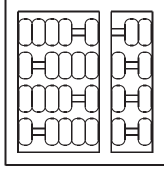

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Specification of Exception Flow in Software Architectures

Fernando Castor Filho

Patrick Henrique da S. Brito

Cecília Mary F. Rubira

Technical Report - IC-06-09 - Relatório Técnico

May - 2006 - Maio

The contents of this report are the sole responsibility of the authors.
O conteúdo do presente relatório é de única responsabilidade dos autores.

Specification of Exception Flow in Software Architectures

Fernando Castor Filho [1] Patrick Henrique da S. Brito [2]
Cecília Mary F. Rubira [3]

Institute of Computing,
State University of Campinas
P.O. Box 6176, 13084-971
Campinas, SP, Brazil

Abstract

In recent years, various approaches combining software architectures and exception handling have been proposed for increasing the dependability of software systems. This conforms with the idea supported by some authors that addressing exception handling-related issues since early phases of software development may improve the overall dependability of a system. By systematically designing the mechanism responsible for rendering a system reliable, developers increase the probability of the system being able to avoid catastrophic failures at runtime. This paper addresses the problem of describing how exceptions flow amongst architectural elements. In critical systems where rollback-based mechanisms might not be available, such as systems that interact with mechanical devices, exception handling is an important means for recovering from errors in a forward-based manner. We present a framework, named Aereal, that supports the description and analysis of exceptions that flow between architectural components. Since different architectural styles have different policies for exception flow, Aereal makes it possible to specify rules on how exceptions flow in a given style and to check for violations of these rules. We use a financial application and a control system as case studies to validate the proposed approach.

Preprint submitted to Elsevier Science *29 April 2006*

1 Introduction

The concept of software architecture [1] has been recognized in the last decade as a means to cope with the growing complexity of software systems. According to Clements and Northrop [2], 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 [3]. Software architectures are described formally using architecture description languages, or ADLs [4]. Most ADLs share the same conceptual basis whose main elements are architectural components (loci of computation or data stores), architectural connectors (loci of interaction between components), and architectural configurations (connected graphs of components and connectors that describe architectural structure) [4]. Software architectures described in an ADL can be analyzed by tools, in order to verify whether it satisfies some desired properties. In particular, automated and semi-automated analysis of software architectures are valuable tools in the construction of dependable systems.

Applications that can cause risks for human lives or risk of great financial losses are usually made fault-tolerant [5], so that they are capable of providing their intended service, even if only partially, when faults occur. Fault-tolerant systems include mechanisms for detecting errors in their states and recovering from these errors. Exception handling [6] is a well-known mechanism for structuring error recovery in software systems. Since exception handling is an application-specific technique, it complements other techniques for improving system reliability, such as atomic transactions [7], 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. An exception handling system (EHS) helps in the construction of dependable software by imposing constraints on the way exceptions and exception handlers may be used in a given language and detecting violations of these constraints. For instance, the EHS of Java detects unhandled exceptions at compile-time, unless the

Email addresses: `fernando@ic.unicamp.br` (Fernando Castor Filho),
`patrick.silva@ic.unicamp.br` (Patrick Henrique da S. Brito),
`cmrubira@ic.unicamp.br` (Cecília Mary F. Rubira).
[1] Supported by FAPESP/Brazil, grant 02/139960-2.
[2] Supported by CAPES/Brazil.
[3] Partially supported by CNPq/Brazil under grant 351592/97-0, and by FAPESP/Brazil, under grant 2004/10663-8.

developer states explicitly that this checking should not be performed (by declaring the exception *unchecked*).

Usually, a system's error detection and handling mechanisms are developed in an ad-hoc manner during system implementation [6,8,9]. However, some researchers argue that, to achieve the desired levels of reliability, mechanisms for detecting and handling errors should be developed systematically from the early phases of development [10,11], that is, from requirements, passing by analysis and design (architectural design and detailed design) phases. One approach that offers potential benefits to the development of dependable systems at the architectural level is to combine ADLs and exceptions during the design of the architecture. By extending formal architecture descriptions with information about exceptions, developers (i) better document their decisions about the flow of exceptions at the architectural level, (ii) make their assumptions about the EHS of the language(s) that will be used during system implementation explicit, and (iii) can check inconsistencies between the architecture description and the assumed EHS. Hence, in this paper, we attempt to answer the following research question:

> **Research question #1**: *How can software architectures be enriched with information about exceptions so that it is easy to verify some useful properties regarding exception flow?*

A generic solution to this question should deal with software architectures based on multiple architectural styles. An architectural style defines a vocabulary of types of design elements which are part of a family of architectures and the rules by which these elements are composed [12]. Well-known examples are Client/Server and Publisher/-Subscriber [13]. Since architectural styles dictate how components in an architecture interact, they also impact the way exceptions flow amongst architectural elements [14]. Therefore, we refine research question #1 to include the notion of architectural style:

> **Research question #2**: *How can software architectures be enriched with information about exceptions so that it is easy to verify some useful properties regarding exception flow while taking into account the constraints imposed by different architectural styles?*

We present the Aereal (Architectural Exceptions Reasoning and Analysis) framework, which supports the extension of architectural descriptions with information about exceptions. These extended descriptions can be analyzed in order to check if they satisfy the constraints imposed by different architectural styles on how exceptions flow between architectural elements. Moreover, it is possible to specify properties of interest regarding exception flow and verify automatically if they are satisfied by an architecture. As enabling technologies, Aereal uses Alloy [15], a first-order relational

language, ACME [12], an interchange language for architecture description, and their associated tool sets.

This work is organized as follows. Section 2 provides some background information on exception handling, ACME, and Alloy. Section 3 gives a motivation for specifying exceptions at the architectural level and lists some requirements that a solution to research questions #1 and #2 should satisfy. Section 4 provides an overview of Aereal approach to architecture-centric software development. Section 5 describes two case studies we have conducted to assess the usefulness of the framework. Section 6 reviews some related works. The last section presents concluding remarks and points out directions for future works.

2 Background

2.1 Exception Handling

When a system receives a service request and produces a response according to its specification, the produced response is said to be *normal*. Conversely, if the system produces a response that does not conform with its specification, this response is said to be *abnormal*, or *exceptional*. Abnormal responses usually indicate the occurrence of an error and since these responses are expected to occur only rarely, they are called *exceptions*. When exceptions occur, the system should handle them in order to return to a coherent state. The part of the behavior of a system that is responsible for handling exceptions is called abnormal, or exceptional, activity. Conversely, the part of the behavior of a system that is responsible for its functionality, as defined by its specification, is called normal activity.

Exception handling [6] is a mechanism for structuring the exceptional activity of a system 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 exceptional activity.

The concept of *idealized fault-tolerant component* (IFTC) [5] defines a conceptual framework for structuring exception handling in software systems. An IFTC is a component [4] in which the parts responsible for the normal and abnormal activities

[4] In a broader sense; an object, a subsystem, a software component, or a whole systems.

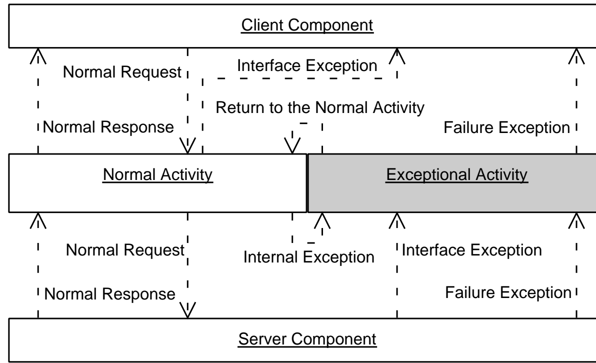

Fig. 1. Idealised Fault-Tolerant Component.

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 *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*. 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 ACME

ACME [12] works as both an interchange language for architecture description and an ADL. It provides a simple structural framework for representing architectures, together with a flexible annotation mechanism. The language does not impose any semantic interpretation of an architectural description, but simply provides a syntactic structure to which semantic descriptions can be associated. This semantic information can then the interpreted by tools.

ACME supports the definition of four distinct aspects of architecture: (1) structure,

(2) properties of interest, (3) constraints, and (4) types and styles. Structure aspect defines the organization of a system into its constituent parts, components, connectors, and attachments between these elements. The properties of interest aspect defines syntactic structures to which semantic information can be associated and analyzed by tools. Constraints are guidelines for how the architecture can change over time. The types and styles aspect defines classes and families of similar architectures.

In ACME, architectural styles are called *families*. An ACME family defines types of architectural elements that can be used in architectures that adhere to a specific architectural style and constraints on instances of these types. An ACME family [5] can extend another one by means of subtyping. Extension in ACME works as in object-oriented languages; all the elements defined by the parent ACME family are visible in and part of the child family. Architectural element types defined by a child ACME family may extend element types defined by the parent ACME family. Subtyping relations between element types work in the same way as subtyping relations between ACME families. Multiple inheritance is allowed and name conflicts result in invalid ACME families/element types.

Figure 2 presents part of the ACME definition of the Client/Server architectural style. The ClientAndServerFam family in the Figure 2 defines two types of components, ClientT and ServerT (Lines 6 and 9), and one type of connector, CSConnT (Line 13). Instances of each component type have exactly one access point, or "port" in ACME terminology (Lines 7 and 10). The ServerT component type has an integer property indicating the maximum number of concurrent requests an instance is capable of handling. In ACME, the access points of a connector (to which component ports are attached) are called "roles". Instances of CSConnT have two roles, one for the client and one for the server (Lines 14 and 15). Moreover, connector type CSConnT defines a simple constraint indicating that connectors of this type have exactly two roles (Line 16).

Figure 3 shows a partial definition of a very simple ACME system. An ACME system is an architecture description adhering to 0 or more architectural styles. Elements in an ACME system are instances of the element types defined by the styles to which it adheres. System NetBanking adheres only to the Client/Server style (Lines 2-3). It defines two components, InternetBankingServer (Lines 6-12) and Client1_WebBrowser (14-20), of types ServerT and ClientT, respectively, and one connector, conn (Line 13), of type CSConnT. Architectural structure is specified by defining attachments between component ports and connector roles. The receivedRequest port of InternetBankingServer is attached to the serverSide role of conn and the sendRequest port of

[5] In the rest of the paper, we use the terms "ACME family" and "architectural style" interchangeably.

```
1 Family ClientAndServerFam = {
2    Port Type ClientPortT = { ... }
3    Port Type ServerPortT = { ... }
4    Role Type clientSideRoleT = { ... }
5    Role Type serverSideRoleT = { ... }
6    Component Type ClientT = {
7        Port sendRequest : ClientPortT =  new ClientPortT;
8    }
9    Component Type ServerT = { ... }
10       Port receiveRequest : ServerPortT =  new ServerPortT;
11       Property max-concurrent-requests : int;
12   }
13   Connector Type CSConnT = {
14       Role clientSide : clientSideRoleT =  new clientSideRoleT;
15       Role serverSide : serverSideRoleT =  new serverSideRoleT;
16       invariant size(self.roles) == 2;
17   }
18 }
```

Fig. 2. ACME definition of the Client/Server style.

