Composition and Testing of Connection Fault Handling Behaviors in Programs with AND/OR Graph

CHIA-CHENG LEE, YU CHIN CHENG AND CHIN-YUN HSIEH
Department of Computer Science and Information Engineering
National Taipei University of Technology, Taiwan 10608
E-mail: t101599006@ntut.edu.tw, {yccheng; hsieh}@csie.ntut.edu.tw

To programs running on components in a distributed system, both network and component failures manifest as connection faults that are represented by exceptions. Although many strategies are available for handling connection faults, it is often necessary to compose multiple strategies in such programs, especially in Internet of Things systems and cyber-physical systems. Moreover, it should be possible to specify a program’s connection fault handling behaviors prior to implementation. Having observed the lack of appropriate design-level constructs, we propose an extended AND/OR graph for modeling composite connection fault handling behaviors in programs. The extended AND/OR graph enables succinct representations of complex fault handling behaviors by composing the constituent strategies of retrying, communicating failure, and ignoring failure. Furthermore, we develop a model-based testing framework that accepts the extended AND/OR graph specification as input and generates tests for checking the connection fault handling behaviors of a program constructed from the specification. The proposed method is illustrated with a Java program to detect and report failure of an device. In particular, the AND/OR graph specification stipulates that the program handles connection faults covering network failures by composing retrying strategy. From this specification, tests are generated by finding all solution trees of the AND/OR graph subject to the number of retries to cover all the paths that lead to the program’s normal and exceptional exits, respectively. The extended AND/OR graph and the model-based testing framework contribute to the means for specifying and testing connection fault handling behaviors of programs.

Keywords: connection fault, exception handling, AND/OR graph, model-based testing, aspect-oriented programming

1. Introduction

A program running on a computer solves a problem by being connected, through the execution environment provisioned by the computer, to the context where the problem is located. Through the connection, relevant phenomena – events and states – are shared between the program and the problem context [1]. The efficacy of the program depends on the quality of the connection and the correctness of the phenomena shared through the
connection: a failed connection could isolate the program from its context and corrupted shared phenomena could prevent it from computing the correct result.

Failed connections and corrupted shared phenomena are aspects of availability and robustness of software [2] that become pervasive in Internet of Things (IoT) applications [3][4][5] and cyber-physical systems (CPS) [6]. A typical IoT application can easily comprise thousands of nodes interconnected through networks of varying degree of reliability. Battery-powered nodes that die or network transmissions that fail are common failures. Programs that run on these constituent nodes should be built to handle unreliable connections and corrupted sensory data and to withstand node and subsystem failures [4][7]. Thus, regardless of the role a node plays in a system – as a server, a hub controller, or a sink node directly connected to sensors and actuators – a node’s capability to cope with failed connection and corrupted shared phenomena is critical to its success in the system [5][8].

In this research, we shall focus on tolerating connections faults through exception handling [2][9]. As long as a program connects to the problem context to share phenomena, connection faults beyond the program’s control will always be present. Between being completely robust and being totally broken, a connection can be intermittent due to certain transient conditions [10]; while a connection is in good order most of the time, it could appear broken when needed. Here are two examples.

- Consider a mobile application that uses a LTE or WiFi connection to access remote services where the acquired connection is intermittent due to terrains or weak wireless airwaves; a required database retrieval could not be completed through the acquired connection because of heavy network traffic bounded for the database server; and so on. Such transient conditions are instances of active connection faults.

- Consider the use of a smart phone as the edge sink/controller in a smart home application that contacts sensors and appliances deployed in a living room through Bluetooth, NFC, or WiFi [5]. Obviously, it is highly desirable for the application to use the best connection. However, when the best connection is not available, it should automatically switch to an alternative connection with a negligible delay.

A program that easily fails on transient connection faults is fragile and not suitable for distributed, mobile, and IoT applications. Usually, connection faults can be tolerated by retrying the failed operation, possibly after a small length of time of waiting. Since retry incurs minimal overhead, accepting failure or attempting some other more expensive strategies (e.g., retrying with an alternative connection) become justified alternative options only after a reasonable number of consecutive retries have failed [11].

Indeed, the need to tolerate transient faults has been well recognized by the development community. In addition to the exception handling constructs of most programming languages, many widely used application programming interfaces (APIs) are available to
developers in composing programs that handle connection faults [12][13][14]. For example, the Microsoft Enterprise Library provides fault detection strategies and backoff strategies (i.e., number of retries to attempt in case of failure and the length of time to wait between two retries), the selections of which are wrapped in an object called the transient fault handling application block, which provides a surrogate around the method to be retried to ride out a transient fault [12].

Both the use of exception handling constructs and the use of APIs can be viewed as implementation-level means for handling connection faults. As pointed out in [2], this often has a consequence that program-level availability is inadequately planned and the resulting program is insufficiently tested [15].

To make connection fault handling a part of the specification of the program under development, a more abstract way above the level of language features or APIs is needed. In particular, the abstraction should be able to express the composition of exception handling operations for implementing the exception handling strategies - including retrying, state restoration, and error reporting [11] – once the situation of the applicable strategy has been determined [16]. Further, the abstraction should also serve as a foundation to automate the subsequent technical tasks, including generating the tests to check an implementation’s conformance to the specification [17].

