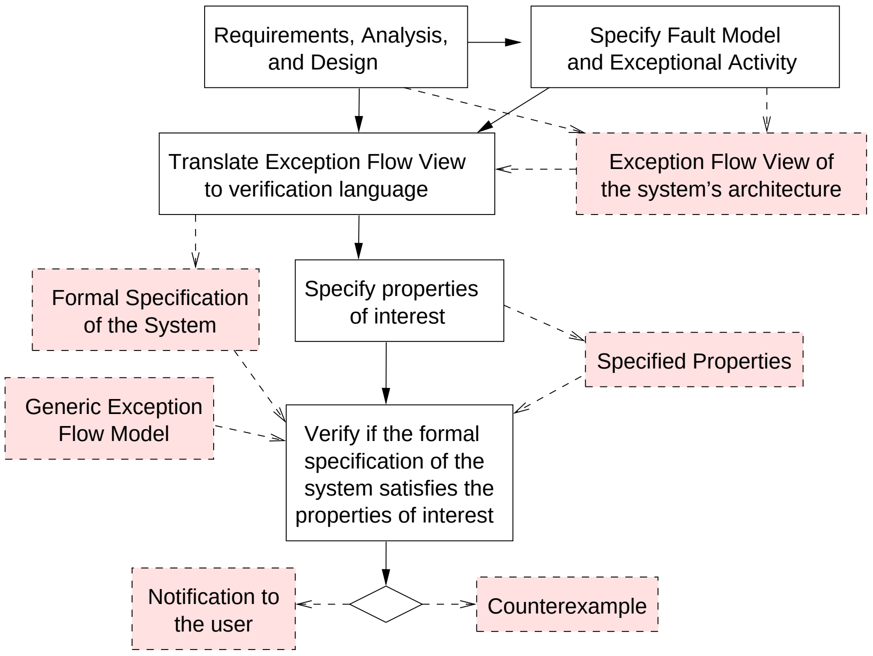
3 Proposed Approach

The construction of robust fault-tolerant systems requires that developers take fault tolerance-related issues into account since the early phases of development [15, 30]. Our ultimate goal is to devise a general approach for the rigorous development of dependable software systems that use exception handling to implement error recovery. This work addresses specifically the issue of verifying properties of interest related to exception flow in software architectures. Figure 2 presents a schematic description of the proposed approach for verifying exception flow at the architectural level. This approach is supported by the Aereal [8] framework. This framework aims assist in documenting, analyzing, and validating exception flow in software architectures. .

In the proposed approach, developers start by performing traditional activities of a software development process, namely, requirements engineering, analysis, and design (both architectural and detailed) of the system. At the same time, they define the scenarios in which the system may fail (fault model), what exceptions correspond to each type of error, and where and how the exceptions are handled (exceptional activity). The specification of the system's fault model and exceptional activity can be conducted as prescribed by some works in the literature [30]. The result of these activities is an architecture description of the system that includes information about the exceptions that can be signaled by each architectural element and what elements are responsible for handling them. We refer to this specialized architecture description as an Exception Flow View [8] of the system's software architecture. In Aereal, exception flow views are specified using the ACME [18] architecture description language [26] (ADL).

To verify if the exception flow view exhibits some properties of interest, it is necessary to translate it to a formal language with adequate support for automated verification (*verification language*). The formal specification produced by translating the exception flow view to the verification language must adhere to a generic meta-model specifying the elements that can be part of an architecture description, how exceptions flow amongst these elements, and how they relate (hereafter called *generic exception flow model*). Both the formal specification and the generic exception flow model are described in the verification language. Up to now, we specified a generic exception flow model using Alloy as verification language. A system is verified by providing its formal specification as input to a constraint solver for the verification language, together with the properties to be verified, and the definition of the generic exception flow model. We used the AA to verify formal specifications in Alloy. If any of the properties of interest does not hold, the AA produces a counterexample.

In the rest of this paper, we focus on specifying the generic exception flow model. A detailed description of the general approach described in this section is available elsewhere [8].



Requirements, Analysis, and Design
Specify Fault Model and Exceptional Activity
Translate Exception Flow View to verification language
Exception Flow View of the system's architecture
Formal Specification of the System
Specify properties of interest
Specified Properties
Generic Exception Flow Model
Verify if the formal specification of the system satisfies the properties of interest
Notification to the user
Counterexample

Fig. 2. Overview of the proposed approach. White rectangles represent activities and shaded rectangles with dashed borders represent artifacts.

4 Generic Exception Flow Model

We use special-purpose architectural connectors to model exception flow between components. These connectors, called exception ducts, are unidirectional point-to-point links through which only exceptions flow. They are orthogonal to "normal" architectural connectors and do not constrain the way in which the architecture is organized [8]. The idea that simple point-to-point connections are suitable general-purpose abstractions for modelling communication between architectural components independently of the architectural styles [32] to which an architecture adheres was proposed by Mehta and Medvidovic [27]. We have tailored this idea to our specific needs. This structural perspective on exception flow was adopted because it is intuitive to architects (who are used to thinking in terms of components and connectors), compatible with well-established views on what exception flow is [14], and does not require the modeling of the complete behavior of the application. Modeling issues such as data- and control-flow are beyond the scope of this work.

It is important to stress that we do not see excepton ducts as implementation-level connectors. They are just an abstraction to describe exception flow. Architects often need to understand the architecture of a system from various perspectives, according to specific quality attributes. *Architectural views* represent various aspects of the same architecture and each view shows how the architecture achieves a particular quality

Table 1. Elements of the proposed exception flow model.

