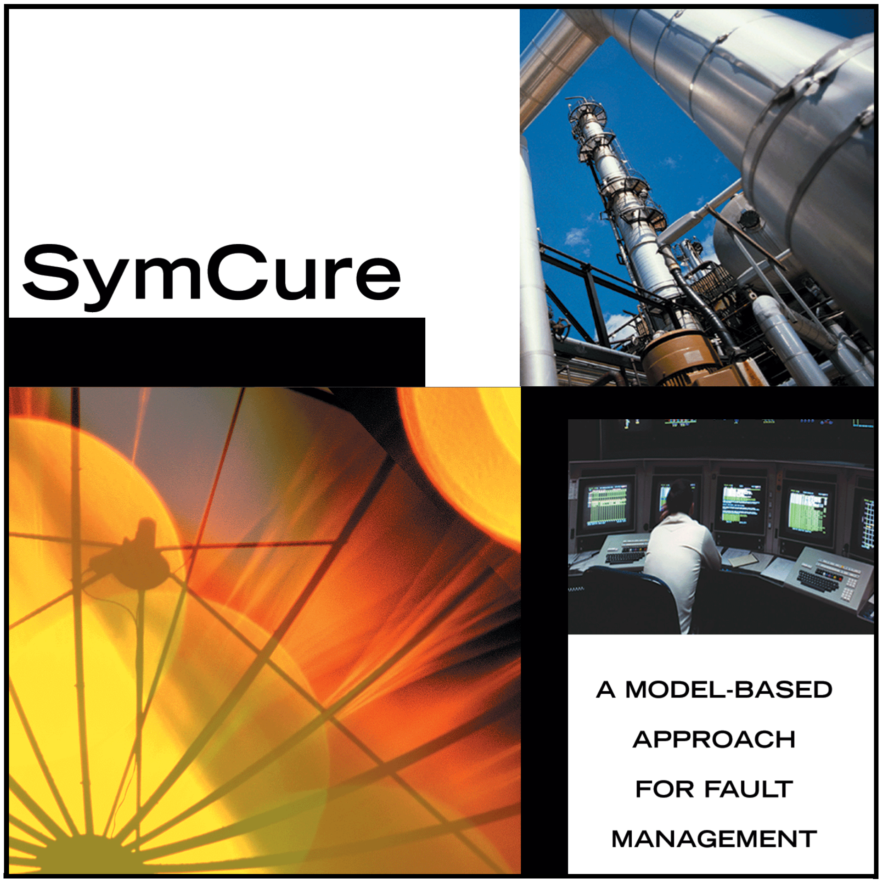
SymCure

User's Guide

Version 5.1 Rev. 0



SymCure User's Guide, Version 5.1 Rev. 0
July 2018

The information in this publication is subject to change without notice and does not represent a commitment by Gensym Corporation.

Although this software has been extensively tested, Gensym cannot guarantee error-free performance in all applications. Accordingly, use of the software is at the customer's sole risk.

Gensym Corporation
52 Second Avenue
Burlington, MA 01803 USA
Telephone: (781) 265-7100
Fax: (781) 265-7101

Part Number: DOC100-510

Contents

Preface

Describes this guide and the conventions that it uses.

About this Guide

SymCure is a development and deployment environment for building and implementing fault management applications that automate real-time fault isolation, testing, repair, and availability management tasks of large-scale operations.

This guide describes:

- SymCure's features and benefits, terms and concepts, and how to build a SymCure model, in general.
- How to begin using SymCure.
- SymCure's modeling language, which provides a wide range of event logic for representing causal diagrams.
- How to build and compile generic fault models, which include generic events, event views, and external actions.
- How to run SymCure applications and to interact with alarms and root causes through browsers.
- How to debug specific fault models.

- How to use root cause episode management.
- Parameters that you can configure at startup to customize SymCure's default behavior.
- The SymCure Application Programmer's Interface (API), which provides programmatic access to SymCure.