In this paper, we extend the AND/OR graph for structured programming [18] to compose the connection fault handling behaviors of a program. A node in the extended AND/OR graph is called an operation, which extends an action in AND/OR programming [18] or a statement in structured programming [19], that terminates with or without an exception. To the collection of basic operations available in the AND/OR programming, we propose to add two additional exception handling operations derived from the try-catch-finally construct that are available in most programming languages. With the extended collection of operations, operations that tolerate connection faults are constructed through the AND and OR compositions [18]. With the extended AND/OR graph, we make two contributions. First, as an artifact that focuses on specifying the connection fault handling behaviors of programs, the extended AND/OR graph complements the existing design artifacts such as sequence diagrams and communication diagrams [20], which focus on specifying the normal behaviors of programs. Second, we demonstrate that the extended AND/OR graph can be used as a model in model-based testing (MBT) [17]. In so doing, we have developed a framework to generate unit tests from the AND/OR graph specification, where both the specification and AND/OR graph search algorithm are implemented with the Prolog programming language [21]. The generated tests, each of which corresponds to a solution tree of the AND/OR graph traversed by the search algorithm, make use of aspect-oriented programming [22] to inject active connection faults as described by the solution tree during testing. A failing unit test indicates that the specification has not been implemented correctly for the particular sequence of connection faults encountered.

The rest of this paper is organized as follows. In Section 2, the state model of an
operation facing connection faults is defined, followed by the definitions of the basic operations and the exception handling operations. Section 3 shows how common strategies for handling connection faults are constructed with the extended AND/OR graph. In Section 4, a working example in composing the connection fault handling behaviors of detecting a device failure function is detailed to illustrate the use of the extended AND/OR graph. In Section 5, we develop a model-based testing framework that generates tests for checking a program’s conformance to its connection fault handling AND/OR graph specification using a detecting a device failure as an illustrating example. A review of some related work is found in Section 6. Finally, we offer our conclusion in Section 7.

2. The extended AND/OR graph for connection fault handling

Our objective is to extend the AND/OR graph formalism to represent a composite operation with the specified connection fault handling behaviors. In order to do this, Section 2.1 defines the connection fault-operation relation and the fault handling behaviors of an operation in response to a connection fault. The basic operations and the proposed extensions of the AND/OR graph for exception handling are detailed in Section 2.2.

2.1 Connection fault, operation, and their relationship

First, we shall define the effects of a connection fault on an operation op. A connection fault is either dormant or active [9] as shown in Figure 1. A dormant connection fault becomes an active connection fault when an external condition breaks the connection; an active connection fault turns dormant when the condition that breaks the connection clears. A dormant connection fault is benign to operation op. That is, it does not affect operation op’s behavior in exchanging information with the external entities. Further, an active connection fault is benign to operation op if the connection with the ongoing active fault is not being used by operation op.

The tolerance of a connection fault by an operation is to ensure that only benign connections are used when the operation communicates with the external entities. Accordingly, an active connection fault causes operation op to exhibit an error if (1) the connection is being used by operation op when the fault becomes active and (2) operation op does nothing to handle or fails to handle the active connection fault.
For the purpose of tolerating connection faults, the behavior of an operation \( \textit{op} \) regarding connection fault can be modeled by the state diagram in Figure 2. Operation \( \textit{op} \) enters its \textit{normal} state \( N \) upon invocation. Regardless of its functionality, operation \( \textit{op} \) exits normally by leaving the state \( N \) and entering the normal accepting state \( X_s \) when all the connections actually used during its execution are benign. When operation \( \textit{op} \) uses a connection with an ongoing active connection fault (state \( A \) in Figure 1), it leaves the normal state \( N \) and enters the \textit{faulting} state \( F \) either by detecting it (e.g., operation \( \textit{op} \) times out in getting data through the connection) or by being informed about it (e.g., operation \( \textit{op} \) throws an exception when establishing the connection). From the faulting state \( F \), operation \( \textit{op} \) either enters the \textit{handling} state \( H \) to mitigate the active connection fault (Section 3), or enters the accepting \textit{erroneous} state \( X_e \) after throwing an exception \( e \). Depending on whether the active connection fault is mitigated (or can be safely ignored) or not, operation \( \textit{op} \) returns to the normal state \( N \) or the faulting state \( F \).

There are two noteworthy points about the state model of Figure 2. First, the state model says nothing about the correctness of operation \( \textit{op} \) when it exits normally in entering \( X_s \); it singularly focuses on modeling the connection fault handling behaviors of operation \( \textit{op} \). Second, the non-presence of an unconditional transition from state \( F \) to state \( N \) or state \( X_s \) in Figure 2 requires operation \( \textit{op} \) not to ignore an active connection fault it has encountered. This requirement ensures that operation \( \textit{op} \) is free of the \textit{internal fault}, or \textit{bug}, of ignoring an encountered active connection fault without checking (Section 3.4). In other words, an operation with behaviors described by the state model in Figure 2 always throws an exception when an encountered active connection fault is not handled.