Set	**Description**
Element	The type of all architectural elements. Supertype of *Component* and *Duct*.
Component	The type of which all components in a system are instances.
Duct	The type of which all exception ducts in a system are instances.
RootException	The type of which all exceptions in a system are instances.

attribute [6]. Since different views target different aspects of an architecture, they usually employ different modeling constructs [23]. Therefore, at the implementation level, exception flow would be materialized by means of the constructs provided by the underlying programming language or infrastructure. For example, in a publisher/-subscriber architecture, exception flow can be materialized as events flowing through a message bus.

4.1 Representation of Components, Ducts, and Exceptions

In our model, components, exception ducts, and exceptions are represented by objects of a certain type. The proposed model employs a simple notion of type that is adopted by some modern formal specification languages, such as Alloy [20] and B [2]. Moreover, it is compatible with the notion of types used in OO languages such as Java and C#. A type T is a set of instances and the subtypes $T_1, T_2 ... T_N$ of T are disjunct subsets of T. Only single inheritance is allowed. An exception is any instance of a type that is a subtype of type *RootException*. The same applies to components and exception ducts, and the types *Component* and *Duct*, respectively. Collectively, components and exception ducts are called "elements". Table 1 lists the basic elements of the proposed model. The sets in the table can also be seen as unary relations and are therefore subject to operations that apply to relations, such as composition.

We represent exceptions as objects, instead of using symbols or global variables, mainly because objects are more flexible and can be used to encode arbitrary information regarding the cause of an exception [17]. Moreover, many large and complex software systems are developed nowadays using object-oriented (OO) languages such as Java, C#, and C++.

The supertype of all exceptions is called *RootException*, instead of a more usual name, such as *Exception* or *Error*, to give developers the flexibility to organize exceptions as required, for instance, based on the adopted programming language. For example, to mimic the EHS of Java, a developer would define at least four exception types: (i) *Throwable*, subtype of *RootException*; (ii) *Exception*, subtype of *Throwable*; (iii) *Error*, subtype of *Throwable*; and (iv) *RuntimeException*, subtype of *Exception*. Application-specific exception types would then be subtypes of one of these types.

4.2 System Structure

We follow the general view of a system configuration as a finite connected graph of components and connectors [26]. We specialize this view, however, so that it can be used to reason about exception flow. In our model, a component is a structural element that catches and/or signals exceptions and an exception duct is a structural element that represents flow of exceptions between two components. The structure of a system is defined in terms of connections between components and exception ducts. The relations $CatchesFrom \in Element \leftrightarrow Element$ and $SignalsTo \in Element \leftrightarrow Element$ specify these connections.

Given an element B, $\{B\}.CatchesFrom$ yields the set of elements that signal exceptions that B catches. Conversely, $\{B\}.SignalsTo$ yields the set of elements that catch exceptions that C signals. The "." operator represents relational composition (or join). Given two relations $A \subseteq T_1 \times T_2 \times ... \times T_n$ and $B \subseteq T_n \times T_{n+1} \times ... \times T_{n+m}$, $A.B$ yields a relation $C \subseteq T_1 \times T_2 \times ... \times T_{n-1} \times T_{n+1} \times ... \times T_{n+m}$. Relation C comprises all the tuples formed by combining tuples from A and B whenever the last element of a tuple from A is the same as the last element of a tuple from B. For example, given $A = \{(e_1, e_2), (e_2, e_3)\}$ and $B = \{(e_2, e_4), (e_2, e_5), (e_3, e_6), (e_7, e_8)\}$, $A.B$ yields $C = \{(e_1, e_4), (e_1, e_5), (e_2, e_6)\}$.

Table 2 lists some constraints on relations $CatchesFrom$ and $SignalsTo$. These constraints specify properties that a system specification adhering to the proposed model should exhibit. Each one is identified by a name matching the pattern BPX, where "BP" stands for basic property and "X" is a positive integer. Properties $BP1$ and $BP2$ specify that the $CatchesFrom$ relation is not reflexive and that it never associates elements of the same type, respectively. Properties $BP3$ and $BP4$ do the same for relation $SignalsTo$. Property $BP5$ states that exception ducts signal exceptions to exactly one element and catch exceptions from exactly one element. If B is a component, $\{B\}.CatchesFrom$ may yield an empty set, in which case B does not catch exceptions. If $\{B\}.SignalsTo$ yields an empty set, exceptions signaled by B, if any, are caught by an implicit component $OperatingSystem$. This is useful to model situations in which a system is not capable of handling a certain error and fails catastrophically by signaling an exception to the operating system. Properties $BP6$ states that the set of elements from which an element catches exceptions consists of all elements that signal exceptions to it. Property $BP7$ states that the set of elements to which an architectural element signals exceptions consists of all elements that catch exceptions from it, respectively. These two properties provide a link between $CatchesFrom$ and $SignalsTo$. Property $BP8$ specifies that the elements from which an architectural element catches exceptions are different from the ones to which it signals exceptions.

For any valid system in the proposed model, the graph formed by using the components of the system as vertices and the ducts as edges is connected. More formally, let $G = (Component, Duct)$ be a graph, where $Component$ is the set of

Table 2. Contraints on the *CatchesFrom* and *SignalsTo* relations.