Audience

This guide is written for developers who want to build fault management applications. Developers should have some familiarity with fault management and with the G2 environment, including the G2 Developers' Utilities, or with Optegrity or Integrity.

Conventions

This guide uses the following typographic conventions and conventions for defining system procedures.

Typographic

Convention Examples	Description
g2-window, g2-window-1, ws-top-level, sys-mod	User-defined and system-defined G2 class names, instance names, workspace names, and module names
history-keeping-spec, temperature	User-defined and system-defined G2 attribute names
true, 1.234, ok, "Burlington, MA"	G2 attribute values and values specified or viewed through dialogs
Main Menu > Start KB Workspace > New Object create subworkspace Start Procedure	G2 menu choices and button labels
conclude that the x of y ...	Text of G2 procedures, methods, functions, formulas, and expressions

Convention Examples	Description
new-argument	User-specified values in syntax descriptions
<u>*text-string*</u>	Return values of G2 procedures and methods in syntax descriptions
File Name, OK, Apply, Cancel, General, Edit Scroll Area	GUIDE and native dialog fields, button labels, tabs, and titles
File > Save Properties	GMS and native menu choices
workspace	Glossary terms
`c:\Program Files\Gensym\`	Windows pathnames
`/usr/gensym/g2/kbs`	UNIX pathnames
`spreadsh.kb`	File names
`g2 -kb top.kb`	Operating system commands

`public void main()`
`gsi_start`
	Java, C and all other external code

Note Syntax conventions are fully described in the *G2 Reference Manual*.

Procedure Signatures

A procedure signature is a complete syntactic summary of a procedure or method. A procedure signature shows values supplied by the user in *italics*, and the value (if any) returned by the procedure <u>*underlined*</u>. Each value is followed by its type:

g2-clone-and-transfer-objects
(*list*: class item-list, *to-workspace*: class kb-workspace,
delta-x: integer, *delta-y*: integer)
-> <u>*transferred-items*</u>: g2-list

Related Documentation

SymCure is designed to be used within the G2 environment, or within Optegrity or Integrity.

Optegrity

- *Optegrity Heater Tutorial*
- *Optegrity User's Guide*
- *SymCure User's Guide*

Integrity

- *Integrity User's Guide*
- *Integrity Utilities Guide*
- *SymCure User's Guide*

G2 Core Technology

- *G2 Bundle Release Notes*
- *Getting Started with G2 Tutorials*
- *G2 Reference Manual*
- *G2 Language Reference Card*
- *G2 Developer's Guide*
- *G2 System Procedures Reference Manual*
- *G2 System Procedures Reference Card*
- *G2 Class Reference Manual*
- *Telewindows User's Guide*
- *G2 Gateway Bridge Developer's Guide*

G2 Utilities

- *G2 ProTools User's Guide*
- *G2 Foundation Resources User's Guide*
- *G2 Menu System User's Guide*
- *G2 XL Spreadsheet User's Guide*
- *G2 Dynamic Displays User's Guide*
- *G2 Developer's Interface User's Guide*
- *G2 OnLine Documentation Developer's Guide*
- *G2 OnLine Documentation User's Guide*
- *G2 GUIDE User's Guide*
- *G2 GUIDE/UIL Procedures Reference Manual*

G2 Developers' Utilities

- *Business Process Management System User's Guide*
- *Business Rules Management System User's Guide*
- *G2 Reporting Engine User's Guide*
- *G2 Web User's Guide*
- *G2 Event and Data Processing User's Guide*
- *G2 Run-Time Library User's Guide*
- *G2 Event Manager User's Guide*
- *G2 Dialog Utility User's Guide*
- *G2 Data Source Manager User's Guide*
- *G2 Data Point Manager User's Guide*
- *G2 Engineering Unit Conversion User's Guide*
- *G2 Error Handling Foundation User's Guide*
- *G2 Relation Browser User's Guide*