Client1_WebBrowser is attached to the clientSide role of conn (Lines 4 and 5). Elements in a system can define instance-specific properties. In the example, both InternetBankingServer and Client1_WebBrowser define instance-specific properties regarding their positions in the screen (Lines 8-9, 16-17).

```
1 import families\ClientAndServerFam.acme;
2 System Netbanking : ClientAndServerFam =
3     new ClientAndServerFam extended with {
4   Attachment Client1_WebBrowser.sendRequest to conn.clientSide;
5   Attachment InternetBankingServer.receiveRequest to conn.serverSide;
6   Component InternetBankingServer : ServerT =  new ServerT extended with {
7     Port receiveRequest : ServerPortT =  new ServerPortT extended with {
8       Property vis-x : float = 20.0;
9       Property vis-y : float = -25.0;
10    };
11    ...
12  };
13  Connector conn : CSConnT =  new CSConnT;
14  Component Client1_WebBrowser : ClientT =  new ClientT extended with {
15    Port sendRequest : ClientPortT =  new ClientPortT extended with {
16      Property vis-y : float = 65.0;
17      Property vis-x : float = 176.0;
18    };
19    ...
20  };
21  ...
22 };
```

Fig. 3. A trivial ACME system.

Architectural modeling in ACME is supported by AcmeStudio [16]. AcmeStudio is an architecture development environment that allows the definition of new architectural styles and the modeling of systems which instantiate these styles using an intuitive graphical user interface. The environment includes a constraint solver called

Armani [17] that checks whether an architecture satisfies the constraints defined by the styles to which it adheres.

Aereal assumes that architecture descriptions are written in ACME. Although other ADLs could be employed, some reasons have made us choose ACME: (i) it focuses on the structure of the system; (ii) it has powerful constructs for defining new architectural styles; (iii) it is extensible by the use of properties; and (iv) it has mature tool support. Since ACME is both an ADL and an interchange language, developers may employ other ADLs for modeling specific aspects of the system and then translate these specifications to ACME [18], so that they can be used with Aereal.

2.3 Alloy

Alloy [15] is a lightweight modeling language for software design. It is amenable to a fully automatic analysis, using the Alloy Analyzer (AA) [19], and provides a visualizer for making sense of solutions and counterexamples it finds. Similarly to other specification languages, such as Z and B [20], 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.

Alloy is used by Aereal for analyzing exception flow mainly because (i) it focuses on how data is specified, an important feature for modeling exceptions as data objects [22]; (ii) it supports automated analysis; (iii) it has a mature constraint solver; and (iv) it is simpler and arguably easier to use than similar languages, such as B.

3 Specification of Exceptions at the Architectural Level

Usually, a large part of a system's code is devoted to error detection and handling [6,8,23]. However, since developers tend to focus on the normal activity of applications and only deal with the code responsible for error detection and handling at the implementation phase, this part of the code is usually the least understood, tested, and documented [6,8]. In order to achieve the desired levels of reliability, mechanisms for detecting and handling errors should be developed systematically throughout all

the phases of software development [11,10]. As pointed out by many authors [3,24,25], the architecture of a software system has a strong impact on its ability to meet its intended quality requirements, for example, security, reliability, availability, and performance. Thus, if a system should be reliable and exception handling is one of the mechanisms that will be used to achieve this goal, it may be beneficial to consider exception handling-related issues during architectural design.

At the architectural level, an exception is a signal (message, event, language-level exception, etc.) employed by an architectural element to indicate to other elements that it has failed. An element that potentially receives an exception should be capable of doing something about it. During architectural design, it may still be too early to know exactly what. However, the design of an architecture involves assigning responsibility to architectural elements [2]. Therefore, it should be known at least which elements raise which exceptions, which elements are responsible for handling a certain exception, and which elements, upon receipt of an exception, just propagate it. This small amount of information is enough to reason about relevant properties regarding exception handling, for example, whether an exception raised by an element is actually handled by some other element in the architecture.

Architects often need to understand the architecture of a system from various perspectives, according to specific quality attributes. *Architectural views* represent different aspects of the same architecture and each view shows how the architecture achieves a particular quality attribute [3]. These quality attributes are usually characterized in terms of properties of interest that the architecture should satisfy. A view for analyzing properties of interest about exception flow should include information such as: (i) the exceptions that each architectural element raises, handles, and propagates; and (ii) how exceptions flow between these elements. The following subsection presents an example of a view that describes the flow of exceptions in the architecture of an air traffic control (ATC) system. This example was extracted from a popular textbook on software architectures [3]. We highlight some of the limitations of the informal approach employed to specify this view. Section 3.2 lists some requirements we believe that any approach for modeling exceptions at the architectural level should satisfy.

3.1 A Motivating Example

An Air Traffic Control (ATC) system is a large-scale complex distributed system with very strict availability and performance requirements, meaning that the system should function 24/7 and timing deadlines must be met absolutely. The ATC system is an en route system which controls aircrafts from soon after takeoff until shortly before landing. Its end users are the air traffic controllers. It has a layered software

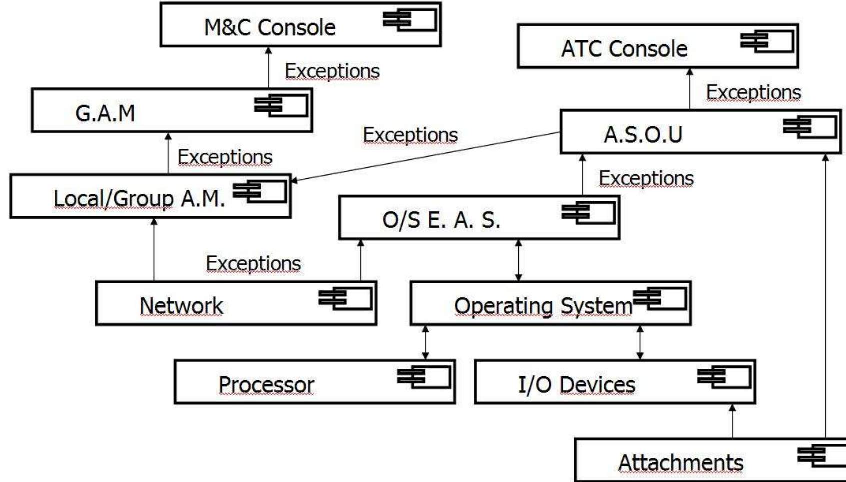

Fig. 4. Fault tolerance architectural view of an ATC system.

architecture with one subsystem cooperating with others for services.

Due to its high availability requirements, a fault tolerance view of the software architecture was defined (Figure 4) [26]. This structure describes how the faults are detected and isolated, and how the system recovers. So, besides presenting ways of detecting and isolating faults which belongs to specific components, the fault-tolerant hierarchy is designed do trap and recover from errors which are a consequence of cross-application interactions.

In Figure 4, the names of the components are abbreviated. Arrows indicate flow of exceptions between components (layers) of the architecture and a layer that receives exceptions from another layer is responsible for handling those exceptions. The idea of the authors is to express a hierarchical depiction of the system where layers that are closer to the top implement more general exception handling strategies. Each one of the eleven components of the architecture presented in Figure 4 has a specific role, as shown in Table 1.

The architecture of Figure 4 conveys useful information about exceptions in the system. For example, it is clear that exceptions flow from the Network component to the Local/Group A.M. and O/S E.A.S. components. It is also clear, due to the topology of the system, that the M&C Console and ATC Console layers have the most general exception handlers. This is expected, since, in an ATC system, users should always be notified of unhandled exceptions [3]. However, in spite of these useful pieces of information, the fault tolerance view of Figure 4 has some important shortcomings. First, this view does not specify what are the exceptions that each layer raises and handles. Second, it is not possible to infer whether there are relevant cause-effect relations between the exceptions that layers receive and signal. For example, there is

Table 1
Components in the fault-tolerance view of the ATC system.

Component	Description
Monitor & Control Console (M&C Console.)	Gives an overview of the state of the system. It has special software to support monitoring and controlling functions and provides the top-level availability management functions.
ATC Console	A user console. It is also used by the controllers and is the access point to services which do not require a strict control of availability.
Global Availability Monitor (G.A.M.)	Manages the availability of functions within the suite. It determines which of the multiple redundant copies of an application program within a sector suite is the primary copy and should thus receive messages
Local/Group Availability Manager (A.M.)	A set of application-level system services. It is responsible for managing the initiation, termination, and availability of the application programs.
Application Software Operational Unit (A.S.O.U.)	Represents the application-level system services
Operating System Extensions Address Space (O/SE.A.S.)	Provides additional system services that are necessary to support a fault-tolerant distributed system. UNIX does not provide all these services.
Network	Provides communication to the redundant services in a transparent way. There are at least two networks: a local communication network (LCN), and a backup communication network (BCN). The latter is used in some LCN failure conditions.
Operating System.	A UNIX operating system.
Processor	The hardware that processes software instructions. Due to a design choice, redundant processors must exist.
I/O Devices	The device tasks are optional and are only present for applications that directly communicate with a device driver.
Attachments	The data, which either is necessary to process the application functions, or is stored as a result of a service.

no way of knowing whether the exceptions signaled by Local/Group A.M to G.A.M are a consequence of the exceptions received by the former from Network. Third, the view expresses a strictly hierarchical structure for exception flow. It might not be appropriate to describe in the same manner the flow of exceptions in software architectures where components are peers. Fourth, it is not clear what a double-headed arrow means. These shortcomings could be alleviated by additional documentation complementing the diagram, but we did not find mention to such a documentation [26].

3.2 Requirements

To avoid the shortcomings highlighted in Section 3.1, we believe an effective approach for modeling exceptions at the architectural level should satisfy thw following basic requirements:

Req.1: It should make it possible to unambiguously distinguish the exceptions that architectural components and connectors signal and catch, and how these exceptions

flow between different architectural elements. Moreover, it should also be possible to specify the exceptions that an architectural element handles, the exceptions it raises, and the ones it propagates.

Req.2: To enhance maintainability, the specification of exceptions at the architectural level should be orthogonal and traceable to the "normal" architecture description.

Req.3: It should be supported by a pictorial (boxes-and-lines) representation, in order to be understandable by non-specialists and easier to use.

Req.4: It should take into account the notion of *architectural styles*. The approach should support the extension of traditional styles, such as layered (as in Section 3.1) and publisher/subscriber, with information about exceptions, in order to take into account the particularities of each style.

Req.5: It should support automated analysis. In this manner, it is possible to verify in a cost-effective way if the architecture presents some desired properties before the system is actually implemented.

Req.6: It should make it easy to verify if an architecture description adheres to rules defined by EHS of real-world programming languages, such as Java, Ada, C++, and C#.

4 The Aereal Framework

Architecture-based development with Aereal starts with the traditional activities of a software development process, namely, requirements analysis and architectural design of the system. It is also necessary to define 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 [10,27]. The most important results of these activities are a description of the system's architecture and informal specifications of the system's fault model and exceptional activity.

The use of Aereal in architecture-centric development involves the following activities:

(1) Defining how exceptions flow between architectural elements for each architectural style used in the architecture description. This information is specified by defining new architectural styles, called *exceptional styles*, that complement the traditional (i.e. Client/Server, Layered) ones. Hereafter we call the latter *normal styles*.

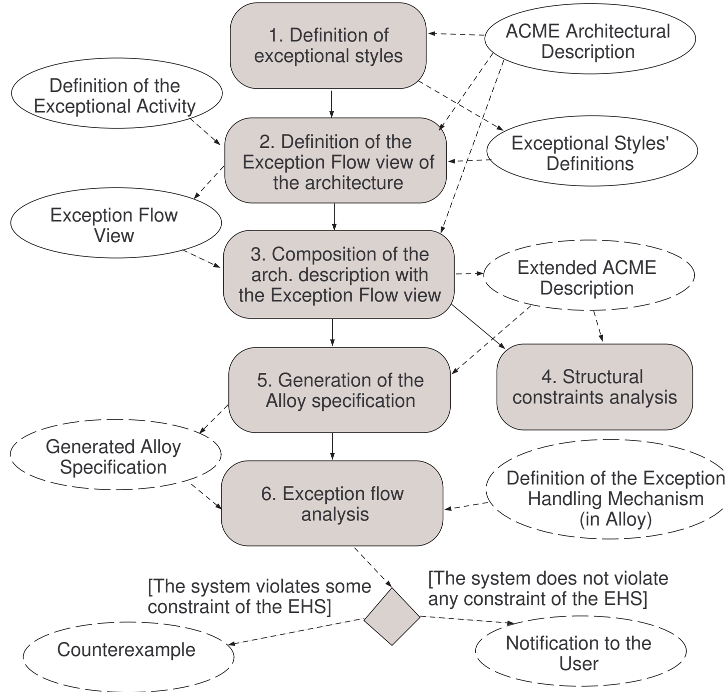

Fig. 5. Overview of the Aereal framework.