Property	**Constraint**
$BP1$	$\forall B \in Element \bullet B \notin \{B\}.CatchesFrom$
$BP2$	$\forall B' \in Element \bullet (B, B') \in CatchesFrom \Rightarrow \neg(B \in Duct \wedge B' \in Duct) \wedge \neg(B \in Component \wedge B' \in Component)$
$BP3$	$\forall B \in Element \bullet B \notin \{B\}.SignalsTo$
$BP4$	$\forall B' \in Element \bullet (B, B') \in SignalsTo \Rightarrow \neg(B \in Duct \wedge B' \in Duct) \wedge \neg(B \in Component \wedge B' \in Component)$
$BP5$	$\forall D \in Duct \bullet \lvert\{D\}.CatchesFrom\rvert = 1 \wedge \lvert\{D\}.SignalsTo\rvert = 1$
$BP6$	$\forall B_1, B_2 \in Element \bullet B_1 \in \{B_2\}.CatchesFrom \Rightarrow B_2 \in \{B_1\}.SignalsTo$
$BP7$	$\forall B_1, B_2 \in Element \bullet B_1 \in \{B_2\}.SignalsTo \Rightarrow B_2 \in \{B_1\}.CatchesFrom$
$BP8$	$\forall B \in Element \bullet \{B\}.CathcesFrom \cap \{B\}.SignalsTo = \{\}$
$BP9$	$\forall C_1, C_2 \in Component \bullet C_1 \in \{C_2\}. * ((SignalsTo \cup CatchesFrom).(SignalsTo \cup CatchesFrom))$

all vertexes and *Duct* is the set of all edges. An edge $D \in Duct$ connects two vertexs $C_1 \in Component$ and $C_2 \in Component$ if $D \in \{C_1\}.CatchesFrom$ and $C_2 \in \{D\}.CatchesFrom$. In order for a graph to be connected, there must be a path between any two vertexes. Property $BP9$ specifies this constraint formally. It states that the reflexive transitive closure ($*$ operator) of any component with respect to the relation $(SignalsTo \cup CatchesFrom).(SignalsTo \cup CatchesFrom)$ contains all the other components of the system. In property $BP9$, the $*$ operator yields the set of all components reachable by composing component C_2 with $(SignalsTo \cup CatchesFrom).(SignalsTo \cup CatchesFrom)$ zero or more times.

4.3 Exception Interfaces and Exception Handling Contexts

As mentioned in previously, we consider a component to be a structural element that catches and signals exceptions. Exception ducts are similar, but simpler. A component includes (i) a collection of exception interfaces, which specify the exceptions the component signals; and (ii) a collection of EHCs, which define regions where exceptions are always handled in the same way. Exception interfaces are associated to components by the *SignalsTo* relation and, for each exception duct in the set $\{C\}.SignalsTo$, there is a corresponding exception interface. The same applies for the *CatchesFrom* relation and EHCs. This represents the fact that a component may signal/catch different exceptions to/from the different exception ducts it is connected to. As imposed by property $BP5$ (Table 2), each exception duct has exactly one exception interface and one EHC.

Models for reasoning about exception flow at the programming language level do not have an explicit separation between exception interfaces and EHCs. This separation is not necessary because these models focus on fine-grained programming constructs, like methods and procedures, where multiple contexts are associated to a single exception interface. At the architectural level, however, this separation is very

Table 3. Contraints on the $PortMap$ relation.

Property	Constraint
$BP10$	$\forall B \in Element \bullet dom(\{B\}.PortMap) = \{B\}.CatchesFrom \wedge ran(\{B\}.PortMap) = \{B\}.SignalsTo$

important, since a component can have multiple access points (ports) and the latter are explicit in the system description.

The $PortMap \in Element \leftrightarrow Element \leftrightarrow Element$ relation associates exception interfaces and EHCs. When an element catches an exception and does not handle it, the $PortMap$ relation specifies to which element the exception will be signaled, based on the element that originally signaled the exception. $PortMap$ associates architectural elements to EHCs and exception interfaces, based on the elements to which these EHCs and interfaces correspond. Table 3 lists the single property associated to the $PortMap$ relation. Property $BP10$ specifies that for every element B, $PortMap$ associates all the elements from which B catches exceptions to some element to which it signals exceptions and vice-versa.

4.4 Exception Flow

Exception flow is specified in terms of five relations: $Signals \in Element \leftrightarrow Element \leftrightarrow RootException$, $Raises \in Element \leftrightarrow Element \leftrightarrow RootException$, $Propagates \in Element \leftrightarrow Element \leftrightarrow RootException \leftrightarrow RootException$, $Catches \in Element \leftrightarrow Element \leftrightarrow RootException$, and $Handles \in Element \leftrightarrow Element \leftrightarrow RootException$. The first two concern the exception interfaces of an architectural element, whereas the last three are related to EHCs. In the rest of this section, we describe these relations in more detail.

The $Signals$ relation defines the exception interfaces of an architectural element. This relation specifies which exceptions an architectural element signals and the elements that catch these exceptions. Let ES be a set of exceptions and B_1 and B_2 be architectural elements such that $B_2 \in \{B_1\}.SignalsTo$. If $\{B_2\}.(\{B_1\}.Signals) = ES$, we say that the element B_1 signals exceptions ES to element B_2. Table 4 lists some properties associated to the $Signals$ relation. Property $BP11$ specifies that elements only signal exceptions to elements to which they are connected, as specified by the $SignalsTo$ relation.

Even though, for the sake of uniformity, we have called the second constraint on Table 4 a "property", it works more as a definition of the $Signals$ relation. Property $BP12$ states that the $Signals$ relation is derived from three other relations. Intuitively, the set of exceptions that a component signals depends on the exceptions it generates (raises) and on exceptions it catches that were signaled by other architectural elements. $Propagated$ and $Unhandled$ are auxiliary relations defined in terms

Table 4. Contraints on the *Signals*, *Raises*, *Catches*, *Handles*, and *Propagates* relations.