### 2.2 Operations and the AND/OR composition

With the state models of Figure 1 and Figure 2 in place, we are ready to define the basic operations and exception handling operation of the AND/OR graph for composing connection fault handling behaviors of an operation.
Fig. 3. Operations: (a) an OR-node to model an operation that can terminate with success \( s \) or with failure conveyed by exception \( e \); (b) a guarded operation \( \text{cond} | \text{op} \) where the operation \( \text{op} \) is executed only if the specified condition \( \text{cond} \) holds; (c) a conditional operation \( \text{op}_{\text{cond}} \) can be constructed by forming an OR-node with a number of guarded operations as its branches, where the conditions are evaluated from the left-most branch to the right-most branch; (d) an exception-stopping AND-node to compose an operation consisting of \( k \) constituent operations \( \text{op}_1, \text{op}_2, \ldots, \text{op}_k \)

2.2.1 Basic operations

According to the state model of Figure 2, an operation \( \text{op} \) can terminate normally without raising an exception \( \text{or} \) it can terminate erroneously with an exception raised. This is denoted as \( \text{Result}(\text{op}) \in \{s, e\} \), where \( s \) denotes normal exit (entering state \( X_s \)) without an exception and \( e \) denotes erroneous exit (entering state \( X_e \)). Operation \( \text{op} \) is modeled as an OR-node as shown in Figure 3(a). The nodes \( s \) and \( e \) are instances of a terminal node that represents an operation where no further expansion is possible. By this definition, an operation that does not raise an exception always terminates with the node \( s \); for brevity, such an operation is represented as a terminal node by omitting the node \( s \).

A guarded operation is an operation that is executed only when the specified condition is satisfied [18][23]. In Figure 3(b), a guarded operation is denoted by \( \text{cond} | \text{op} \), where operation \( \text{op} \) is executed only if condition \( \text{cond} \) specified at the inbound edge evaluates to \( \text{true} \). The guarded operation and the OR-composition are combined to express a conditional statement as shown in the conditional operation \( \text{op} \) in Figure 3(c), where \( \text{cond}_1 \) is evaluated before \( \text{cond}_2 \) and \( \text{cond}_2 \) is evaluated only if \( \text{cond}_1 \) evaluates to \( \text{false} \). Note that \( \text{cond}_1 \) and \( \text{cond}_2 \) need not be mutually exclusive. For a conditional operation \( \text{op} \) with \( k \) \((k \geq 1) \) guarded operations \( \text{cond}_i | \text{op}_i \) \((1 \leq i \leq k) \), \( \text{Result}(\text{op}) = \text{Result}(\text{op}_j) \) if there exists a \( j, 1 \leq j \leq k, \text{cond}_j = \text{true} \) and \( \text{cond}_1 \lor \ldots \lor \text{cond}_{j-1} = \text{false} \); otherwise, \( \text{Result}(\text{op}) = s \).

An operation \( \text{op} \) consisting of a sequence of \( k \) \((k \geq 1) \) operations \( \text{op}_1, \text{op}_2, \ldots, \text{op}_k \) is modeled as an exception-stopping AND-node as shown in Figure 3(d). The constituent operations \( \text{op}_1, \text{op}_2, \ldots, \text{op}_k \) are executed sequentially; that is, operation \( \text{op}_i \) is executed before operation \( \text{op}_j \) if \( i < j, 1 \leq i, j \leq k - 1 \). \( \text{Result}(\text{op}) = s \) if all constituent operations \( \text{op}_i, i = 1, \ldots, k \) end in \( s \); \( \text{Result}(\text{op}) = e \) if all operations before operation \( \text{op}_j \) end in \( s \) and operation \( \text{op}_j \) \((j \geq 1) \) ends in \( e \), upon which operations \( \text{op}_i, i = j + 1, \ldots, k \) are not executed. Skipping the latter constituent operations explains why the node of operation \( \text{op} \) of Figure 3(d) is called an exception-stopping AND-node: it terminates the normal execution and propagates to its caller the exception thrown by the failing constituent op-
eration. The exception-stopping AND-node is consistent with the termination model of
the continuation of control flow upon exception [24], which is adopted by all of the pro-
gramming languages analyzed in [25], including C++ and Java. An exception-stopping
AND-node degenerates into a regular AND-node if none of its constituent operations
throws an exception. Where no confusion is possible, the term AND-node will be used
for brevity.

2.2.2 Exception handling operations

The try-statement that is available in most languages is modeled as a composite
operation. Since the semantics of the try-statements cannot be expressed by the basic
operations, two new types of operations are introduced: the try-catch operation and the
try-finally operation.

Operations corresponding to the three variants of the try-statement are depicted in
Figure 4. Figure 4(a) denotes the try-catch operation op, which is stereotyped with the
symbol \( \ll \text{t-c} \gg \). Operation op is composed of the normal operation \( \text{op}_t \) reached through
the branch labeled try and the conditional operation \( \text{op}_c \) reached through the branch
labeled catch for exception handling, respectively. Operation \( \text{op}_t \) is executed uncon-
ditionally. Operation \( \text{op}_c \) is executed only if operation \( \text{op}_t \) throws an exception
\( e \) that matches any conditions in the ordered list \( e = e_1, e = e_2, \ldots, e = e_k \) of operation \( \text{op}_c \).