Bridges and External Systems

- *G2 ActiveXLink User's Guide*
- *G2 CORBALink User's Guide*
- *G2 Database Bridge User's Guide*
- *G2-ODBC Bridge Release Notes*

- *G2-Oracle Bridge Release Notes*
- *G2-Sybase Bridge Release Notes*
- *G2 JMail Bridge User's Guide*
- *G2 Java Socket Manager User's Guide*
- *G2 JMSLink User's Guide*
- *G2-OPC Client Bridge User's Guide*
- *G2-PI Bridge User's Guide*
- *G2-SNMP Bridge User's Guide*
- *G2-HLA Bridge User's Guide*
- *G2 WebLink User's Guide*

G2 JavaLink

- *G2 JavaLink User's Guide*
- *G2 DownloadInterfaces User's Guide*
- *G2 Bean Builder User's Guide*

G2 Diagnostic Assistant

- *GDA User's Guide*
- *GDA Reference Manual*
- *GDA API Reference*

Customer Support Services

You can obtain help with this or any Gensym product from Gensym Customer Support. Help is available online, by telephone, by fax, and by email.

To obtain customer support online:

➔ Access G2 HelpLink at `www.gensym-support.com`.

You will be asked to log in to an existing account or create a new account if necessary. G2 HelpLink allows you to:

- Register your question with Customer Support by creating an Issue.
- Query, link to, and review existing issues.
- Share issues with other users in your group.
- Query for Bugs, Suggestions, and Resolutions.

To obtain customer support by telephone, fax, or email:

➔ Use the following numbers and addresses:

	Americas	Europe, Middle-East, Africa (EMEA)
Phone	(781) 265-7301	+31-71-5682622
Fax	(781) 265-7255	+31-71-5682621
Email	service@gensym.com	service-ema@gensym.com

Introduction to SymCure

Provides an overview of fault management, describes SymCure's features and benefits, defines concepts and terms, describes event propagation logic, provides guidelines for building a generic fault model, and describes the architecture of a SymCure application.

What is SymCure?

SymCure is a development and deployment environment for building and implementing fault management applications that automate real-time fault isolation, testing, repair, and availability management tasks of large-scale operations.

Fault Management

Fault management plays a vital role across a broad spectrum of commercial and industrial applications, ranging from service level management and telecommunications network management in the Information Technology (IT) world, to abnormal condition management in manufacturing, chemical, oil and gas industries. The size and complexity of these applications often necessitates automated expert-system support for fault management. A small number of root cause problems in IT communication networks often result in a large number of messages and alarms that human operators cannot handle in real time. Failure to identify and repair the root cause problems results in increased system downtime and poor service levels. Abnormal conditions in manufacturing and processing plants may result in unplanned shutdowns, equipment damage, safety hazards, reduced productivity, and poor product quality.

Fault management across these industries shares some common goals, such as improving application availability and utilization, reducing operator overload, and minimizing operation costs. To achieve these goals, it is necessary to develop fault management tools with the following capabilities:

- *Symptom monitoring*. Symptoms are manifestations of underlying root causes and must be monitored to detect the occurrence of problems as soon as they happen.
- *Diagnosis.* Diagnosis identifies the root causes of known symptoms. Diagnosis is also often referred to as fault isolation. Some studies have shown that 80% of the fault management effort is spent in identifying root causes after the manifestation of symptoms.
- *Correlation*. Correlation is the process of recognizing and organizing groups of events that are causally related to each other for diagnostic inference and presentation to system operators. Usually such events share one or more root causes.
- *Prediction*. Early prediction of the impacts of underlying root causes before the effects are manifested is critical for proactive maintenance, safety, and optimal system utilization.