Property	**Constraint**
$BP11$	$\forall B \in Element \bullet dom(\{B\}.Signals) \subseteq \{B\}.SignalsTo$
$BP12$	$Signals = Raises \cup Propagated \cup Unhandled$
$BP13$	$Raises \subseteq Signals$
$BP14$	$\forall B \in Element \bullet dom(\{B\}.Catches) \subseteq \{B\}.CatchesFrom$
$BP15$	$\forall B \in Element \bullet \forall B' \in \{B\}.CatchesFrom \bullet$ $\{B\}.(\{B'\}.Signals) = \{B'\}.(\{B\}.Catches)$
$BP16$	$\forall B \in Element \bullet dom(\{B\}.Handles) \subseteq \{B\}.CatchesFrom$
$BP17$	$\forall B \in Element \bullet dom((\{B\}.Propagates).RootException) \subseteq$ $\{B\}.CatchesFrom$
$BP18$	$\forall B \in Element \bullet \forall B' \in \{B\}.CatchesFrom \bullet \lvert\{B'\}.(\{B\}.Propagates)\rvert > 0$ $\Rightarrow (dom(\{B'\}.(\{B\}.Propagates)) \cap \{B'\}.(\{B\}.Handles)) = \{\}$

of the relations that specify a component's EHCs (described in the following paragraphs). The *Raises* relation specifies the exceptions that components generate when erroneous conditions are detected. These conditions are dependent on the semantics of the application and on the assumed failure model. For reasoning about exception flow, the fault that caused an exception to be raised is not important, just the fact that the exception was raised. Let ES be a set of exceptions and B_1 and B_2 be architectural elements such that $B_2 \in \{B_1\}.SignalsTo$. If $\{B_2\}.(\{B_1\}.Raises) = ES$, we say that the element B_1 raises exceptions ES to B_2. The third property of Table 4 specifies that all the exceptions an element raises to another element are also signaled to the latter. This is coherent with the view that only external exceptions matter at the architectural level.

Exception handling contexts are defined in terms of three relations: *Catches*, *Handles*, and *Propagates*. *Catches* specifies, for an arbitrary element B, the exceptions B receives from the elements in the set $\{B\}.CatchesFrom$. Let ES be a set of exceptions and B_1 and B_2 be architectural elements such that $B_2 \in \{B_1\}.CatchesFrom$. If $\{B_2\}.(\{B_1\}.Catches) = ES$, we say that the element B_1 catches exceptions ES from element B_2. Table 4 shows the basic properties associated with *Catches*, *Handles*, and *Propagates*. Propety $BP14$ specifies that elements only catch exceptions from elements to which they are connected, as specified by the *CatchesFrom* relation. Property $BP15$ states that the exceptions that an element catches are the same as the exceptions signaled to it.

The *Handles* relation specifies the exceptions that are handled by a component. By "handled", we mean that the component is capable of taking some action that stops the propagation of the exception and makes it possible for the system to resume its normal activity. Modeling the behavior of the exception handlers is beyond the scope of this work. We are just interested in the effect the handler has on the flow of exceptions. Let ES be a set of exceptions and B_1 and B_2 be architectural elements such that $B_2 \in \{B_1\}.CatchesFrom$. If $\{B_2\}.(\{B_1\}.Handles) = ES$, we say that the

Table 5. Properties that define auxiliary relations *Unhandled* and *Propagated*.

Property	**Constraint**
$BP19$	$Propagated = \{\ T \in Element \times Element \times RootException \mid s(T) \in \{f(T)\}.SignalsTo \wedge t(T) \in (((\{f(T)\}.PortMap).\{s(T)\}).(\{f(T)\}.Catches \backslash \{f(T)\}.Handles)).(((\{f(T)\}.PortMap).\{s(T)\}).(\{f(T)\}.Propagates))\ \}$
$BP20$	$Unhandled = \{\ T \in Element \times Element \times RootException \mid s(T) \in \{f(T)\}.SignalsTo \wedge t(T) \in (((\{f(T)\}.PortMap).\{s(T)\}).(\{f(T)\}.Catches \backslash \{f(T)\}.Handles)) \backslash ((((\{f(T)\}.PortMap).\{s(T)\}).(\{f(T)\}.Propagates)).(\{s(T)\}.(\{f(T)\}.Propagated)))\ \}$

element B_1 handles exceptions ES from element B_2. Only property $BP16$ in Table 4 is directly associated to the $Handles$ relation. This property states that elements only handle exceptions signaled by elements to which they are connected. We could also restrict $Handles$ to be a subset of $Catches$ just like $Raises$ is a subset of $Signals$. We do not impose this restriction, however, because sometimes it is useful to specify general handlers, that is, handlers capable of dealing with any exception.

The $Propagates$ relation is a variation of $Handles$ where the handler ends its execution by raising an exception. $Propagates$ specifies a cause-consequence relationship between an exception that an element catches and an exception that it signals. Let E and E' be exceptions and B_1 and B_2 be architectural elements such that $B_2 \in \{B_1\}.CatchesFrom$. If $\{B_2\}.(\{B_1\}.Propagates) = (E, E')$, we say that the element B_1 **explicitly propagates** (or simply "propagates") exception E' from E, signaled by B_2. The last two properties in Table 4 are directly related to the $Propagates$ relation. Property $BP17$ states that an element can only propagate exceptions signaled by an element from which it catches exceptions. Property $BP18$ specifies that an element can handle or propagate the exceptions it catches from another element, but not both.

Now we can go back to the definition of $Signals$ and define $Propagated$ and $Unhandled$. The $Propagated \in Element \leftrightarrow Element \leftrightarrow RootException$ relation specifies the subset of $Signals$ comprising exceptions that are explicitly propagated by an element. It associates an element to the exceptions it propagates explicitly and the elements that catch these propagated exceptions. The $Unhandled \in Element \leftrightarrow Element \leftrightarrow RootException$ relation associates the set of exceptions that an architectural element **implicitly propagates** and the elements to which these exceptions are signaled. An exception is said to be implicitly propagated when an architectural element catches it but does handle it or explicitly propagate an exception from it. Such an exception ends up being signaled to some other architectural element. Table 5 presents formal definitions for relations $Propagated$ and $Unhandled$. The constraints in the table use three auxiliary functions, $f()$, $s()$, and $t()$, that take a triple as argument and return the first, second, and third elements of the triple, respectively.