We assume that \( \text{op}_t \) and \( \text{op}_c \) behave according to the state model of Figure 2. The
result of executing operation \( \text{op} \), \( \text{Result}(\text{op}) \), is determined in three different cases ac-
cording to Figure 2:

- If operation \( \text{op}_t \) exits normally, \( \text{Result}(\text{op}) = \text{Result}(\text{op}_t) = s \). In this case, opera-
tion \( \text{op}_t \) – and therefore operation \( \text{op} \) – makes the transition from state \( N \) to state \( X_t \)
in Figure 2; or

- operation \( \text{op}_t \) terminates in error raising an exception \( e \neq e_j (j \in \{1, \cdots, k\}) \). Op-
eration \( \text{op} \) propagates exception \( e \) and \( \text{Result}(\text{op}) = \text{Result}(\text{op}_t) = e \). In this case, opera-
tion \( \text{op}_t \) – and therefore operation \( \text{op} \) – makes the transition from state \( F \) to state \( X_e \) in Figure 2; or else

- operation \( \text{op}_t \) terminates in error with an exception \( e = e_j \) raised for some \( j \in \{1, \cdots, k\} \). The first matching operation \( \text{op}_j \) of operation \( \text{op}_c \) is executed and
\( \text{Result}(\text{op}) = \text{Result}(\text{op}_c) = \text{Result}(\text{op}_j) \). In this case, operation \( \text{op}_t \) makes the
transition from state \( F \) to state \( X_e \), causing operation \( \text{op} \) to make the transition from
state \( F \) to state \( H \) to invoke the operation \( \text{op}_c \) in Figure 2 to handle exception \( e \).

The try-finally operation \( \text{op} \) is modeled as in Figure 4(b), where the node is stereo-
typed with the symbol \( \ll \text{t-f} \gg \). The operation \( \text{op}_t \) is executed unconditionally. Then,
the cleanup operation \( \text{op}_f \) of the finally branch is always executed regardless of the
result of executing operation \( \text{op}_t \) of the try branch. The try-finally operation \( \text{op} \) has the
Fig. 4. Exception handling operations based on try-statement

result of $\text{Result}(\text{op}_f)$ if $\text{Result}(\text{op}_f) \neq s$; otherwise, it has the result of $\text{Result}(\text{op}_t)$. Note that the try-finally operation $\text{op}$ does not enter the state $H$ of Figure 2; that is, it does not handle any exception thrown by operation $\text{op}_t$.

The try-catch-finally operation can be constructed by plugging in the try-catch operation into the try branch of the try-finally operation and is stereotyped with the symbol $\ll<>t-c-f\gg$; see Figure 4(c).

3. Modeling the common connection fault handling strategies

An operation $\text{op}$ selects from three strategies for handling an active connection fault: retrying, communicating failure, and ignoring failure [26][27][11]. For retrying, there are two options: retrying with the original and retrying with an alternative [28]. If the failure is caused by a bug or by an incorrect action of the user, retrying with the original does not help at all. The strategy retrying with an alternative is applicable to both connection faults and bugs provided that the alternative operation is bug-free. Finally, if the failure is caused by the user, the proper strategy is to involve the user to fix the cause after communicating failure [27].

3.1 Retrying with the original

The strategy retrying with the original is implemented with a surrogate that controls the repeated calls to the original operation. As depicted in Figure 5(a), operation $\text{op}$ is made to tolerate connection faults represented by exception $e$ by inserting the try-catch node $\text{op}(e|I)$ above it, where the notation $e|I$ means that at most $I(I \geq 0)$ retries are attempted for a caught exception of type $e$.

While Figure 5(a) is appropriate as a design level notation for operation $\text{op}(e|I)$, during implementation time, it can be expanded to show further details, including the

\[\text{In general, the surrogate operation could be written as } \text{op}_b([e_i|I_i]), i = 1, \ldots, k, \text{ if there are } k \text{ exceptions for which the strategy retrying with the original is applicable.}\]
Fig. 5. Retrying with the original. (a) Making operation \( op \) tolerate transient faults represented by exception \( e \) by retrying it at most \( I \) times with the surrogate operation \( op(e|I) \); (b) a possible implementation expansion of operation \( op(e|I) \).

screening operation for excluding exceptions from retry and the backoff strategies before retry. Figure 5(b) shows a possible expansion of node \( op(e|I) \). In the expansion, the try-catch node \( op(e|I) \) is composed of the try operation \( op \) and the catch operation \( op \_eh(e|I) \). The AND-node \( op \_eh(e|I) \) is sequentially composed of operation \( screen(e) \) and operation \( retry(I) \). Operation \( screen(e) \) terminates by throwing exception \( eop \) (with exception \( e \) as its cause) if exception \( e \) thrown by operation \( op \) is not in scope for retrying with the original. By being not in scope, we mean that the exception is not appropriate to handle with retrying, e.g., when the exception is caused by an error committed by the user. Ideally, such a user error should be represented by an exception other than \( e \). In practice, however, the same exception type is often used for both connection fault and user error. The exception used in such a practice is called a homogeneous exception [29]. Thus, operation \( screen(e) \) is employed to further check for exceptions representing non-connection faults. Section 4 gives an example of how the screening operation deals with a homogeneous exception.

When \( screen(e) \) exits without throwing an exception, exception \( e \) is appropriate to handle with retrying with the original. This is implemented with \( retry(I) \), which ends either with \( op \) exiting in success or in failure by throwing exception \( eop \) when the number of retries \( I \) has been exhausted. Note that a retry is attempted after a waiting period determined by \( wait(I) \), which could implement an appropriate backoff strategy such as exponential backoff [14].

3.2 Retrying with an alternative

Let \( op \) be an operation called by operation \( op_a \) as depicted in Figure 6(a). The operation \( op_a \) is made to tolerate failure of operation \( op \) by inserting a surrogate try-catch operation \( opReAlt \) between operation \( op_a \) and operation \( op \) and providing an alternative operation \( opAlt \) that is different from but functionally the same as operation \( op \). The
Fig. 6. Retrying with an alternative. (a) Operation $op_a$ calls operation $op$; (b) tolerating failure of operation $op$ represented by exception $e_{op}$ with an alternative operation $opAlt$ through the surrogate operation $opReAlt$.