(2) Specifying the Exception Flow View of the software architecture. This view depicts the components that catch or signal exceptions (called *exceptional components*), as well as special connectors through which exceptions flow between these components (*exception ducts*). An exception flow view adheres to one or more exceptional styles.
(3) Composing the architecture description with the exception flow view, producing an architecture description extended with information about exceptions (or simply *extended architecture description*).
(4) Analyzing structural constraints, that is, checking whether the architecture description violates any of the constraints defined by the exceptional styles to which it adheres.
(5) Generating an Alloy specification from the extended architecture description.
(6) Analyzing exception flow based on the generated Alloy specification.

Figure 5 illustrates the main components of Aereal. In the figure, ovals represent artifacts and rectangles with rounded corners represent activities. Some of the ovals are dashed to indicate that they are either part of the infrastructure of the framework or generated automatically. An Aereal specification consists of a set of ACME system and family descriptions. One ACME system specifies the components and connectors view of the system [3]. The remaining ACME systems specify the exception flow views of the architecture. The ACME families specify architectural styles, both normal and

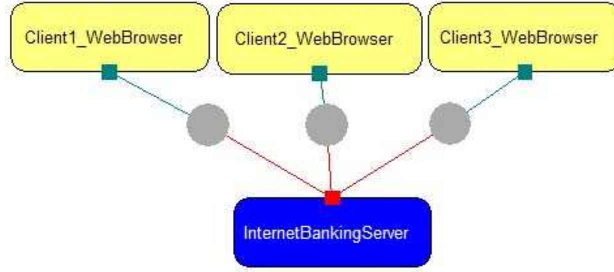

Fig. 6. Architecture of a simple Internet Banking system.

exceptional, to which the ACME systems adhere. The definition of the exceptional activity and the ACME architectural description in Figure 5 are assumed to exist and be produced as part of the software development process being used.

Figure 6 shows a high level components and connectors view of the architecture of an Internet Banking (IB) system. In the figure, components and connectors are represented by rectangles and circles, respectively. The architecture of the sytem adheres to the Client/Server architectural style. In the rest of this section, we use the IB system to make a more detailed presentation of Aereal.

The following subsections provide a detailed description of the workflow supported by Aereal.

4.1 Defining exceptional styles

Aereal uses 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. Exception ducts may be refined when new exceptional styles are defined. In this manner, the specificity of each architectural style may be taken into account. The idea that simple point-to-point connections are suitable general-purpose abstractions for modeling style-independent communication between architectural components was proposed by Mehta and Medvidovic [28]. We have tailored this idea to our specific needs. This structural perspective on exception flow was adopted because: (i) it is intuitive to architects, who are used to thinking in terms of components and connectors; (ii) it is compatible with well-established views on what exception flow is [6]; and (iii) it is easy to integrate with the concept of architectural style. It abstracts away issues such as flow of control and flow of data.

Aereal includes a basic architectural style (an ACME family) called SingleExcFam on which all exceptional styles are based. SingleExceptionFam defines types that designate

architectural elements that catch and/or signal exceptions. A new exceptional style must extend (be subtype of) two styles: SingleExcFam and the normal style to which it corresponds. The latter must be extended in order to make it possible to specify invariants for the new exceptional style that also involve elements of its corresponding normal style. It is possible to create many different exceptional styles for a single normal style and use some or all of them in an architecture description.

Style-specific features are introduced in a new exceptional style by creating subtypes of the element types that SingleExcFam defines and adding new internal elements (e.g. new ports in a component type), properties, and invariants. ExceptionalComponent is the supertype of all architectural components that deal with exceptions. By default, instances of ExceptionalComponent have no ports. New exceptional styles have to define subtypes of ExceptionalComponent that have ports, in order to specify whether instances of the type signal or catch exceptions, or both. SingleExcFam defines two port types, CatcherPortT and SignalerPortT, indicating that a component catches and signals exceptions, respectively. The supertype of all exception ducts is (for historical reasons) ExceptionalConnector. Since exception ducts are point-to-point channels, instances of this type have exactly two roles, of types CatcherRoleT and SignalerRoleT.

Example

Figure 7 presents a partial definition of the Exceptional Client/Server architectural style (ExceptionalClientAndServerFam ACME family). ExceptionalClientAndServerFam extends SingleExceptionFam and ClientAndServerFam (Lines 3 and 4). It defines two types of exceptional components, ExceptionalClientT and ExceptionalServerT (Lines 5 and 9), corresponding to clients and servers, respectively, and one type of exception duct, ExceptionalCSConnT (Line 13). Instances of ExceptionalClientT have a single port, of type CatcherPortT (Line 6). Instances of ExceptionalServerT also have a single port, of type SignalerPortT (Line 9). The invariants in lines 7 and 11 guarantee that ExceptionalClientT and ExceptionalServerT have no other ports. The invariant in lines 14-22 describes a general pattern of exception flow that can be specialized by each exceptional style. It states that if there is an exception duct linking two exceptional components in an exception flow view, then these components must exist as normal components in the normal architectural description and there must be a normal connector (a connector of type CSConnT, as shown in Line 20) linking them.

4.2 Specifying the exception flow view

An exception flow view comprises the components from the architecture description that signal or catch exceptions and exception ducts connecting these components. It

```
1 import families\SingleExceptionFam.acme;
2 import families\ClientAndServerFam.acme;
3 Family ExceptionalClientAndServerFam extends SingleExcFam,
4     ClientAndServerFam with {
5   Component Type ExceptionalClientT extends ExceptionalComponent with {
6     Port catchesPort : CatcherPortT =  new CatcherPortT;
7     invariant size(self.ports) == 1;
8   }
9   Component Type ExceptionalServerT extends ExceptionalComponent with {
10    Port signalsPort : SignalerPortT =  new SignalerPortT;
11    invariant size(self.ports) == 1;
12  }
13  Connector Type ExceptionalCSConnT extends ExceptionalConnector with {}
14  invariant Forall c1 : ExceptionalClientT in self.components |
15    Forall c2 : ExceptionalServerT in self.components |
16      Forall conn : ExceptionalCSConnT in self.connectors |
17        ((attached(conn, c1) AND attached(conn, c2) AND
18          connected(c1, c2)) -> (satisfiesType(c1, ClientT) AND
19            satisfiesType(c2, ServerT) AND
20              (Exists normalConn : CSConnT in self.connectors |
21                attached(normalConn, c1) AND attached(normalConn, c2)) ))
22                  <<label : string = "OK.";errMsg : string = "Problems.";>>;
23 }
```

Fig. 7. Partial definition of the Exceptional Client/Server style.

adheres to one or more exceptional styles, depending on the normal styles the architecture description uses. Each element in an exception flow view is annotated with information about exceptions. All the elements in an exception flow view are instances of types that the exceptional styles to which it adheres define. Aereal represents exception flow views as ACME systems.

The element types that SingleExcFam defines declare properties that are used to associate information about exceptions to architectural elements. In the exception flow view, instances of these types assign values to these properties. This information is taken into account during exception flow analysis (Section 4.4). Table 2 lists the properties declared by the element types of SingleExcFam and informally describes their semantics. All the properties in the table, except for the last one, are declared by ExceptionalComponent and ExceptionalConnector. Element types CatcherPortT and CatcherRoleT only declare properties handles, catches, and propagates. Element types SignalerPortT and SignalerRoleT only declare properties raises and signals. Only instances of SignalerPortT use property catcherPorts.

In the exception flow view, values assigned to properties declared by exceptional components and exception ducts add up to the values assigned to the homonym properties declared by associated ports and roles, respectively. For example, if an exceptional component C assigns value $\{RemoteException\}$ to property raises and signaler port P of C assigns value $\{IOException\}$ to its homonym property, it is assumed that the value of property $raises$ for P is $\{RemoteException, IOException\}$.

Table 2
Properties employed by the exception flow view to associate information about exceptions to architectural elements.

Property	Description
signals	The set of exceptions signaled by an architectural element. Aereal automatically computes signals from the values assigned to raises, handles, and propagates for all the elements in an exception flow view. Optionally, it is possible to manually specify a set of exceptions. In this case, Aereal verifies if the element actually signals the exceptions during exception flow analysis.
catches	The set of exceptions caught by an architectural element. Similarly to signals, computed automatically by Aereal.
handles	The set of exceptions that an architectural element handles. In this case, "handles" means that the handler for the exception does not end its execution by raising an exception. After handling, normal execution is resumed.
propagates	This property is a variation of handles where the handler ends its execution by raising an exception. The propagates property specifies a cause-consequence relationship between an exception that an element catches and an exception that it signals.
raises	The set of exceptions that an element raises. Since only external exceptions are taken into account by Aereal, raises is a subset of signals
catcherPorts	The set of ports of type CatcherPortT associated to a port of type SignalerPortT. Exceptions caught by these catcher ports and not handled or propagated by the element are signaled through the associated signaler port. If catcherPorts is left unspecified for a given signaler port, Aereal assumes that this port is associated to all the catcher ports of the element.

This semantics makes it possible to specify that different ports of the same component manipulate different sets of exceptions while retaining the ability to specify a common denominator. A similar rationale applies to exception ducts and roles.

Usually, unlike exceptional components, exception ducts are connectors that do not exist in the normal architecture description. They are orthogonal to "regular" connectors, in the sense that they do not necessarily follow the same rules for message/data passing and transfer of control. For instance, an exception duct between components

that adhere to the Publisher/Subscriber style may or may not indicate transfer of control, depending on the semantics of the application. In our approach for exception flow analysis, this is not of utmost importance because we are not modeling system behavior. Moreover, even though a set of components in a Publisher/Subscriber architecture may communicate through a single normal connector, there may be several exception ducts between these components, depending on how the components may fail and which is responsible for handling which exceptions.

Example

The IB system's exception flow view only uses the Exceptional Client/Server style, defined by ExceptionalClientAndServerFam, since only the Client/Server style is used in the architecture description. It is important to notice, however, that various architectural styles could have been used.

Figure 8 shows part of the ACME definition of the IB system's exception flow view. This ACME system adheres to the Exceptional Client/Server architectural style (Lines 2-3). Line 9 specifies that the InternetBankingServer component signals an exception called RequestNotProcessedException. This exception is raised by the component itself (Line 8). Lines 14 and 15 state that exeception duct ExceptionalCSConnT0 catches exceptions of type RequestNotProcessedException and signals exceptions of type RemoteException, respectively. Furthermore, Lines 16-17 specify that ExceptionalCSConnT signals RemoteException as a consequence of catching RequestNotProcessedException.

```
1 import families\ExceptionalClientAndServerFam.acme;
2 System ExceptionalNetbanking:ExceptionalClientAndServerFam=
3       new ExceptionalClientAndServerFam extended with {
4
5   Component InternetBankingServer : ExceptionalServerT =
6           new ExceptionalServerT extended with {
7     Port signalsPort : SignalerPortT =  new SignalerPortT extended with {
8       Property raises : Set{} = {RequestNotProcessedException};
9       Property signals : Set{} = {RequestNotProcessedException};
10    };
11  };
12  Connector ExceptionalCSConnT0 : ExceptionalCSConnT =
13      new ExceptionalCSConnT extended with {
14    Property catches : Set{} = {RequestNotProcessedException};
15    Property signals : Set{} = {RemoteException};
16    Property propagates : Sequence<> =
17      < RequestNotProcessedException, RemoteException >;
18  };
19  ...
20 };
```

Fig. 8. Exception flow view of the IB system.