4.5 Exception Propagation Cycles

Informally, an exception propagation cycle is a situation where an exception is propagated (implicitly or explicitly) indefinitely, without ever being handled, not even by the special *OperatingSystem* component, in an architecture that adheres to basic properties $BP1 - BP20$. Furthermore, exceptions propagated in an exception propagation cycle might potentially have never been raised. For these reasons, valid exception flow views are not allowed to have exception propagation cycles.

A conservative way of preventing the occurrence of exception propagation cycles is to completely disallow structural cycles in the graph formed by the components and exception ducts in a system. For most systems, this solution is sufficient without being overly restrictive. However, for software architectures where components are peers, like multi-agent and publisher-subscriber, this approach is not acceptable. At the implementation level, exception propagation cycles are not a problem. In languages that support exception handling, such as Java and Ada, each EHC is kept in the stack, which is finite, and removed from it when controls returns from the EHC (possibly due to exception propagation). Therefore, in a language-level exception propagation cycle, eventually all the EHCs will be removed from the stack and exception propagation will stop. At the architectural level, however, this is not always the case, as an exception is not necessarily implemented as a language-level exception [9, 8]. In the rest of this section, we formalize the concept of exception propagation cycle and show how such cycles can be easily detected.

An **exception propagation** is a tuple $\phi = (B, E, E', B')$, with $B, B' \in Element$ and $E, E' \in RootException$. We use the functions $f()$, $s()$, $t()$, and $g()$ to obtain the first, second, third, and fourth elements of a propagation, respectively. Any propagation ϕ must satisfy the following well-formedness predicate:

$$\begin{aligned} g(\phi) \neq f(\phi) \wedge g(\phi) \in \{f(\phi)\}.SignalsTo \wedge f(\phi) \in \{g(\phi)\}.CatchesFrom \wedge \\ t(\phi) \in \{f(\phi)\}.(\{g(\phi)\}.Catches) \wedge t(\phi) \in \{g(\phi)\}.(\{f(\phi)\}.Signals) \wedge \\ \exists CF \in \{f(\phi)\}.CatchesFrom \bullet (CF, g(\phi)) \in \{f(\phi)\}.PortMap \wedge \\ s(\phi) \in \{CF\}.(\{f(\phi)\}.Catches) \wedge s(\phi) \notin \{CF\}.(\{f(\phi)\}.Handles) \wedge \\ (s(\phi) \in dom(\{CF\}.(\{f(\phi)\}.Propagates)) \Rightarrow \\ t(\phi) \in \{s(\phi)\}.(\{CF\}.(\{f(\phi)\}.Propagates))) \end{aligned}$$

It is easy to compute the set of all exception propagations in a software architecture adhering to the proposed model. Figure 3 presents an algorithm for computing all the exception propagations associated to an architectural element.

In order to impose a partial order between two exceptions propagations, we introduce the notion of consecutiveness. Two propagations ϕ_1 and ϕ_2 are said to be **consecutive** if they satisfy the following predicate:

$$g(\phi_1) = f(\phi_2) \wedge f(\phi_1) \in \{f(\phi_2)\}.CatchesFrom \wedge$$

```
void computePropagations(Element B) {
  foreach catchesP in B.Catches and not in B.Handles {
  // catchesP is an (Element, RootException) pair
    Propagation prop = new Propagation();
    foreach portMapP in B.PortMap such that portMapP.f() == catchesP.f() {
    // portMapP is an (Element, Element) pair
      if(there is a Triple propagatesT in B.Propagates such that
        propagatesT.f() == catchesP.f() && propagatesT.s() == catchesP.s()) {
      // propagatesT is an (Element, RootException, RootException) triple
           prop.t = propagatesT.t();
      } else { prop.t = catchesP.s(); }
      prop.s = catchesP.s();
      prop.f = B;
      prop.g = portMapP.s();
    }
    B.propagations.add(prop);
  }
}
```

Fig. 3. An algorithm for computing all the exception propagations associated to an element.

$$f(\phi_2) \in f(\phi_1).SignalsTo \wedge f(\phi_1) \neq f(\phi_2) \wedge t(\phi_1) = s(\phi_2) \wedge$$
$$t(\phi_1) \in \{f(\phi_1)\}.(\{f(\phi_2)\}.Catches) \wedge s(\phi_2) \in \{f(\phi_2)\}.(\{f(\phi_1)\}.Signals)$$

In this case, ϕ_1 is said to be the predecessor of ϕ_2 and ϕ_2 is the successor of ϕ_1. We indicate that propagations ϕ_1 and ϕ_2 are consecutive with the notation $\phi_1 \rightharpoonup \phi_2$. A **sequence of propagations** φ of length n is a set of propagations $\phi_1, \phi_2, \phi_3 ..., \phi_{n-1}, \phi_n$ such that $\phi_1 \rightharpoonup \phi_2, \phi_2 \rightharpoonup \phi_3, ..., \phi_{n-1} \rightharpoonup \phi_n$. For simplicity, we assume that all sequences of propagations are finite. A sequence of propagations $\varphi = \phi_1, \phi_2, ..., \phi_n$ forms an **exception propagation cycle** iff $\phi_n \rightharpoonup \phi_1$.

To verify if a software architecture has exception propagation cycles, it is necessary to build the directed graph formed by all the exception propagations of the architecture. This graph is constructed in two steps: (1) compute the exception propagations for all the elements in the architecture and use them as vertexes; and (2) create a directed edge between two propagations whenever they are consecutive, from the predecessor to the sucessor. Detecting exception propagation cycles in the resulting graph is only a matter of using a regular algorithm for finding cycles in directed graphs.