Fig. 7. Ignoring an exception $e$, where $nop$ stands for no operation.

alternative operation $opAlt$ is executed when operation $op$ raises exception $e_{op}$; see Figure 6(b).

3.3 Communicating failure

When it is appropriate for operation $op_a$ to allow the failure of $op$ to become its own failure, this strategy is modeled with the exception-stopping AND-node and make operation $op$ a child of operation $op_a$ (Figure 3(d)). That is, i.e., both operations $op$ and $op_a$ make the transition from state $F$ to state $X_e$ in Figure 2.

3.4 Ignoring failure

Sometimes it is necessary to ignore a connection failure. An implementation of ignoring failure is shown in Figure 7(a). The surrogate operation $op(\bar{e})$ catches the active connection fault conveyed by exception $e$ but does nothing to fix it (represented by $nop$ - no operation, which always has the result of success $s$.) as indicated by the bar notation over the exception $e$. Note that, however, it may be necessary to screen for exceptions that should not be ignored when the homogeneous exception is used, in which case the expansion in Figure 7(b) is used. Strategy ignoring failure is applicable, for example, when closing a database connection fails after successfully accessing the database. In this case, it is reasonable for the access operation to ignore the failure (probably after logging)
4. An illustrating example with detecting and reporting failure of an analog device

In this section, we illustrate the use of the extended AND/OR graph to compose the connection fault handling behaviors of an operation. The illustrating example is adapted from the Patient Monitoring problem [1]. Our focus is on detecting failures of the analog device.

A patient-monitoring program is required for the intensive care unit (ICU) in a hospital. Each patient is monitored by an attached analog device which periodically measures factors such as pulse, temperature, blood pressure, and skin resistance. The monitoring program reads these factors on a periodic basis. If an analog device fails, the nurses’ station is notified by the monitoring program. In the event that notification fails, an SMS message is sent to the phone of the medical staff on duty.

Figure 8 shows the context diagram of the sub-problem of detecting and notifying failure of analog devices. We are going to assume that the analog device is implemented with a network-capable embedded system such as Arduino [30] or Raspberry Pi [31], which allows the monitoring program to read the vital factors through a RESTful API [32], (e.g., using the library [33] on Arduino.)

An analog device failure can come from three sources:

- The network connection between the monitoring program and the analog device is experiencing a breakage.
- The analog device is broken and exhibits a reading out of the normal range; e.g., a temperature reading that is out of range of living human body temperature (88° ~
A sensor is detached from the patient and exhibits a reading out of the normal range.

Of the three sources, only the first source is related to connection fault handling; the other two sources are domain errors whose detection requires building a domain model of the analog device. For the first source, we shall apply the strategy of retrying with the original to ride out the transient connection fault. For the domain error, we shall apply the strategy of error reporting.

Since these faults can occur during the read operation performed by the monitoring program, we are going to compose the two strategies in the operation \textit{read}. In Java, the signature of \textit{read} is

\begin{verbatim}
void read() throws fad,
\end{verbatim}

where \( f_{ad} \) is an exception representing an analog device failure. Likewise, the operation \textit{notify} has the signature

\begin{verbatim}
void notify() throws fns,
\end{verbatim}

where \( f_{ns} \) is an exception representing failure to notify the nurse station, which is appropriate since either the network connection to the nurse station or the nurse station itself could be down.

The operations are composed into the operation \textit{readOrNotify} with a signature as follows:

\begin{verbatim}
void readOrNotify() throws fns.
\end{verbatim}

Finally, to satisfy the requirement, the operation \textit{monitor} is composed as in Figure 9. In a nutshell, in the event that both analog device and the nurse station fail, an SMS message is sent through the phone network to the medical staff in charge.
Composition and Testing of Connection Fault Handling Behaviors

4.1 Implementing the strategies of communicating failure and retrying with the original

With the top level operation monitor specified as in Figure 9, we can go a step further to consider the fault handling strategies applicable operation read. Figure 10 shows the expanded tree of subtree rooted at operation read in Figure 9. The operation read is composed of the operations get, checkStatus, and checkValue. Specifically, operation get reads a measurement from an analog device and returns the standard HTTP status code used by the RESTful API [32]. The status code is a 3-digit integer where the first digit defines a class of response: a 2 indicates the request has succeeded, a 4 indicates the client error, and a 5 indicates the server error [34]. Operation checkStatus checks for connection fault and throws an instance of exception fad if a client error (return code = 4xx) or a server error (return code = 5xx) is detected, e.g., code 408 for timeout and code 503 for an overloaded device. Finally, operation checkValue checks for domain error and throws an instance of exception fad if the reading has a value that is nonsensical. Note that in either case, the cause is recorded in a data field of exception fad.

Operation read has the semantics of communicating failure, i.e., it deals with both connection fault and domain error by throwing the exception fad. The use of homogeneous exception fad is a common practice for exception handling design [29] [35] [36].

While operation read satisfies the requirement, an intermittent network connection can send an alarm to nurse station. By the time the medical staff on duty reaches the alleged failed analog device, it could happen that no failure has found. This is clearly undesirable and should be further improved.

In Figure 11 operation read (fad | I) is introduced as a surrogate around operation read to ride out transient connection faults by applying strategy of retrying with the original.