4.3 Composing architecture description and exception flow views

Structural constraints are evaluated after an extended architecture description is generated. This description is produced by the *Composer* tool, which is provided by Aereal. This tool reads the architecture description and the exception flow view and updates the former with exception-related information defined by the latter. This organization promotes separation of concerns at the architectural level, since the architecture description does not refer to the exception flow view, and the two exist as separate design artifacts and are composed automatically.

Figure 9 presents pseudo-code describing part of the functioning of the Composer. In the pseudo-code, variables representing architectural elements have fields through which it is possible to access their internal elements. For example, variable `excFlowView` represents the whole exception flow view and has the fields `families` and `components`, among others. The first is the set of all exceptional styles to which the exception flow view adheres and the second is the set of components in it. The Composer works by copying elements (components, connectors/exception ducts, ports in a component, attachments, etc.) from the exception flow view to the architecture description. Each element is identified by its name. The scope of an element is the architectural element of which it is part, for example, the scope of a component is a whole system, the scope of a port is a component, etc.

When the Composer attempts to copy an exceptional component for which there is a homonym component in the architecture description, the tool copies the properties and ports that store exception-related information (Table 2) to the component in the architecture description. As for exception ducts, to avoid conflicts between homonym ducts that appear in different exception flow views, the names of these elements are altered during composition and attachments involving them are updated to reflect these changes. It is not possible to combine existing exception ducts in the same way as exceptional components because this could potentially create exception ducts linking more than two components. Since the composition process has, by default, a "union" semantics, instead of "overwrite", many different exception flow views can be composed with the same architecture description simultaneously. This supports a step-by-step development process where various exception flow views are specified for different parts of the architecture description. In later phases of system development, these views can be combined and analyzed in conjunction.

Structural constraints analysis employs the Armani constraint solver [17] or AcmeStudio (which includes Armani) to check if the extended architecture description satisfies the invariants defined by the exceptional styles to which it adheres. Violations of these invariants result in error messages.

```
1 Composer(ACMESystem excFlowView, ACMESystem archDescription) {
2   // families stores the set of families to which a system adheres
3   for each F in excFlowView.families
4     if (F not in archDescription.families) add F to archDescription.families
5   for each EC:ExceptionalComponent in excFlowView.components {
6     if (archDescription.components contains an element NC:Component
7         such that NC.name = EC.name) {
8       // types stores the set of types of which an element is an instance
9       for each T in EC.types
10        if (T not in NC.types) add T to NC.types
11      for each P:Set in EC.properties such that SingleExcFam defines P
12        // NC also has P because it is an instance of all the
13        // types of which EC is an instance
14        NC.P = NC.P + EC.P // set union and assignment
15      for each port PSE:SignalerPortT in EC.ports
16        if (there is no PSN in NC.ports such that PSN.name = PSE.name)
17          add PSE to NC.ports
18        else if (SignalerPortT in PSN.types)
19          for each P:Set in PSE.properties such that SingleExcFam defines P
20            PSN.P = PSN.P + PSE.P // set union and assignment
21        else report an error
22      ... // The same for ports of type CatcherPortT
23    }
24    else add EC to archDescription.components
25  }
26  ... // copy exception ducts/roles
27  for each A in excFlowView.attachments
28    if not (A in archDescription.attachments)
29      add A to archDescription.attachments
30    else report an error
31 }
```

Fig. 9. Algorithm executed by the Composer tool.

Example

Figure 10 presents the components and connectors view of the extended architecture description of the IB system, as produced by the Composer. Notice that between the server and each client there are two connectors, a normal Client/Server connector and an exception duct.

4.4 Analyzing exception flow

To analyze how exceptions flow in the architecture, it is necessary to generate an Alloy specification corresponding to the extended ACME description. Since generating the Alloy specification manually is a difficult and error-prone activity, Aereal provides a tool, called Converter, that automatically translates an extended ACME description to Alloy. The Converter works according to the following steps:

(1) Read the ACME descriptions corresponding to the extended architecture description and the architectural styles (both normal and exceptional).

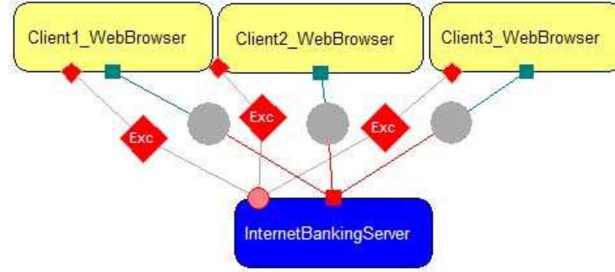

Fig. 10. Extended architecture description of the Internet Banking system.

(2) Build a language-independent representation of the architecture. This intermediate representation is based on a system model described elsewhere [29]. It includes architectural elements that are instances of types defined by exceptional styles and exception-related information associated to these elements through relations that correspond to the properties listed of Table 2. During the construction of the intermediate representation, the Converter discards information about architectural styles because this information is not used during exception flow analysis.
(3) Compute signaled and caught exceptions. As pointed out in Section 4.2, the sets of exceptions that each architectural element signals and catches are automatically computed by Aereal. Manually-specified sets of signaled and/or caught exceptions are preserved by this process.
(4) Write the text file comprising the Alloy specification of the system.

Exception flow analysis consists in verifying if the generated Alloy specification 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 exception handling mechanism assumed by Aereal, which is based on C++', and how software architectures are structured. Examples of basic properties are presented below, stated informally.

BP1. Architectural elements signal all the exceptions they raise.
BP2. All exception ducts catch exceptions from and signal exceptions to exactly one exceptional component.
BP3. The graph formed by using exceptional components as vertexes and exception ducts as edges is connected.
BP4. Architectural elements signal all the exceptions they catch and do not handle.

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

DP1. Architectural elements do not have handlers for 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 (explicit exception propagation [22]).
DP3. No unhandled exceptions.

Application-specific properties are rules regarding the flow of exceptions in a specific application. The Alloy definition of the exception handling mechanism used by Aereal (Figure 5) 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.

By default, all exceptions are subtypes of RootException. It is possible to specify in Alloy a type hierarchy for the exceptions. For example, to mimic the EHS of Java, at least four exception types would be necessary: (i) Throwable, subtype of RootException; (ii) Exception, subtype of Throwable; (iii) Error, subtype of Throwable; and (iv) RuntimeException, subtype of Exception. Application-specific exceptions would inherit from these exception types. This type hierarchy is then taken into account when exception flow is analyzed. If no exception type hierarchy and no checks beyond the default ones are necessary, no knowledge of Alloy is required to use the framework.

Example

Figure 11 defines two Alloy predicates named `bp1` and `dp1`, formally specifying properties BP1 and DP1, 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 that associate exception information to the elements of the system. For instance, the `signals` relation specifies what are the exceptions that a component signals and the exception ducts that catch them. The "." operator represents relational join. For example, given `raises` ∈ `Component`↔`Duct`↔`RootException` and `C` ∈ `Component`, where `Component`, `Duct`, and `RootException` are sets, the formula `C.raises = ES` constrains the image of `C` under the relation `raises` to be `ES` ∈ `Duct`↔`RootException`. Predicate `bp1()` 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. A more detailed description of the system model defined by Aereal is available elsewhere [29].

```
1 /* Basic property BP1 */
2 pred bp1() {
3   all C : Component | (C.raises in C.signals)
4 }
5 /* Desired property DP1 */
6 pred dp1() {
7   all C : Component | all D : Duct | D in C.catchesFrom &&
8   D.(C.handles) in D.(C.catches) &&
9     (D.(C.catches))<:(D.(C.propagates))=D.(C.propagates)
10 }
```

Fig. 11. Properties BP1 and DP1 specified in Alloy.

Figure 12 shows a counterexample generated by the AA when we analyzed exception flow in the Alloy specification of the IB system. This counterexample indicates that the generated Alloy specification violates basic property BP1. The error has been detected because we have modified the exception flow view of the IB system (Figure 8) so that InternetBankingServer signals RequestNotProcessedException but it does not raise it.

5 Evaluation

To assess whether Aereal meets the requirements of Section 3.2, we undertook two case studies to answer the following experimental questions:

EQ1. Does the abstraction for exception flow employed by Aereal, exception ducts, support the specification of exception flow views based on different architectural styles?
EQ2. Can Aereal help developers in gaining a deeper understanding of a system's exceptional activity?
EQ3. How useful is Aereal in the task of finding design errors and inconsistencies in the exceptional activity of a system?
EQ4. Does the approach employed by Aereal scale up well for systems with a large number of components, ducts, and/or exceptions?

The targets of the case studies were (i) a textbook example mining control system [30] (Section 5.1) and (ii) a financial system that registers and controls checkbooks, account contracts, and credit limits (Section 5.2). The latter was developed in an industrial setting and is part of a real application deployed in several companies.

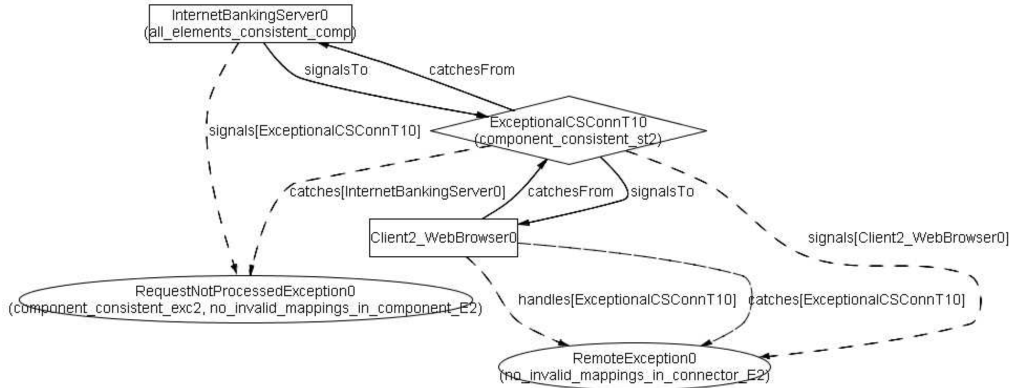

Fig. 12. A counterexample generated by AA.

5.1 Mining Control System Case Study

The case study is based on a simplified version of the control system for the mining environment [30]. 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. A schematic representation of the mining system is given in Figure 13. The mining control 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. When the water reaches a high level, the pump is turned on and the sump is drained until the water reaches a low level. A water flow sensor is able to detect the flow of water in the pipe. However, the pump is situated underground, and for safety reasons it must not be started, or continue to run, when the amount of methane in the atmosphere exceeds a safety limit. For controlling the level of methane, there is an air extractor control station that 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 also controlled from the surface via an operator console that should handle any emergencies raised by the automatic system.