5 Materializing the Model

We have translated the generic exception flow model described in Section 4 to Alloy. In this section, we show how to specify systems adhering to this model and how to perform verification. We use a well-known textbook example [33] to make the explanation more concrete.

Figure 4 shows the components and connectors view of a control system for the mining environment [30]. Rectangles represent architectural components and arrows

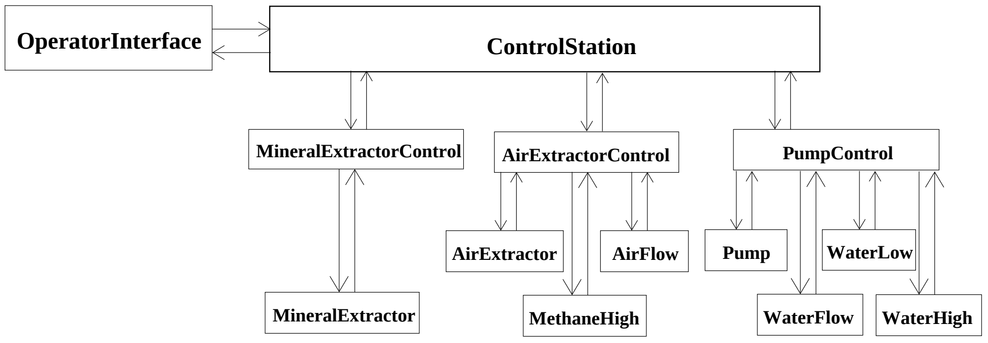


Fig. 4. Layered architecture of a mining control system.

represent data flow. The extraction of minerals from a mine produces water and releases methane gas to the air. The mining control system is used to drain mine water from a sump to the surface, and to extract air from the mine when the methane level becomes high. The system consists of three control stations: one that monitors the level of water in the sump, one that monitors the level of methane in the mine, and another that monitors the mineral extraction. For safety reasons, the extraction of minerals should be interrupted when the amount of methane in the atmosphere exceeds a safety limit. The air extractor control station monitors the level of methane inside the mine, and when the level is high an air extractor is switched on to remove air from the mine. The whole system is controlled from the surface via an operator console.

The system can fail in several ways. For simplicity, we only consider the case where the AirExtractorControl component fails by signaling the exception `AirExtractorOffException`. This exception is caught and handled by the `ControlStation` component. The handler ends its execution by propagating the exception `EmergencyException`. A detailed description of the exceptional activity of the mining system is available elsewhere [30].

Figure 5 shows the Alloy specification of the mining system. In Alloy, a signature (`sig` keyword) specifies a type. The `one` keyword indicates that a signature has exactly one instance. We use signatures for modeling structural elements and exceptions. The relations defined in Section 4, such as $CatchesFrom$, $Handles$, etc., are explicitly instantiated by means of facts, predicates that the AA must assume to be true when evaluating constraints. For instance, the fact `SystemStructure` in the snippet above states, among other things, that component `ControlStation` catches exceptions from the exception duct `AEC_CS`, which connects it to the `AirExtractorControl` component. Moreover, the fact `ExceptionFlow` states, among other things, that component `ControlPanel` catches the exception `AirExtractorControl`, signaled by the `AEC_CS` duct, and signals exception `EmergencyException` to the `CS_OI` duct. `ControlStation` propagates the latter exception from the former. The `open` clause in the beginning of the specification imports the definitions of the basic types of the generic exception

```
open ExceptionHandlingSystem
one sig AirExtractorOffException, EmergencyException extends RootException{}
one sig ControlStation, OperatorInterface, AirExtractorControl
    extends Component{}
one sig CS_OI, AEC_CS extends Duct{}
...
fact SystemStructure{
  ControlStation.CatchesFrom = AEC_CS
  ControlStation.SignalsTo = CS_OI
  AEC_CS.CatchesFrom = AirExtractorControl
  AEC_CS.SignalsTo = ControlStation
  ... }
fact ExceptionFlow{
  AirExtractorControl.Signals = AEC_CS->AirExtractorOffException
  AirExtractorControl.Raises = AEC_CS->AirExtractorOffException
  AEC_CS.Catches = AirExtractorControl->AirExtractorOffException
  AEC_CS.Signals = ControlStation->AirExtractorOffException
  ControlPanel.Catches = AEC_CS->AirExtractorOffException
  ControlPanel.Signals = CS_OI->EmergencyException
  no ControlPanel.Handles
  ControlPanel.Propagates =
      AEC_CS->AirExtractorControlOffException->EmergencyException
  ... }
fact PortMap{
  ControlPanel.PortMap = AEC_CS->CS_OI
  ...}
```

Fig. 5. Partial Alloy specification of the mining control system.

flow model, `Element`, `Component`, `Duct`, and `RootException`. Moreover, it imports the predicates that specify the basic properties defined in Section 4.

Verification consists in checking if the Alloy specification of a system satisfies Alloy predicates corresponding to properties of interest. The properties of interest that a system must satisfy are split in three categories: basic, desired, and application-specific. Basic properties define the well-formedness rules of the model, the characteristics of valid systems. They specify the functioning of the exception handling mechanism and how software architectures are structured. We have formally specified all the basic properties of the generic exception flow model in Section 4.

Desired properties are general properties that are usually considered beneficial, although they are not part of the basic exception handling mechanism. They assume that the basic properties hold. Some examples are the following.

DP1. Architectural elements do not handle exceptions they do not catch.
DP2. All the exceptions caught by an architectural element are handled by it, even if some of its handlers end their execution by raising exceptions.
DP3. No unhandled exceptions.