- *Testing*. In large systems, it is impractical and sometimes impossible to monitor every variable. Instead, key observable variables are monitored to generate symptom events. Diagnostic inference typically identifies a set of suspected root causes. Additional variables can then be examined by running associated tests to complete the diagnosis process.
- *Automated recovery*. Identifying and automating recovery procedures allows for growth in equipment, processes, and services, without increasing the supervisory burden on system operators.
- *Notification*. Operators must be notified of the presence of root causes and their potential impacts. Raw alarms, which can overload an operator with redundant information, must be replaced with concise diagnostic summaries of root causes and their impacts.
- *Postmortem*. Information from the diagnostic problem solving is fed back to the fault management system for historic record keeping and proactive fault management in the future.

Features

SymCure addresses a number of fault management functions, including diagnosis, correlation, prediction, testing, automated recovery, and notification. It provides a powerful object-oriented, model-based framework to specify diagnosis knowledge in the form of a persistent, generic (class-based), graphical, fault model library. It performs diagnosis and prediction by combining the fault models with specific domain information and incoming events at run time. It detects and resolves multiple system failures, and notifies the results of its diagnostic reasoning to external systems by using messages and other suitable means. SymCure's methodology is domain independent. It has been used for fault management in diverse applications across different industries, including abnormal condition management for heaters and service management for enterprise wide software systems.

SymCure provides the following capabilities:

- Automates fault and availability management of operations in domains as diverse as communications networks, enterprise-wide software applications, and manufacturing process plants.
- Performs online event correlation and interactive diagnosis to address the full life cycle of problem identification, based on symptoms, root-cause analysis, diagnostic testing, fault isolation, and recovery.
- Provides a powerful model-based framework consisting of generic, class-based fault models, which are tied to an object-oriented domain representation and scalable algorithms.
- Understands the complex relationships between each object, process, and event.

- Anticipates and diagnoses problems, based on this understanding.
- Requires minimal on-site customization for practical deployment. SymCure automatically accounts for configuration changes in equipment, topology, or operating modes in managed operations. Minimal customization eliminates the need for expensive reconfigurations of the fault management applications, enabling application developers to build reusable solutions that implement fault and availability management capabilities quickly and reliably.
- Accepts events and data from external sources.
- Provides a graphical language for automating labor- or reasoning-intensive tasks, such as root cause analysis, testing, fault mitigation, response, and recovery.
- Performs impact analysis to predict the effects of problems, and to measure the potential business impacts, for example, the impact on service level agreements in the networking industry or the impact of shutdowns in the manufacturing industry. SymCure can perform offline "what-if" simulation of failures to rapidly identify any potentially harmful effects of suspected root causes in a system.
- Guides operators through testing and recovery.
- Provides built-in configurable message browsers for system operators and developers.

Benefits

The depth and breadth of SymCure's causal reasoning capability; its ability to automate time-consuming, labor-intensive, and reasoning-intensive fault management tasks; and its ability to factor in the business impact of each event make SymCure the most powerful solution for managing complex fault management applications. SymCure's major benefits include:

- Rapid diagnosis and response to problems.
- Increased system availability.
- Optimization of personnel and system resources.
- Comprehensive impact analyses for more accurate contingency planning and other system-related business modeling.
- Improved service and equipment availability.

Terms and Concepts

Term or Concept	Definition
SymCure	The name of the product, which addresses the full life cycle of root cause diagnosis, including monitoring, fault isolation, impact analysis, diagnostic testing, and recovery from failure. The name is derived from Symptom Cure.
correlation	The process of recognizing that a group of events are causally related.
diagnosis	The process of identifying the root causes of a group of correlated events.
event	A logical assertion about a domain object, which is used to indicate the presence or absence of a problem. An event can represent a symptom, that is, an effect, or it can represent an underlying failure, that is, a root cause. Events can be observable or they can be introduced in the system purely for convenience in modeling to represent effects that cannot be seen directly but are required for fault propagation and root-cause analysis. For typical diagnostic management problems, SymCure must respond to an event with one or more of the following actions: • Investigate it in further detail by looking for its causes. • Predict its impact. • Report it. • Initiate one or more tests to obtain a true or false value for the event. • Initiate recovery actions to “repair” the target object of the event.
test	Events can be asynchronous, which means they can arrive without notice at any time. It is, however, possible to request a value for an event. Typically, the value of an event is determined by executing a test that yields its value. A test can be arbitrarily complex. After a test is initiated, it can return a value for the event asynchronously.