Figure 14 shows the components and connectors view of the architecture of the mining system [10]. This representation is related to the logical architecture of the system, that is, it describes the conceptual organization of the design elements into groups, independently of their physical packaging. In this architecture, collaborations between components are performed from higher to lower layers. Arrows between the software components indicate data flow between them. Components from upper layers make service requests to components from lower layers and the latter produce responses that can be normal or exceptional. This architecture view uses only the layered ar-

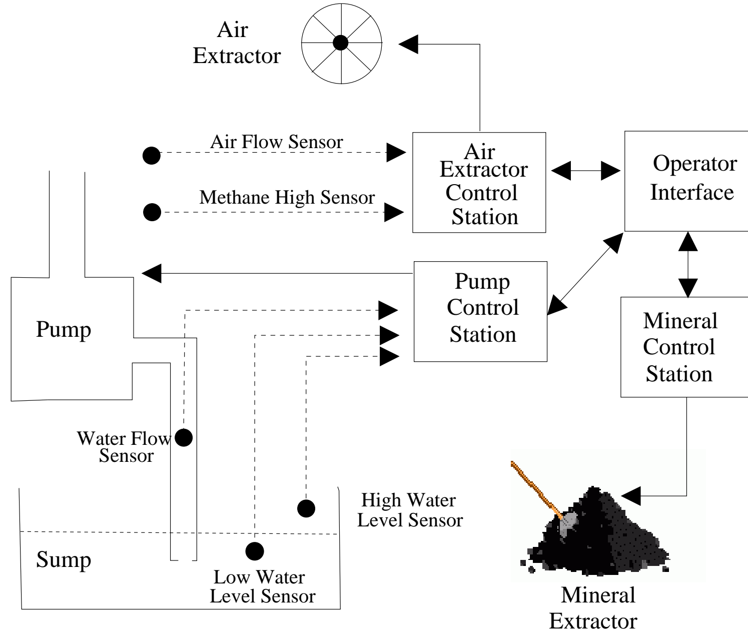

Fig. 13. Schematic diagram of the mining system.

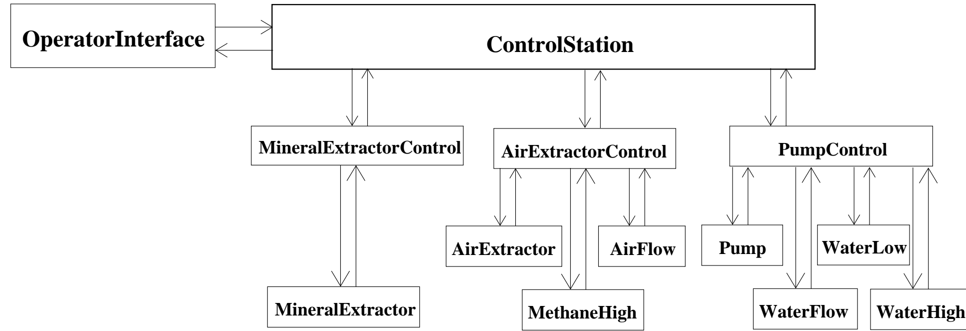

Fig. 14. Architecture design of the mining control system.

chitectural style [1].

The system can fail in several ways. All the sensors (MethaneHigh, AirFlow, WaterHigh, WaterLow, and WaterFlow) and actuators (AirExtractor, Pump, and MineralExtrator) may fail. Errors in these components are detected by the corresponding control stations. Sensors can fail by stopping to send information to their corresponding control stations. However, when they do send information, the latter is assumed to be correct. Actuators can fail by not working when switched on and not stopping to work when switched off. Whenever an error is detected, the detecting control station signals an exception to the ControlStation component, which attempts to interrupt execution and activate an alarm. We assume that communication between components is reliable. Table 3 shows the different types of exceptions that can be raised by AirExtractorCon-

Table 3
Exceptions related to the AirExtractorControl component.

Exception	Description
AirExtractorOffException	The level of methane is high, the air extractor is on, but no air flow is detected.
AirExtractorOnException	The level of methane is normal, the air extractor is off, but the sensor detects air flow.
SensorFailureException	A timeout occurs while the AirExtractorControl component is waiting for information from the sensors.

trol. Exceptions raised by the other control stations follow the same pattern, except for MineralExtractorControl, which does not signal sensor-related exceptions. A detailed description of the exceptional activity of the mining system is available elsewhere [10].

This case study was conducted in two phases. Phase 1 (Section 5.1.1) addresses experimental questions EQ2 and EQ3. Phase 3 (Section 5.1.2) specifically targets experimental question EQ1. Both phases were conducted by one of the authors.

5.1.1 Applying Aereal - Phase 1

We started by applying the workflow presented in Figure 5 to the mining control system. The first activity consisted in describing the architecture of Figure 14 in ACME, using the definition provided by AcmeStudio for the layered architectural style. We then specified an exceptional layered style, represented by the ExceptionalLayeredFam ACME family. The latter is similar to the ExceptionalClientAndServerFam family (Section 4.1), but simpler. This simplicity is mainly due to two factors. First, exceptional components in layered architectures do not have ports by default, since any component is capable of (not)signaling/catching exceptions. This differs from Client/Server architectures, where responsibility for signaling and catching exceptions is unambiguously assigned to each type of component. Second, layered architectures have only one type of component, namely, the layer. A strictly layered style [13], where only adjacent layers communicate, would impose additional constraints, but we have not taken these constraints into account.

The following activity consisted in specifying the exception flow view of the architecture. Initially, this view comprised only the AirExtractorControl, PumpControl, MineralExtractorControl, and ControlStation components. The former three are responsible for detecting errors and signaling exceptions to the latter, which handles them. According to the specification of the system's exceptional activity [10], the expected behavior of ControlStation when it receives exception AirExtractorOnException is to

"activate an external alarm, in order to notify the system operator about the existence of a safety threat, and shut down the system". Since the alarm is external to the system and its activation does not involve any exceptions being signaled by it or by ControlStation, we have not included it in the exception flow view. This approach faithfully reflects the specification of the system's exceptional activity. The problem with it, though, is that the operator of the system does not receive any information about the safety threat when the alarm rings. This seems unrealistic since the reason to ring the alarm is to notify the operator that it might be necessary to manually intervene, in order to avoid catastrophic damage. If the operator does not know anything about the safety threat that triggered the alarm, it is not possible to respond promptly.

In order to avoid the aforementioned problem, the ControlStation component should inform the operator about the nature of the threat. The natural way to do this is through the OperatorInterface component. However, the original specification of the mining system states that the two components only interact to start/stop mineral extraction. We interpret this is an incompleteness in the specification. To solve the problem, we modified the handling strategy of ControlStation. Besides attempting to shut down the system and activating the alarm, the handler now signals an exception, EmergencyException, to OperatorInterface. This new exception encapsulates the one that was caught, which includes information about the safety threat, and is handled by the OperatorInterface component. Two reasons motivated the choice of representing this interaction as an exception, instead of an additional service request: (i) the ControlStation component is notifying the OperatorInterface that an error occurred and exceptions are the natural means to do this; and (ii) ControlStation is located in a lower layer and service requests only flow from upper layers to lower layers, not the other way around. Figure 15 show part of the final specification of the ControlStation component.

After finishing the specification of the exception flow view, we used the Composer and Converter tools to produce the Alloy specification corresponding to the architecture of the mining system. We then used the AA to analyze exception flow and the latter reported that the specification satisfied all the basic and desired properties.

5.1.2 Applying Aereal - Phase 2

The second phase of the case study consisted in describing the mining control system using different modeling approaches. More specifically, our intent was to use different architectural styles, in order to provide at least an initial answer to experimental question EQ1. We produced two alternative designs for the mining system. The first one adheres to the C2 [31] architectural style and is an adaptation of another work [32].

```
1  Component ControlStation : layerT, ExceptionalLayerT =
2      new layerT, ExceptionalLayerT extended with {
3    ... // other stuff
4    Port CatcherPortT0 : CatcherPortT =  new CatcherPortT extended with {
5      Property propagates : Sequence<> =
6          < AirExtractorOnException, EmergencyException, ... >;
7      Property catches : Set{} = {AirExtractorOnException,
8          AirExtractorOffException,SensorFailureException};
9     };
10   Port CatcherPortT1 : CatcherPortT =  new CatcherPortT extended with {
11     Property propagates : Sequence<> =
12         < SwitchPumpOnException, EmergencyException, ... >; ...  };
13   Port CatcherPortT2 : CatcherPortT =  new CatcherPortT extended with { ... };
14   Port SignalerPortT0 : SignalerPortT =  new SignalerPortT extended with {
15     Property catcherPorts : Set{} =
16         {CatcherPortT0, CatcherPortT1, CatcherPortT2};
17     Property signals : Set{} = {EmergencyException};
18   };
19 };
```

Fig. 15. Partial ACME specification of the ControlStation component.

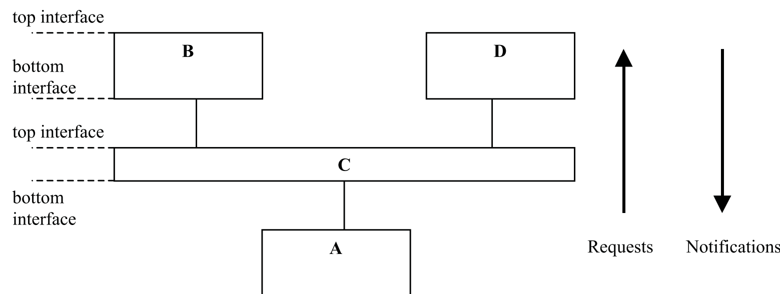

Fig. 16. An example of software architecture based on the C2 style.

The second alternative design for the mining system was an exercise aimed at assessing the difficulty of describing an exception flow view for an architecture that adheres to more than one architectural style.

In the C2 style, components communicate by exchanging asynchronous messages sent through connectors, which are responsible for the routing, filtering, and broadcasting of messages. Figure 16 shows a software architecture using the C2 style where the elements A, B, and D are components, and C is a connector. A configuration is a set of components, connectors, and links between these elements. Components and connectors have a *top interface* and a *bottom interface* (Figure 16). Systems are composed in a layered style, where the top interface of a component may be linked to the bottom interface of a connector and its bottom interface may be linked to the top interface of another connector. Each side of a connector may be connected to any number of components or connectors. Two types of messages are defined by the C2 style: requests, which are sent upwards through the architecture, and notifications, which are sent downwards.

As in Phase 1, we started by defining exceptional styles corresponding to the normal styles used in the architecture of each system. In the case of C2, we also had to specify the ACME family corresponding to the normal style. Specifying the exceptional C2 style in ACME was more difficult than doing the same for the layered and Client/Server styles. There were two reasons for this: (i) C2 has more complex topological rules than the styles we had seen so far and the corresponding exceptional style must respect these rules; and (ii) since connectors in C2 can be linked directly and ACME does not allow this, we had to model both C2 components and C2 connectors as ACME components. ACME connectors were used as links. Figure 17 presents an invariant from the ExceptionalC2Fam ACME family. This complex invariant states that there can only be an exception duct linking the signaler port of a component C1 and the catcher port of a component C2 if there is also a normal connector between these components such that: (i) the bottom interface of component C1 is linked to the top interface of the connector; and (ii) the top interface of component C2 is linked to the bottom interface of the connector.

```
1 invariant Forall conn : ExceptionalC2MessageBusT in self.connectors |
2  Exists normalConn : C2MessageBusT in self.components |
3   (Forall c2 : ExceptionalC2ComponentT in self.components |
4    (reachable(normalConn, c2) AND (Exists cp : CatcherPortT in c2.ports |
5     Exists cr : CatcherRoleT in conn.roles |
6      (attached(cp, cr)) ) ) -> (Exists link2 : C2LinkT in self.connectors |
7      (Exists tdp : TopPort in c2.ports |
8       Exists bdp : BottomPort in normalConn.ports |
9        size(intersection(tdp.attachedRoles, link2.roles)) > 0 AND
10       size(intersection(bdp.attachedRoles, link2.roles)) > 0 AND
11       size(intersection(tdp.attachedRoles, bdp.attachedRoles)) == 0 ))) AND
12       (Forall c1 : ExceptionalC2ComponentT in self.components |
13        reachable(normalConn, c1) -> (Exists sp : SignalerPortT in c1.ports |
14        Exists sr : SignalerRoleT in conn.roles |
15         (attached(sp, sr)))  -> (Exists link1 : C2LinkT in self.connectors |
16         (Exists bdp : BottomPort in c1.ports |
17          Exists tdp : TopPort in normalConn.ports |
18           size(intersection(bdp.attachedRoles, link1.roles)) > 0 AND
19           size(intersection(tdp.attachedRoles, link1.roles)) > 0 AND
20           size(intersection(bdp.attachedRoles, tdp.attachedRoles)) == 0)));
```

Fig. 17. An invariant defined by the ExceptionalC2Fam ACME family.