Application-specific properties are rules regarding the flow of exceptions in a specific application. For the mining system, a possible application-specific property is the following, which guarantees that the OperatorInterface component does not receive domain-specific exceptions.

AP1. No architectural element signals to the OperatorInterface component an exception different from `EmergencyException`.

The Alloy definition of the generic exception flow model includes the specifications of several basic and desired properties that can be used "as-is". Developers only specify additional desired properties and application-specific properties, if any. The AA is employed to analyze exception flow. If a property of interest is violated, the AA generates a counterexample with a configuration of the system for which the violated property of interest does not hold. Otherwise it notifies the user that the system may be valid.

Figure 6 defines four Alloy predicates named `bp13`, `dp1`, `dp2`, and `ap1`, formally specifying properties $BP13$, $DP1$, $DP2$, and $AP1$, respectively. Alloy predicates are logic sentences that must be checked by the AA. In the body of the predicates, `Raises`, `Signals`, `Catches`, `Propagates`, `Handles`, and `CatchesFrom` are names of relations corresponding to the homonymous relations described in Section 4. Predicate `bp13()` states that the set of exceptions that a component raises is a subset of the exceptions it signals. Predicate `dp1()` specifies that the set of exceptions that a component handles is a subset of the exceptions it catches. The operators `all`, `<:`, `&&`, and `in` represent, respectively, universal quantification, domain restriction, logical conjunction, and subset. Predicate `dp2()` selects, for each component in the Alloy specification, all the exceptions that the component catches but does not handle and checks if exceptions are propagated from them. The operators `-`, `=>`, and `#` mean set subtraction, logical implication, and set cardinality, respectively, and the declaration `let` associates an alias to an expression. Predicate `ap1()` is a direct translation from the informal description of property $AP1$.

6 Related Work

Several works propose static analyses of source code that generate information about exception flow. Usually, this information consists in the exception propagation paths in a program and is used, for example, to discover uncaught exceptions in languages with polymorphic types, such as ML. Chang et al [11] present a set-based static analysis of Java programs that estimates their exception flows. This analysis is used to detect too general or unecessary exception specifications and handlers. Yi [35] proposes an abstract interpretation that estimates uncaught exceptions in ML programs. Fähndrich [16] and coleagues have employed their BANE toolkit to discover uncaught exceptions in ML. Schaefer and Bundy's [31] work describes a model for reasoning about exception flow in Ada programs. This model is used by a tool that tracks down uncaught exceptions and provides exception flow information to programmers. In a similar vein is the research of Robillard and Murphy [29] in the JEX tool, which ana-

```
/* Basic property BP13 */
pred bp13() {  all C : Component | (C.Raises in C.Signals)  }
/* Desired property DP1 */
pred dp1() {
  all C : Component | all D : Duct | D in C.CatchesFrom &&
  D.(C.Handles) in D.(C.Catches) &&
    (D.(C.Catches))<:(D.(C.Propagates))=D.(C.Propagates)
}
/* Desired property DP2 */
pred dp2() {
 all C: Component | let nonHandled = (C.Catches - C.Handles)
  | (all CF : C.CatchesFrom | #(CF <: nonHandled) > 0 =>
     ((#nonHandled > 0 => #(C.Propagates) > 0) &&
      all E: CF.nonHandled |  #(E.(CF.(C.Propagates))) > 0))
}
/* Application-specific property AP1 */
pred ap1() {
 all D: OperatorInterface.CatchesFrom |
   OperatorInterface.(D.Signals) = EmergencyException
}
```

Fig. 6. Alloy specifications of properties $BP13$, $DP1$, $DP2$, and $AP1$.

lyzes exception flow in Java programs. The tool includes a GUI to display a program's exception propagation paths and detects handlers that are too general.

Our approach leverages previous proposals for exception flow analysis, most notably Schaefer and Bundy's [31], but differs in focus. The proposed approach targets the early phases of development and is broader in scope. It describes how an architectural-level exception handling mechanism works and leverages existing verification tools to check for adherence to the rules prescribed by this mechanism. Furthermore, it supports the definition of new properties of interest and their automated verification. Moreover, as mentioned in Section 4.5, existing exception flow models do not take exception propagation cycles into consideration, as they are not a problem that occurs at the implementation level.

Jiang and coleagues [22] describe an approach for the analysis of exception propagation based on a data structure called exception propagation graph. The goal of the authors is to use exception propagation graphs mainly as a basis for automatically generating structural tests. They do not address exception propagation cycles, as their work focuses on implementation-level exception flow analysis. Moreover, they do not show how the proposed approach can be employed to check whether a system exhibits some properties of interest, such as absence of useless handlers.

Several approaches for specifying software architectures so that they are passive to automated analysis have been proposed. Most of them define new ADLs that target specific aspects of a software system. These ADLs are usually based on some underlying formalism that is well-supported by tools. Wright [3] specifications can be translated to CSP and analyzed for deadlock freedom and interface compatibility. Rapide [24] is based on partially-ordered event sets. The language supports simulation of architecture descriptions and analysis of the event patterns produced by components. Darwin [25] is based on π-calculus and can be used to specify dynamic software architectures. Abowd and his coleagues [1] use Z to formalize and compare architectural styles. Ours is yet another work along this line. It emphasizes the specification of exception flow at the architectural level, uses an existing ADL which supports extension, ACME, and is based on the Alloy specification language. To the best of our knowledge, no ADLs currently available focus on the verification of properties related to exception flow.

In a previous work, Castor and coleagues [7] described an initial version of the model presented in this paper. This early work does not unify the definitions of components and ducts. As a consequence, it is harder to use and less scalable. Furthermore, it does not take exception propagation cycles into account. Finally, it is not integrated with a more general approach for the rigorous development of fault-tolerant systems that uses exception handling as a mechanism to implement error recovery. In a recent paper [10], Castor et al describe a model for reasoning about exception flow in cooperative concurrent systems. This work is complementary to ours, as it addresses a different class of systems.