The original operation read throws a homogeneous exception fad for both a domain error and a connection fault. Operation screen is introduced to single connection faults for the application of the strategy of retrying with the original. In this case, operation retry is attempted at most I times (I ≥ 0). In each attempt, an appropriate backoff strat-
5. Specification-based testing

The connection fault handling behavior composition graph can serve as a specification for constructing the connection fault handling behaviors of an operation. In particular, the extended AND/OR graph representation of an operation can be easily represented with a rule-based program for generating various artifacts that describe an operation’s connection fault handling behaviors, including code and tests.

In this section, we shall focus on generating tests for checking the connection fault handling behaviors of an operation from the specification. Using terms of AND/OR graph searching [21], a node is said to be solved when it exits either normally or erroneously. A solution tree is a sub-graph of the AND/OR graph where the result of executing the root node, which is the top-level operation whose connection fault handling behavior is being modeled, is known.

Using the example of Section 4, we assume that code of the operation \texttt{read(fad | I)} to detect failure of an analog device has been manually constructed according to the AND/OR graph of Figure 11. We want to know if the implementation conforms to the specification through tests generated from the solution trees obtained by searching the AND/OR graph. Used in this way, the AND/OR specification is a model for generating tests in model-based testing (MBT) [17].

5.1 System model

Figure 12 shows a system that uses an AND/OR specification to generate and run tests for checking the connection fault handling behaviors. We shall assume that the Java environment is used. The system makes use of aspect oriented programming (AOP) with AspectJ [22] and unit testing with JUnit [37].
In Figure 12, the system consists of the following components. Test Generator creates a collection of tests based on a given Specification, which is an instance of the extended AND/OR graph. A generated Test contains code for invoking Operation Under Test as well as Directives to be consulted by Advice on deciding what to insert at a joint point (i.e., a specific invocation of a constituent operation) of Operation Under Test, e.g., whether to let the constituent operation go through normally or to inject an exception to emulate an active connection fault.

The interaction between components are labeled with numbers to be read in increasing order. Tests are first generated by Test Generator (step 1) according to the AND/OR graph specification of connection fault handling behaviors for the Operation Under Test. A Test is run by going through steps 2 to 6. First, Directives to be consulted by Advice are written by the Test (step 2). As the execution continues, Operation Under Test is invoked (step 3) with Advice inserted before specific constituent operations specified by a joint point and then executed (step 4). Advice then reads the previously written Directives (step 5), and performs the required action (step 6), i.e., to continue normally or to throw a designated exception.

5.2 The AND/OR graph in Prolog

The AND/OR graph specification of operation \texttt{read}(f_{ad} | I) in Figure 11 has been translated into the equivalent specification in Prolog in Figure 13. The AND/OR graph is an extension of the implementation of [21].

We now briefly describe the notational conventions in the following user-defined operators. In lines 1-3, the three operators ‘\texttt{\rightarrow\rightarrow\rightarrow}’, ‘\texttt{\cdot\cdot\cdot\cdot\cdot}’, and ‘\texttt{\rightarrow\rightarrow\rightarrow\rightarrow}\’ are infix, right-associative operators of precedences 100, 500, and 600, respectively, with smaller precedence value binding more strongly. Operator ‘\texttt{\rightarrow\rightarrow\rightarrow\rightarrow}\’ is a relation from a node to its successors. Operator ‘\texttt{\cdot\cdot\cdot\cdot\cdot}\’ marks the node type (and, or, try-catch and try-finally) in its first argument and
1 \text{ :- op( 100, xfx, >> ).}
2 \text{ :- op( 500, xfx, ! ).}
3 \text{ :- op( 600, xfx, --> ).}
4 \text{ read-I/R--->try_catch: (read/Rt,}
5 \text{ (eh*I)*Rt/Rc).}
6 \text{ read/R--->or: [s, f(ad)].}
7 \text{ (eh*I)*Rt/R--->and: [screen(Rt)/R1, ['=',Rt,f(ad)]>>(retry*I)/R2].}
8 \text{ (retry*I)/R--->or: ['<='I,0]>>(throw(f(ad))/R, ['>',I,0]>>(read+J)/R}
9 \text{ J is I-1.}
10 \text{ screen(Rt)/R--->or: [s].}
11 \text{ throw(F)/R--->or: [F].}

Fig. 13. An extended AND/OR specification of the Operation Under Test \text{read(fad|I)} in Prolog.

the successor nodes as a list in its second argument. For example, node a with two OR successors b and c is represented by the clause:

\text{a --> or: [b,c].}

The operator '>>' binds the guard condition to a node. In the following clause, node a*I has the successor b if I=<0 and the successor c if I>0, where '*' is a built-in operator of precedence 400 that binds node a with the variable I:

\text{a*I --> or: [I =< 0 >> b, I > 0 >> c].} \footnote{In Prolog, the condition Cond is constructed from a list CondList with the built-in operator '='..' in the clause Cond =..CondList. Thus, the condition I =< 0 is written as [='=<', I, 0].}

Lastly, we use the binary operator / to bind a node and the result R of evaluating the node. Thus, the node

\text{read*I/R}

denotes that operation read is attempted at most I times with R as the result.