After finishing the ACME definition of the exceptional C2 style, we specified the C2-based exception flow view of the mining system. This task was straightforward because of the experience acquired in Phase 1. We then proceeded to generate the Alloy specification, and analyze exception flow. The AA reported that the specification satisfied all the basic and desired properties.

The second alternative design for the mining control system uses three different architectural styles: Publisher/Subscriber, layered, and Client/Server. The subconfiguration that comprises components OperatorInterface and ControlStation adheres to the Client/Server style. The subconfiguration comprising ControlStation, MineralExtractor-

Control, PumpControl, AirExtractorControl, and the sensor components adheres to the Publisher/Subscriber architectural style. Finally, the subconfiguration that comprises the three actuators and their corresponding control stations adheres to the layered style. Some of the components play different roles in different subconfigurations. For example, ControlStation is a server in the first subconfiguration, and a publisher and subscriber in the second.

The exception flow view for the second alternative design only uses two exceptional styles: exceptional Publisher/Subscriber (ExceptionalPubSubFam ACME family) and exceptional Client/Server. It does not adhere to the exceptional layered style because no exceptions flow between the components of the layered subconfiguration. Figure 18 presents a diagram of the system's extended architecture description. The larger rectangles represent components, the circles and the vertical bar are normal connectors, and the small rectangles and the losangle with the Exc label are exception ducts.

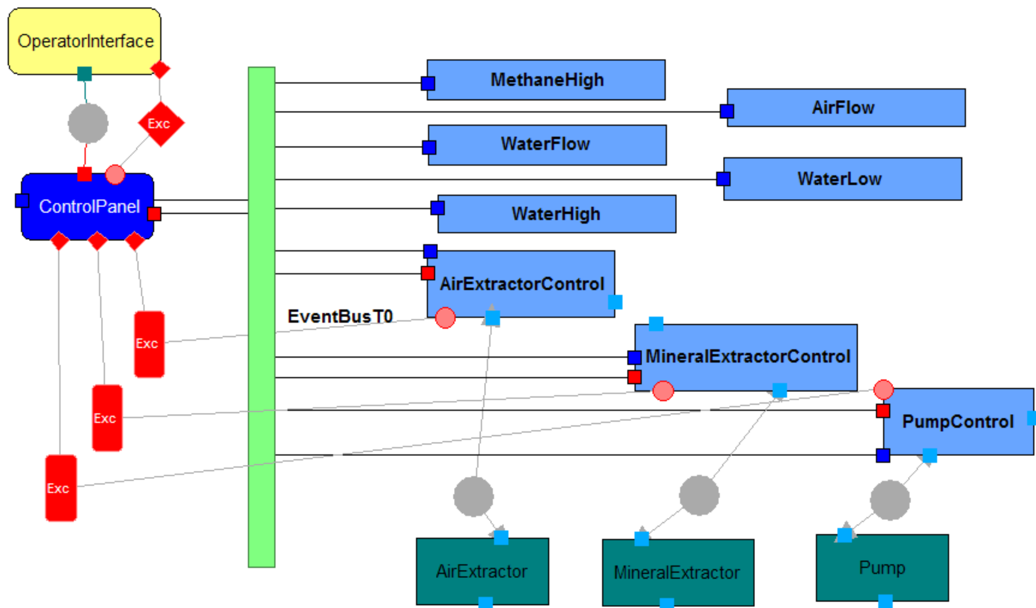

Fig. 18. Extended architecture description of the multi-style mining control system.

5.1.3 Discussion

By modeling the flow of exceptions in the architecture of the mining system, we perceived that, even though proper handling of errors required manual intervention from the system's operator, the latter did not receive any information about the nature of these errors. This was stressed by the fact that the component responsible for handling exceptions did not directly interact with the operator, nor provided any information to components that did. Arguably, applying the Aereal approach to the

mining control system gave us a deeper understanding of the system and, thus, helped in uncovering a design problem.

The experience of using three different approaches to model the mining system was positive. The ACME language allows the specification of different exceptional styles and adoption of more than one style by the same architecture description. Since Aereal is based on ACME and adopts a structural approach for the specification of exception flow, it leverages the features supported by the language. As shown in Section 5.1.2, it was possible to describe even complex exceptional style invariants. Furthermore, the task of describing a multi-style exception flow view was straightforward because all the exceptional styles have to extend the same ACME family and adhere to a predefined set of design rules (Section 4.1). This approach makes the specification of exception flow views uniform and, to a certain degree, independent of different exceptional styles. Furthermore, the Alloy specification abstracts away style-related information (Section 4.4) and retains only structural and exception-related information from the exception flow views. Therefore, in spite of the differences between the three designs of the mining system, the generated Alloy specifications for them are almost identical.

The first phase of the case study required approximately 13 developer hours. The second phase, including the specification of the C2Fam and ExceptionalC2Fam ACME families, required 16 developer hours.

5.2 Financial System Case Study

This case study consisted on developing part of a financial system with strict dependability requirements. The system belongs to the domain of banking applications and was being developed by a medium-sized Brazilian company specialized in banking automation. The part of the system that we used for the case study supports six basic operations:

(1) Solicitation of checkbooks. The customer requests a check-book (in person, by phone, through the Internet, etc.).
(2) Delivery of checkbooks. The system manages the delivery of previously requested check-books.
(3) Cancellation of checks. In cases of loss, theft, or another specific reason, the customer can cancel checkbooks.
(4) Retention of checks. The retention of a check occurs during a deposit in check. These checks are restrained and processed for future payment.
(5) Cancellation of accounting contract. At any time, the customer can lose the credit of his account. In these cases, the contract of the customer must be canceled.

(6) Inclusion of additional limit. Depending on necessity and credit conditions, the customer can receive an additional credit limit.

Operations #1-3 can be initiated by an operator or by a customer. Operations #4-6 can only be initiated by an operator of the system. Operations Cancellation of Accounting Contract and Cancellation of Checks have very strict availability requirements and should be online 24/7. The operations presented above were described as use cases. In this paper, for the sake of simplicity, we focus on the Cancellation of Accounting Contract use case. A very detailed specification of the financial system is available elsewhere [33]. The specification of each use case includes the input information expected by the system, normal, alternative, and exceptional scenarios, and assertions (pre and postconditions). Exceptional scenarios were derived from violations of assertions and alternative scenarios triggered by errors.

The financial system was developed using the MDCE+ method [27], which is an extension of the UML Components process [34] that includes activities for specifying a system's exceptional activity. The case study was planned by the authors and executed by two other developers, one of them a specialist in the domain of the application. The validation of the specified architecture was performed by one of the authors. The goal of this case study was twofold: (i) to gather experience from the use of Aereal in an industrial setting; and (ii) to assess the scalability of the Aereal approach.

The architecture of the financial system adheres to the layered architectural style. It has four layers: user interface, system, business, and database. The system and business layers implement the application-dependent and application-independent business rules, respectively. Components in the business layer can be reused across different applications from the same domain, since they are application-independent. The user interface and database layers are self-explanatory. Figure 19 presents part of the components and connectors view of the financial system. Following the previously adopted notation, components are represented by rectangles and normal connectors by circles. The diagram depicts the part of the system's architecture that is relevant to the Cancel Accounting Contract use case. Component AccountOps from the system layer implements the business logic of the use case. To provide its intended functionality, AccountOps interacts with three components from the business layer, AccountMgr, AgencyMgr, and ParticipantMgr. The existence of two AccountMgr components stems from availability requirements of the system.

The system can fail in several ways and for a number of different reasons. For example, the AccountOps component alone signals 15 different types of exceptions. In total, there are more than 50 types of exceptions that flow between architectural components. This large number of exceptions stems from a development policy adopted by the organization where the case study was conducted. According to this policy, even

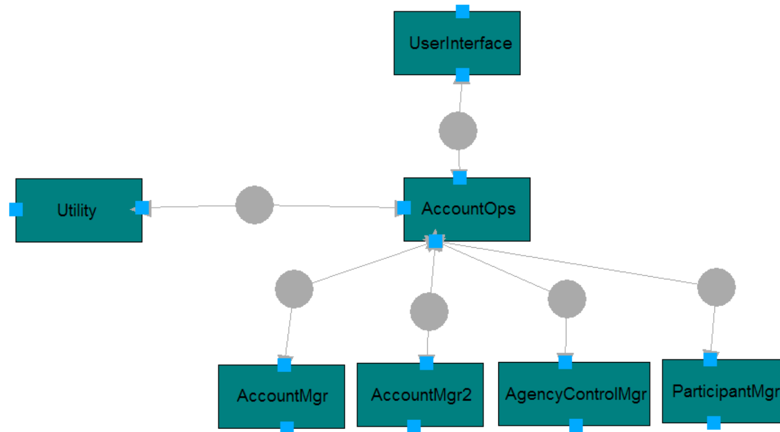

Fig. 19. Partial components and connectors view of the architecture of the Financial System

very similar errors and situations that are not normally treated as errors generate new exception types. Table 4 shows some of the exceptions that AccountOps may signal.

Table 4
Some exceptions signaled by the AccountOps component.

Exception	**Description**
InvalidAgencyException	The provided agency number does not belong to any valid agency.
InvalidAccountException	The provided account number does not match any of the accounts of the customer.
AlreadyCanceledException	The accounting contract for the provided account has already been canceled.

5.2.1 Applying Aereal

We created four different exception flow views for the architecture of the financial system. The exception flow views specify the exceptional activity of the system in the execution of use cases Solicitation of checkbooks, Delivery of checkbooks, Cancellation of checks, Retention of checks, and Cancellation of accounting contract. It was not necessary to define new ACME families for this case study because the architecture of the system adheres only to the layered architectural style. The effort necessary to describe the four exception flow views based on the development documentation of the system was approximately 8 developer hours.

Exception flow analysis was performed in two phases. In the first phase, we composed each exception flow view with the architecture description and generated an Alloy

specification for each resulting extended architecture description. During exception flow analysis, the AA pointed out some misspellings and omissions in the Alloy specifications. These problems were a consequence of errors in the exception flow views. After fixing the errors, we attempted to verify the Alloy specifications again, but the AA produced counterexamples indicating that the specifications did not satisfy desired property DP2 (Section 4.4). Figure 20 shows the formal specification of this property in Alloy. The operators `-`, `=>`, and `#` mean set subtraction, logical implication, and set cardinality, respectively, and the declaration `let` associates an alias to an expression. For each component in the Alloy specification, predicate `dp2()` selects all the exceptions that the component catches but does not handle and checks if it propagates them. It is important to stress that the terms "handle" and "propagate" follow the terminology introduced in Table 2.

```
1 pred dp2() {
2  all C: Component | let nonHandled = (C.catches - C.handles)
3   | (all CF : C.catchesFrom | #(CF <: nonHandled) > 0 =>
4       ((#nonHandled > 0 => #(C.propagates) > 0) &&
5        all E: CF.nonHandled |  #(E.(CF.(C.propagates))) > 0))
6 }
```

Fig. 20. Alloy specification of property DP2.