7 Concluding Remarks

This paper presented a model for reasoning about the flow of exceptions at the architectural level. This model is part of the Aereal framework. We have shown that the model supports the specification of several properties of interest related to exception flow. Furthermore, we have described how systems adhering to it can be automatically analyzed using the AA, in order to verify whether they exhibit these properties.

In another study [8], we assessed the scalability of the proposed model. We discovered that, for software architectures with a large number of exceptions (30+), it does not scale up well. Therefore, our most immediate future work is to improve scalability. We envision two complementary approaches. The first is to optimize the system model by removing redundant information. For example, relations $Catches$ and $Signals$ could be unified into a single abstraction. The two exist only because of historical reasons. The same applies for $CatchesFrom$ and $SignalsTo$. The second is to implement a tool that checks if an Alloy specification satisfies all the basic properties of the EHS supported by Aereal. This would drastically reduce the complexity of the checks the AA performs, hence decreasing the amount of memory that verification requires. This change will not compromise the flexibility of the framework, since the basic properties do not change and any valid system must satisfy them.

References

1. G. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM TOSEM*, 4(4):319–364, October 1995.

2. J. R. Abrial. *The B-Book Assigning Programs to Meanings*. Cambridge U. Press, 1995.
3. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.
4. T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Springer-Verlag, 2nd edition, 1990.
5. L. Bass, P. C. Clements, and R. Kazman. Air traffic control: A case study in designing for high availability. In *Software Architecture in Practice*, chapter 6. Addison-Wesley, 2nd edition, 2003.
6. L. Bass, P. C. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003.
7. F. Castor Filho, P. H. da S. Brito, and C. M. F. Rubira. Modeling and analysis of architectural exceptions. In *Proceedings of the FM'2005 Workshop on Rigorous Engineering of Fault-Tolerant Systems*, pages 112–121, July 2005.
8. F. Castor Filho, P. H. da S. Brito, and C. M. F. Rubira. Specification of exception flow in software architectures. *Journal of Systems and Software*, 2006.
9. F. Castor Filho, P. A. de C. Guerra, and C. M. F. Rubira. An architectural-level exception-handling system for component-based applications. In *Proceedings of the 1st LADC*, LNCS 2847, pages 321–340. Springer-Verlag, October 2003.
10. F. Castor Filho, A. Romanovsky, and C. M. F. Rubira. Verification of coordinated exception handling. In *Proceedings of the 21st ACM Symposium on Applied Computing*, April 2006. To appear.
11. B.-M. Chang, J.-W. Jo, K. Yi, and K.-M. Choe. Interprocedural exception analysis for java. In *Proceedings of the 16th ACM Symposium on Applied Computing*, 2001.
12. P. C. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures*. Addison-Wesley, 2003.
13. P. C. Clements and L. Northrop. Software architecture: An executive overview. Technical Report CMU/SEI-96-TR-003, SEI/CMU, February 1996.
14. F. Cristian. Exception handling. In T. Anderson, editor, *Dependability of Resilient Computers*, pages 68–97. Blackwell Scientific Publications, 1989.
15. R. de Lemos and A. Romanovsky. Exception handling in the software lifecycle. *International Journal of Computer Science and Engineering*, 16(2):167–181, 2001.
16. M. Fahndrich et al. Tracking down exceptions in standard ml. Technical Report CSD-98-996, University of California, Berkeley, 1998.
17. A. Garcia, C. Rubira, A. Romanovsky, and J. Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197–222, November 2001.
18. D. Garlan et al. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, chapter 3. Cambridge U. Press, 2000.
19. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
20. D. Jackson. Alloy: A lightweight object modeling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, April 2002.
21. D. Jackson. Alloy home page, March 2006. Available at `http://sdg.lcs.mit.edu/alloy/default.htm`.
22. S. Jiang, B. Xu, and L. Shi. An approach to analyzing exception propagation. In *Proceedings of the 8th IASTED International Conference on Software Engineering and Applications*, Cambridge, USA, November 2004.
23. P. Krüchten. The 4+1 view model of software architecture. *IEEE Software*, pages 42–50, November 1995.
24. D. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, April 1995.
25. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, September 1995.
26. N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of Joint 5th ACM SIGSOFT FSE/6th ESEC*, pages 60–76, September 1997.
27. N. R. Mehta and N. Medvidovic. Composing architectural styles from architectural primitives. In *Proceedings of Joint 9th ESEC/11th ACM SIGSOFT FSE*, pages 347–350, September 2003.
28. D. Reimer and H. Srinivasan. Analyzing exception usage in large java applications. In *Proceedings of ECOOP'2003 Workshop on Exception Handling in Object-Oriented Systems*, July 2003.

29. M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology*, 12(2):191–221, April 2003.
30. C. M. F. Rubira, R. de Lemos, G. Ferreira, and F. Castor Filho. Exception handling in the development of dependable component-based systems. *Software – Practice and Experience*, 35(5):195–236, March 2005.
31. C. F. Schaefer and G. N. Bundy. Static analysis of exception handling in ada. *Software: Practice and Experience*, 23(10):1157–1174, October 1993.
32. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Addison-Wesley, 1996.
33. M. Sloman and J. Kramer. *Distributed Systems and Computer Networks*. Prentice-Hall, 1987.
34. W. Weimer and G. Necula. Finding and preventing run-time error handling mistakes. In *Proceedings of OOPSLA'2004*, pages 419–433, Vancouver, Canada, October 2004.
35. K. Yi. An abstract interpretation for estimating uncaught exceptions in standard ml programs. *Science of Computer Programming*, 31(1):147–173, 1998.