With the operators defined and the notational convention out of the way, we are ready to explain the conversion of the AND/OR graph of operation \text{read(fad|I)} in Figure 11 into the Prolog code of Figure 13. In lines 4-5, the node \text{read*I/R} is a try-catch node with node \text{read/Rt} at the try-branch and node \text{(eh*I)*Rt/Rc} at the catch-branch. The node \text{read*I/R} stipulates that operation \text{read} will be attempted at most I times, at the end of which the result R is obtained. The node \text{read/Rt} is always executed with Rt being equal to s for success, f(ad) for a connection fault while detecting an analog device; see line 6. At the catch branch the AND-node \text{(eh*I)*Rt/Rc} comprises operation \text{screen(Rt)/R1} (which always exits successfully with s for simplicity in line 10) and the guarded retry operation \text{(retry*I)/R2} (line 7) that is expanded only if the exception is \text{fad}. The clause in lines 8-9 stipulates that exception \text{fad} is thrown (line 11) if all allowed retries have been exhausted; otherwise, operation \text{(read*(I-1))/R} is executed. For simplicity, the wait operation of Figure 11 is omitted.

5.3 The AND/OR graph search algorithm

Figure 14 shows the key clauses of the Prolog procedure

\text{compose(Node/R, Node/R--->Tree)}

for exploring an AND/OR graph using depth-first search, where Node/R--->Tree is a
composition and testing of connection fault handling behaviors

17

compose( Node/R, Node/R--->Tree) :-
  Node/R-->>or:Nodes,  % OR-node
  member( CondList>>Node1/R, Nodes), % Select a branch if Cond
  Cond =.. CondList,
  call(Cond), !,
  compose( Node1/R, Tree).

compose( Node/R, Node/R--->Tree) :-
  Node/R-->>or:Nodes,  % OR-node
  member( Node1, Nodes), % Select a branch
  compose( Node1/R, Tree).

compose( Node/R, Node/R--->and:Trees) :-
  Node/R ---> and:Nodes,  % AND-node
  composeall( Nodes, Trees, R).  % Solve all Node's successors

% try catch - try success
compose( Node/s, Node/s--->try_catch:TreeT) :-
  Node/s ---> try_catch:(Try/s,Catch),
  compose(Try/s,TreeT).

% try - catch: try throws exception uncaught by Catch
compose( Node/R, Node/R--->try_catch:TreeT) :-
  Node/R ---> try_catch:(Try/R,Catch*R1/RC),
  getLeavesOrNode(Catch*R1/RC,Catches),
  failureCaught(Catches,FC),
  EFC = [s|FC],  % extend the failure list by s
  compose(Try/R,TreeT),
  \+ member(R,EFC).

% try - catch: try throws exception caught by Catch
compose( Node/R, Node/R--->try_catch:TreeC) :-
  Node/R ---> try_catch:(Try/R1,Catch*R1/RC),
  compose(Try/R1,TreeC),
  \+ member(R1,R).

Fig. 14. The key clauses of a depth-first AND/OR graph search algorithm in Prolog.

solution tree with result R. The rule for processing an OR-node with guarded branches of
Figure 3(c) is found in lines 1-6. Line 2 expands a node that matches with an OR-node. In
line 3, operator >> separates the condition (which is the list CondList) from the guarded
operation (which is the term Node1/R). Line 4 assembles the list CondList into a
term, and line 5 calls the assembled term as a goal. For example, in line 4 of Figure 13,
the condition list ['=',Rt, f(ad)] is assembled into the term Rt = f(ad), which
is then executed as a goal by the built-in predicate call. In line 6, the node Node1/R is
executed if the condition Cond in line 4 evaluates to true.

We summarize the rest of the Prolog program in Figure 14 without going into detail.
The clause at lines 8-11 explores the OR-node of Figure 3(a). The clause at lines 13-15
explores the AND-node of Figure 3(d). The clause at lines 17-20 explores a node that
corresponds to a try_catch node where the try clause succeeds without an exception.
The clause at lines 22-29 explores a node that corresponds to a try_catch node where
the try clause throws an exception that is not caught by any of catch clauses. Lastly, the
clause at lines 31-37 explores a node that corresponds to a try_catch node where the
try clause throws an exception that is caught by a matching catch clause.

After loading the programs of Figure 13 and Figure 14, the Prolog query
Fig. 15. Solution tree of the AND/OR graph of Figure 11 where operation \texttt{read} first exits with a \texttt{fad} (when I = 1) and then exits with success (when I = 0). The rectangular nodes are terminal.

\[
\text{- compose(read}^1/R, T). \]

generates all of the possible results R with solution trees T for tolerating at most one active connection fault encountered by operation \texttt{read}. Figure 15 shows the solution tree T where the surrogate operation \texttt{read}^1 ends with R=s after the surrogated operation \texttt{read} first encountered a \texttt{fad} (when I = 1) and then exited with success (when I = 0).

5.4 Generated tests

Figure 16 shows three of the JUnit tests generated by \textit{Test Generator} with the Prolog specification in Figure 13 translated from the extended AND/OR graph of Figure 11 with the Prolog query

\[
\text{- compose(read}^1/R, T). \]

In what follows, we shall explain in detail the test from at lines 1-14 in Figure 16 corresponding to the solution tree in Figure 15.

The intent and result of a test are succinctly summarized in the name of the test (line 2). In this case, the test checks that operation \texttt{read}(1) indeed terminates in success as indicated by the suffix ‘\texttt{ts}’ in the method name, where ‘\texttt{t}’ and ‘\texttt{s}’ denote that the surrogated operation \texttt{read} first encounters a transient connection fault (when I = 1) and a success (when I = 0), respectively.