The fact that the Alloy specification does not satisfy property DP2 can be a problem or not, depending on (i) the implementation language of the system, and (ii) the design principles adopted for system implementation. In some languages, such as CLU [35] and Guide [36], a method must handle all the exceptions it catches, even if handling consists in simply re-raising the exceptions. Moreover, there are methodologies [27,10] that advocate the use of explicit exception propagation even for languages that do not require it, such as Java and C#. In these two cases, a specification of the flow of exceptions in the system should satisfy property DP2. Otherwise, it will not faithfully represent the assumptions made about how the system will be implemented.

In the second phase of exception flow analysis, we composed all the exception flow views with the architecture description. The resulting extended architecture description provides a complete picture of how exceptions flow between architectural elements in the financial system. Moreover, it makes it easier to understand how the exception flow views interfere with one another. Figure 21 shows the result of composing the four exception flow views. For simplicity, components and connectors that are not related to the system's exceptional activity are not shown. From the extended architecture description, we generated the Alloy specification, which comprised 9 components, 8 ducts, and 40 exceptions, and employed the AA to perform exception flow analysis. To our surprise, the AA ran out of memory in a few minutes and terminated verification abnormally. We attempted to change some parameters of the AA, allocate more memory for the JVM (the AA is a Java application), and move verification to a

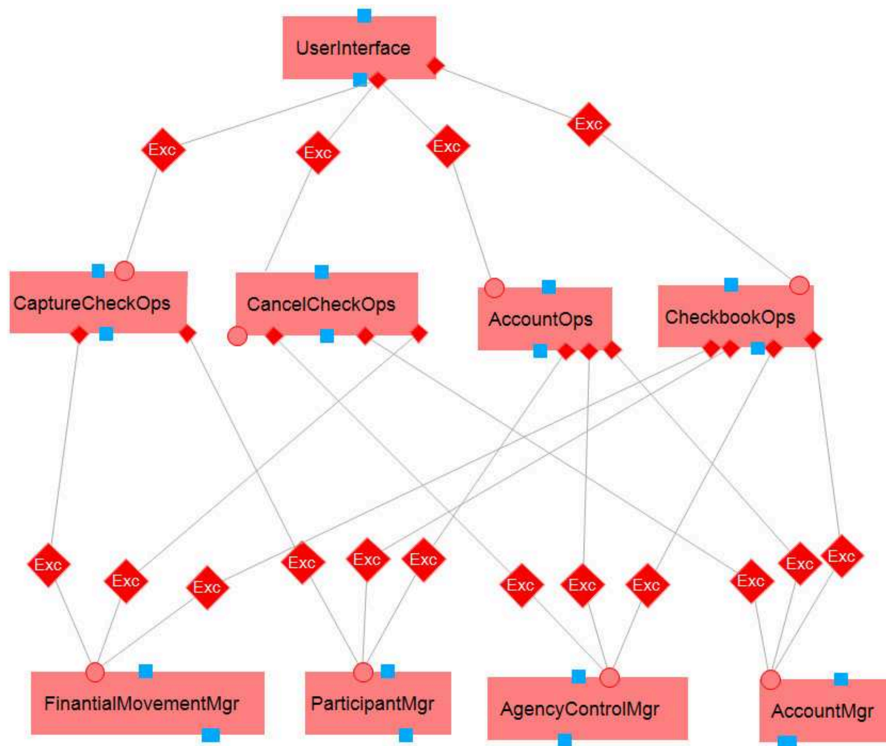

Fig. 21. Composition of the Financial System's exception flow views.

PC with twice the amount of RAM memory (from 512MB to 1GB), but the AA still could not reach the end of the verification.

5.2.2 Discussion

Aereal provided valuable assistance in the task of finding mistakes in the specification of the Financial System's exceptional activity. Although most of these problems are simple to correct, failure to address them can result in problems that are harder to correct in later phases of development. Moreover, it was possible to automatically validate a policy adopted by the EHSs of some programming languages without having to actually implement the system. The results we obtained from this case study and the one described in Section 5.1 do not demonstrate the universal usefulness of Aereal in the construction of software systems with strict dependability requirements. They do show, however, that the Aereal approach is useful in some cases, and justify further studies that can provide more substantial evidence.

This case study has shown that scalability is still a limitation of the Aereal approach. For an Alloy specification comprising 9 components, 8 ducts, and 40 exceptions, it was not possible to perform exception flow analysis. This problem does not represent

a general trend, as we have successfully conducted five case studies so far and 40 is an unusually large number of exceptions visible at the architectural level. It requires immediate attention, nevertheless, because it was detected during the development of a real application, not a textbook example. Section 7 points directions for future work in this area.

6 Related Work

6.1 Architectural Analysis and ADLs

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 [37] specifications can be translated to CSP and analyzed for deadlock freedom and interface compatibility. Rapide [38] is based on partially-ordered event sets. The language supports simulation of architecture descriptions and analysis of the event patterns produced by components. Darwin [39] is based on π-calculus and can be used to specify dynamic software architectures. Abowd and his coleagues [40] use Z to formalize and compare architectural styles.

ADLs such Wright and Rapide, which target the specification of system behavior, have many interesting features and could be used to analyze exception flow in architecture descriptions. However, these languages fail to address some important requirements. For instance, it is not possible to define a type hierarchy for events in Wright. Hence, it is not possible to check if a component catches exceptions by subsumption [6] . Moreover, Rapide does not support the notion of architectural style. Although the language allows certain styles to be simulated, this is not possible in many commonplace situations [41]. A more general problem is that these ADLs do not separate the specification of a system's normal and exceptional activities. This separation decreases the impact of error recovery mechanisms on the overall complexity of the system [5,42,43]. Finally, using these ADLs and associated tools to analyze exception flow in software architectures would require that developers specify the EHS to be supported from the ground up, a cumbersome and daunting task. In this work, we propose a more specific approach for analyzing exception flow in architecture descriptions. On the one hand, it does not present the aforementioned shortcomings. On the other hand, it has a

[6] An exception E is caught by subsumption if it is caught by a catch clause that targets a supertype E' of E.

narrower scope and is not intended to be a general-purpose solution for architecture design.

Some works have focused on formally characterizing architectural styles and using them as a basis for analysis of software architectures [12,25]. Aereal builds upon these works but focuses on the analysis of exception flow at the architectural level. To the best of our knowledge, this is an aspect of software architecture that has not been addressed in the literature.

Approaches that take a formal description of a software architecture as starting point to produce code that preserves reliability-related properties achieved by the architecture have been proposed with different motivations. SADL [44] is an ADL for building hierarchies of architectural descriptions where descriptions located at lower levels of a hierarchy are refinements of descriptions at upper levels. Theorem proving techniques are used to show that refinements are valid and preserve safety, security, and fault tolerance properties. Saridakis and Issarny [45] attempt to characterize the semantics of software architectures from the standpoint of reliability properties. This characterization makes it possible to reuse software architectures in different systems while guaranteeing that the desired reliability properties are maintained in all refinement steps Our work differs from these works because it focuses on the verification of specific properties related to exception flow. These properties can be verified in the context of architectural styles, architectural elements, or whole systems. The aforementioned works emphasize more general fault tolerance properties related to the behavior of a system as a whole. In this sense, our work is complementary to previous efforts. Moreover, Aereal separates the definition of the normal and exception activities of a system, whereas previous works do not make this distinction. Finally, these works do not attempt to model the EHS of existing programming languages. This restricts their applicability because exception handling is usually employed in the implementation of fault tolerance mechanisms.

6.2 Exceptions at the Architectural Level

In recent years, several works proposing the use of exception handling at the architectural level to build dependable systems have appeared in the literature. Bass et al [26] report that, during the development of an air-traffic control system, a system with high dependability requirements, it was necessary to devise a new architectural view that explicitly represented the flow of exceptions between components in the system. This view should provide enough information to help developers to understand how the system deals with errors. Section 3.1 described this work in detail.

Issarny and Bantre [46] describe an extension to existing ADLs for specifying global invariants whose violations are called "configuration exceptions". This work emphasizes fault treatment [5] at the architectural level, by means of architecture reconfiguration. The concept of idealized C2 component [42] defines a structure for associating exception handlers to architectural components that adhere to the C2 architectural style [31]. Castor et al [14] have refined the notion of idealized C2 component and proposed an architectural EHS and implementation infrastructure addressing the specific concerns of component-based systems. Unlike the Aereal approach, these works do not provide means for defining how exceptions flow in different architectural styles. Moreover, they do not focus on the description and analysis of exceptions at the architectural level. Although the work on configuration exceptions presents a proposal for extending existing ADLs with information about exceptions, it focuses on a very specific type of exception that is not signaled or handled in the traditional sense (because exceptions do not flow between architectural elements). Hence, it is not straightforward, based on this proposal, to specify what exceptions are signaled or caught by a given component.

Some authors have proposed frameworks [47,48] that support the detection and handling of exceptions in component-based platforms, such as J2EE. These frameworks provide means to introduce error detection mechanisms, such as monitors, pre, and postconditions, in components in a non-invasive way. Moreover, they define hotspots that simplify the implementation of handlers and the association of these handlers with components. These implementation infrastructures are complementary to Aereal. They provide a link between architecture description and system implementation when designing for specific architectural styles and component platforms.

6.3 Exception Flow Analysis

Several works [49–52] 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. Robillard and Murphy [50] present a brief survey of these techniques and tools.

Our approach leverages previous proposals for exception flow analysis, most notably Schaefer and Bundy's [51], but differs in focus. On the one hand, the Aereal approach targets the early phases of development and is broader in scope. It prescribes a conceptual framework for documenting and analyzing the flow of exceptions between architectural elements, and provides mechanisms to integrate this conceptual framework with existing tools for architecture-centric software development and soft-

ware verification. Furthermore, since the emphasis of Aereal is on the architectural design phase, it explicitly takes into account the concept of architectural style. On the other hand, static analysis tools are employed when an application is already implemented, mainly to find bugs in exception handling code and to improve program understanding.

7 Conclusions and Future Work

In this paper we presented Aereal, a framework for analyzing exception flow in software architectures. Aereal works as an adaptable architectural-level EHS, that is, developers can add or remove constraints to the EHS according to their needs. Due to its combination of ACME and a structural approach for representing exception flow, it is possible to define how exceptions are propagated in a style-specific manner. This is a powerful feature since different architectural styles usually impose different constraints on how components communicate.

Developers of systems with strict dependability requirements benefit from using Aereal in the following ways: (i) by better documenting their decisions about exceptions that flow amongst architectural elements; (ii) by making explicit their assumptions about the EHS of the language(s) that will be used during system implementation; (iii) by checking structural constraints imposed by exceptional styles; and (iv) by verifying inconsistencies between the architecture description and the assumed EHS. Moreover, Aereal completely separates the exception flow view of a software architecture from its normal components and connectors view, and provides tools to compose these views automatically. This feature promotes better understandability and maintainability because concerns do not get cluttered in a single architecture description.

There are currently several limitations to the Aereal approach, besides the ones discussed in Section 5. The notion of architectural style is based on ACME's and, as such, focuses only on the structure of a system. However, an architectural style involves several other issues [28,41], for example, communication, control flow, and data flow. Hence, our approach for describing exception flow is effective only as long as it is possible to describe exception flow abstractly in terms of system structure. Also, there are still no automated means for extracting an exception flow view from the implementation of a system. Tools such as Jex [50] provide some help in this task, but do not implement a complete solution and further studies are required to evaluate how productive this approach would be. Improving traceability between code and architecture is an active area of research [53,54].

The evaluation of Aereal conducted so far indicates directions for future work. First, and most obvious, is improving scalability. We envision two complementary approaches to alleviate the scalability problems we discovered while conducting the Financial System case study. The first is to optimize Aereal's system model by removing redundant information, in order to decrease the amount of RAM memory required to analyze exception flow. For example, ACME properties signals and catches (Section 4.2) and their representations in Aereal's system model could be unified into a single abstraction. The two exist only because of historical reasons. 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.

The second area of future work is further evaluation. We would like to gain more substantial experience in specifying exceptional styles that dictate how exceptions flow in complex architectural styles. Two promising candidates are the Mission Data System [55] architectural style and the C2-based style supported by the PRISM-MW [56] middleware infrastructure. Furthermore, we still have not attempted to evaluate the usefulness of the Aereal approach in the context of software evolution. We believe the framework can be a valuable tool to detect inconsistencies and conflicts introduced by software architecture evolution in a system's exceptional activity.

An interesting issue that arises with the composition of exception flow views and architecture description is what exactly does it mean to adhere to an architectural style. For example, since one of the invariants of the ClientAndServerFam ACME family states that a client has exactly one port, does adding a catcher port to a client component in a Client/Server architecture makes that component conceptually invalid? In terms of ACME invariants, the answer is "yes". In this situation, it is necessary to modify violated invariants of the "normal" style in order for the extended architecture description to be valid. For the invariant above, this would mean stating that a client has exactly one port of type ClientPortT. This modified invariant expresses the intent of the original invariant without being overly restrictive. In the studies we conducted so far (five case studies involving 10 different ACME families, normal and exceptional), this kind of modification was sufficient to accommodate the introduction of exceptional styles. Further evaluation is required, though, to understand if it applies in general.

8 Acknowledgments

We would like to thank the anonymous referees for providing several useful comments, corrections, and suggestions. The authors are partially supported by FINEP/Brazil under grant 1843/04 of CompGov, which is a project for a Shared Library of Components for e-Government.

References

[1] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Addison-Wesley, 1996.

[2] P. C. Clements, L. Northrop, Software architecture: An executive overview, Tech. Rep. CMU/SEI-96-TR-003, SEI/CMU (February 1996).

[3] L. Bass, P. C. Clements, R. Kazman, Software Architecture in Practice, 2nd Edition, Addison-Wesley, 2003.

[4] N. Medvidovic, R. N. Taylor, A framework for classifying and comparing architecture description languages, in: Proceedings of Joint 5th ACM SIGSOFT Symposium on Foundations of Software Engineering/6th European Software Engineering Conference, 1997, pp. 60–76.

[5] T. Anderson, P. A. Lee, Fault Tolerance: Principles and Practice, 2nd Edition, Springer-Verlag, 1990.

[6] F. Cristian, Exception handling, in: T. Anderson (Ed.), Dependability of Resilient Computers, Blackwell Scientific Publications, 1989, pp. 68–97.

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

[8] D. Reimer, H. Srinivasan, Analyzing exception usage in large java applications, in: Proceedings of ECOOP'2003 Workshop on Exception Handling in Object-Oriented Systems, 2003.

[9] J. Viega, J. M. Voas, Can aspect-oriented programming lead to more reliable software, IEEE Software 17 (6) (2000) 19–21.

[10] C. M. F. Rubira, R. de Lemos, G. Ferreira, F. Castor Filho, Exception handling in the development of dependable component-based systems, Software – Practice and Experience 35 (5) (2005) 195–236.

[11] R. de Lemos, A. Romanovsky, Exception handling in the software lifecycle, International Journal of Computer Science and Engineering 16 (2) (2001) 167–181.

[12] D. Garlan, et al., Acme: Architectural description of component-based systems, in: Foundations of Component-Based Systems, Cambridge University Press, 2000, Ch. 3.

[13] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture: A System of Patterns, John Wiley and Sons, West Sussex, England, 1996.

[14] F. Castor Filho, P. A. de C. Guerra, C. M. F. Rubira, An architectural-level exception-handling system for component-based applications, in: Proceedings of the 1st Latin American Symposium on Dependable Computing, LNCS 2847, Springer-Verlag, 2003, pp. 321–340.

[15] D. Jackson, Alloy: A lightweight object modeling notation, ACM Transactions on Software Engineering and Methodology 11 (2) (2002) 256–290.

[16] B. Schmerl, D. Garlan, Acmestudio: Supporting style-centered architecture development, in: Proceedings of the 26th International Conference on Software Engineering, 2004, pp. 704–705.

[17] R. Monroe, Capturing architecture design expertise with armani, Tech. Rep. CMU-CS-96-163, Carnegie Mellon University, School of Computer Science (1998).

[18] D. Garlan, Z. Wang, A case study in software architecture interchange, in: Proceedings of COORDINATION'99, 1999.

[19] D. Jackson, et al., Alcoa: The alloy constraint analyzer, in: Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland, 2000.

[20] J. R. Abrial, The B-Book Assigning Programs to Meanings, Cambridge University Press, 1995.

[21] D. Jackson, Alloy home page, available at `http://sdg.lcs.mit.edu/alloy/default.htm` (2004).

[22] A. Garcia, C. Rubira, A. Romanovsky, J. Xu, A comparative study of exception handling mechanisms for building dependable object-oriented software, Journal of Systems and Software 59 (2) (2001) 197–222.

[23] W. Weimer, G. Necula, Finding and preventing run-time error handling mistakes, in: Proceedings of OOPSLA'2004, Vancouver, Canada, 2004, pp. 419–433.

[24] P. C. Clements, R. Kazman, M. Klein, Evaluating Software Architectures, Addison-Wesley, 2003.

[25] B. Spitznagel, D. Garlan, Architecture-based performance analysis, in: Proceedings of the 10th International Conference on Software Engineering and Knowledge Engineering, 1998.

[26] L. Bass, P. C. Clements, R. Kazman, Air traffic control: A case study in designing for high availability, in: Software Architecture in Practice, 2nd Edition, Addison-Wesley, 2003, Ch. 6.

[27] P. H. S. Brito, et al., A method for modeling and testing exceptions in component-based software development., in: Proceedings of the 2nd Latin American Symposium on Dependable Computing, LNCS 3747, Springer-Verlag, 2005.

[28] N. R. Mehta, N. Medvidovic, Composing architectural styles from architectural primitives, in: Proceedings of Joint 9th European Software Engineering Conference/11th ACM SIGSOFT Symposium on Foundations of Software Engineering, 2003, pp. 347–350.

[29] F. Castor Filho, P. H. da S. Brito, C. M. F. Rubira, Modeling and analysis of architectural exceptions, in: Proceedings of the FM'2005 Workshop on Rigorous Engineering of Fault-Tolerant Systems, 2005, pp. 112–121.

[30] M. Soman, J. Kramer, Distributed Systems and Computer Networks, Prentice-Hall, 1987.

[31] R. N. Taylor, et al., A component- and message- based architectural style for GUI software, in: Proceedings of the 17th International Conference on Software Engineering, 1995, pp. 295–304.

[32] P. A. C. Guerra, C. M. F. Rubira, R. de Lemos, A fault-tolerant architecture for component-based software systems, in: R. de Lemos, C. Gracek, A. Romanosvsky (Eds.), Architecting Dependable Systems, LNCS 2677, Springer-Verlag, 2003.

[33] P. H. S. Brito, A method for modeling exceptions in component-based development (in portuguese), Master's thesis, State University of Campinas, Brazil (2005).

[34] J. Cheesman, J. Daniels, UML Components, Addison-Wesley, 2000.

[35] B. Liskov, A. Snyder, Exception handling in clu, IEEE Transactions on Software Engineering 6 (5) (1979) 546–558.

[36] S. Lacourte, Exceptions in guide, an object-oriented language for distributed applications, LNCS 512, Springer-Verlag, 1991.

[37] R. Allen, D. Garlan, A formal basis for architectural connection, ACM Transactions on Software Engineering and Methodology 6 (3) (1997) 213–249.

[38] D. Luckham, J. Vera, An event-based architecture definition language, IEEE Transactions on Software Engineering 21 (9) (1995) 717–734.

[39] J. Magee, N. Dulay, S. Eisenbach, J. Kramer, Specifying distributed software architectures, in: Proceedings of the 5th European Software Engineering Conference, 1995, pp. 137–153.

[40] G. Abowd, R. Allen, D. Garlan, Formalizing style to understand descriptions of software architecture, ACM Transactions on Software Engineering and Methodology 4 (4) (1995) 319–364.

[41] E. D. Nitto, D. Rosenblum, Exploiting adls to specify architectural styles induced by middleware infrastructures, in: Proceedings of the 21st International Conference on Software Engineering, 1999, pp. 13–22.

[42] P. A. de C. Guerra, C. M. F. Rubira, R. de Lemos, An idealized fault-tolerant architectural component, in: Proceedings of ICSE Workshop on Architecting Dependable Systems, 2002.

[43] B. Randell, J. Xu, The evolution of the recovery block concept, in: Software Fault Tolerance, John Wiley Sons Ltd., 1995, Ch. 1, pp. 1–21.

[44] V. Stavridou, A. Riemenschneider, Provably dependable software architectures, in: Proceedings of the Third ACM SIGPLAN International Software Architecture Workshop, ACM, 1998, pp. 133–136.

[45] T. Saridakis, V. Issarny, Developing dependable systems using software architecture, in: Proceedings of the 1st Working IFIP Conference on Software Architecture, 1999, pp. 83–104.

[46] V. Issarny, J. P. Banatre., Architecture-based exception handling, in: Proceedings of the 34th Annual Hawaii International Conference on System Sciences, 2001.

[47] K. Simons, J. Stafford, Cmeh: Container-managed exception handling for increased assembly robustness, in: Proceedings of the 7th International Symposium on Component-Based Software Engineering, LNCS 3054, Springer-Verlag, 2004, pp. 122–129.

[48] G. Vecellio, M. Thomas, R. Sanders, Container services for high confidence software, in: Proceedings of the 7th ECOOP Workshop on Component-Oriented Programming, 2002.

[49] M. Fahndrich, et al., Tracking down exceptions in standard ml, Tech. Rep. CSD-98-996, University of California, Berkeley (1998).

[50] M. P. Robillard, 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) (2003) 191–221.

[51] C. F. Schaefer, G. N. Bundy, Static analysis of exception handling in ada, Software: Practice and Experience 23 (10) (1993) 1157–1174.

[52] K. Yi, An abstract interpretation for estimating uncaught exceptions in standard ml programs, Science of Computer Programming 31 (1) (1998) 147–173.

[53] M. Abi-Antoun, J. Aldrich, D. Garlan, B. R. Schmerl, N. H. Nahas, T. Tseng, Modeling and implementing software architecture with acme and archjava, in: Proceedings of the 27th International Conference in Software Engineering (ICSE'2005), St. Louis, USA, 2005, pp. 676–677.

[54] I. Gorton, L. Zhu, Tool support for just-in-time architecture reconstruction and evaluation: an experience report, in: Proceedings of the 27th International Conference in Software Engineering (ICSE'2005), St. Louis, USA, 2005, pp. 514–523.

[55] R. Roshandel, B. R. Schmerl, N. Medvidovic, D. Garlan, D. Zhang, Understanding tradeoffs among different architectural modeling approaches, in: Proceedings of the 4th Working IEEE / IFIP Conference on Software Architecture, Oslo, Norway, 2004, pp. 47–56.

[56] M. Mikic-Rakic, N. Medvidovic, Adaptable architectural middleware for programming-in-the-small-and-many, in: ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, 2003, pp. 162–181.