Line 3 stipulates to tolerate at most one transient fault. Lines 4-7 are the directives written by the test for an advice to consult (see Figure 12 and Section 5.5) and consist of injecting one \texttt{ADFailureException} (implementation of terminal node \texttt{fad}) followed by ‘\texttt{s}’ for the execution of operation \texttt{read}. As a result, the call \texttt{monitoring.read(I)} is expected to complete without throwing an exception; see lines 8 - 13.
An implementation of the surrogate operation `read(I)` with `I = 1` that passes the three tests in Figure 16 conforms to the specification that it retries at most once for the connection fault, and terminates successfully or throws an exception as required by the connection fault handling specification in Figure 11 and Figure 13 and the state model in Figure 2.

5.5 Advice

Figure 17 shows the partial AspectJ implementation of Advice [22]. The pointcut looks for invocations of `Monitoring.read()` (line 2). The before advice stipulates that before an instance of `Monitoring.read()` is executed, one of the two alternative actions is taken. Depending on the instruction read from Directives (line 6) set by the test, it allows the invocation to go through (lines 7-8); replaces it with a mocked `ADFailureException` with a HTTP status code 408 (`Request Timeout`) (lines 9-10).
6. Related work

The present work is based on the previous results in fault handling [38][39] [40][41][25], although with a much narrower scope of modeling and testing connection fault handling behaviors. In what follows, we shall relate the present work to the various techniques used, including AND/OR graph, aspect-oriented programming, and software testing.

AND/OR graphs model problems that are solved by decomposition in artificial intelligence, e.g., in symbolic integration [42], game-playing, theorem proving, and many other problems [43][44]. The use of AND/OR graph to represent a program was proposed in [18]. Indeed, an AND-node subsumes sequential and parallel composition and an OR-node subsumes non-determinism and conditionals; thus, it can be used for structured programming. The extended AND/OR graph proposed in this paper can be seen as parallel to the AND/OR program of [18] by including exception handling operations, but is simplified to solely focus on the specification of connection fault handling behaviors. It should be noted that the extension can be achieved with monads to extend the normal operation to account for exceptions as is used in functional programming [45].

Model-based testing has been proposed for testing standard specification conformance [46]. Basically, standard specification written in a natural language is first translated into a finite state machine description, which is then used to generate tests for testing conformance. It has been found that the tests generated from model were able to exceed the test suites directly crafted from the natural language specification. The AND/OR graph formalism proposed for modeling connection fault handling behaviors and for generating test cases is an instance along the line of MBT research [17]. In particular, the MBT technique implemented in this paper achieves redundancy in generating tests from the AND/OR graphs for checking the conformance of an implemented program to the specification.

In this research, we use aspect oriented programming (AOP) as part of the testing framework to inject exceptions and share states between test cases and the operation under test. It is interesting to note that AOP has been used in extracting exception handling

```java
public aspect AspectConnection {
    public pointcut read() : call( * Monitoring.read() );
    OpRespRepo repo = OpRespRepo.getRepo();
    before() throws ADFailureException : read() {
        String OpRes[] = repo.emitNext();
        if (OpRes[1].equals("s")) {
            // do nothing, call original or provide a viable replacement
        } else if (OpRes[1].equals("f(ad)")) {
            throw new ADFailureException("Analog device fails", 408);
        }
    }
}
```
behaviors to aspects so that normal behaviors and exceptional behaviors are separated [47]. The AND/OR graph composition of connection fault handling behaviors could, in principle, be implemented as aspects to be weaved into the join points identified by point cuts. The main benefit is the potential of reusing the exception handling aspects, especially if only generic and homogeneous handling such as logging and reporting is involved. However, there are limitations when the handler’s behavior depends on the exceptional context [48]. For example, the *retrying with the original* strategy of Figure 11 uses a local variable I in the faulting method to control the number of retries to attempt, which is not available to the extracted aspect. Although this can be fixed by making the control variable I a field of the object that owns the faulting method and by making the object an argument passed to the pointcut and associated advices, the need to specify the object’s class in the parameter declaration limits the aspect’s reuse. One way to get around the limitation is to have the object owning the faulting method implement an interface created solely for the purpose of accessing the control variable. Therefore, it seems unlikely that reuse of aspects is possible without some consideration given to the design of exception handling. In either way, it seems inevitable that endowing an object with connection fault handling behaviors will unavoidably tangle normal and exceptional behaviors.

Numerous researches for exception handling testing based on static analysis of program code have been published [49][50]. In contrast, the proposed method is based on specification and testing. In software development life cycle, static analysis, specification, and testing techniques complement each other for software quality assurance. For example, during requirement analysis and development, the proposed method for AND/OR composition of connection fault handling can be used for specification and testing, while static analysis techniques such as control-flow or data-flow analyses can be applied afterwards to check for further exception handling defects.

7. Conclusion

We have presented the extended AND/OR graph for composing connection fault handling behaviors in programs. The extended AND/OR graph not only enables the description of specification to be followed by developers in building programs with the required connection fault handling behaviors, but can also be used to generate tests for checking a program’s conformance to the specification. The extended AND/OR graph and the model-based testing framework are illustrated with the implementation and testing of a program to detect and report failure of an analog device. The extended AND/OR graphs are easily converted into Prolog programs. Solution trees obtained by searching the AND/OR graph are used to generate tests.
ACKNOWLEDGMENT

This research is supported by the Ministry of Science and Technology (MOST) of Taiwan under grant contracts 104-2221-E-027-007 and 105-2221-E-027-076.

REFERENCES