Term or Concept	Definition
root cause	An event that is an underlying cause for manifested alarms and other events. A root cause cannot itself be caused by any other event.
alarm	An event that is a manifestation of an underlying root cause, which is considered to be important enough to notify operators.
upstream propagation	The propagation of event values from effects to their root causes. Upstream propagation is required for root cause analysis, that is, fault isolation.
downstream propagation	The propagation of event values from causes to their effects. Downstream propagation is required for impact prediction.
context-dependent propagation	Event propagation that depends on the state of the domain, for example, propagation through a valve or a switch might depend on its position.
generic fault model	Generic causal relations among generic events defined over a set of classes. Events defined on different domain objects propagate to each other when the domain objects are connected or related in some way. A generic fault model for a class of domain objects defines the propagation of failures within its instances and to domain objects of other classes via generic domain relationships. Such fault models constitute a generic "library", independent of any specific collection of domain objects and relationships actually present at any particular site. Generic fault models are created by domain experts at development time.
domain map	An object-oriented model of a managed system, which can represent physical equipment and abstract entities, such as sensors, controllers, services, and software applications. The domain map includes domain object classes and their instances representing the managed entities, their connectivity, containment, and other relationships. The domain map is an input to SymCure.

Term or Concept	Definition
specific fault model	Describes the propagation of events within and across specific domain objects. SymCure constructs a specific fault model at run time, starting from any incoming event, by combining the generic fault models and the domain map of the managed system.
mutually exclusive events	Set of events defined on the same target object such that only one of them can be true at any given time. In other words, if any event in a set of mutually exclusive events is true, all the other events in that set must be false. It is possible for all the events to be false at the same time.

Event Propagation

An event can have the following values:

- True—The event is known to have occurred.
- False—The event is known not to have occurred.
- Unknown—It is not known whether the event has occurred.
- Suspect—It is suspected that the event may be true.

The status of an event indicates the justification for the event's value. The status can be:

- Specified—The value of the event is observed.
- Upstream inferred—The value of the event is inferred from one of its effects.
- Downstream inferred—The value of the event is inferred from one of its causes.

SymCure uses **event propagation** to compute the value and status of an event during diagnosis and impact analysis. Propagation takes place in two directions:

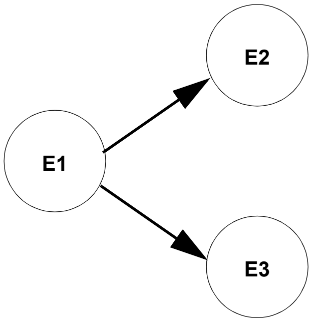
- **Upstream propagation** refers to the inference process that determines the values of upstream events from their downstream effects. SymCure uses upstream propagation to diagnose root causes from observed symptoms.
- **Downstream propagation** is the inference process that predicts the values of downstream events from their upstream causes. SymCure uses downstream propagation to predict effects and invoke downstream tests.

Causal Models

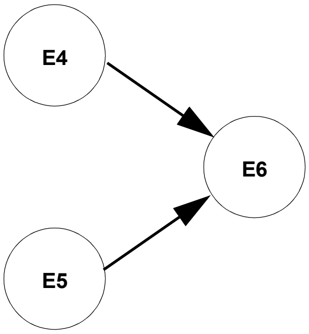
A fault model describes the relationship between events in the form of a **causal directed graph** or **causal model**. Two events X and Y are causally related when there is a directed edge between them; when the edge is directed from X to Y, we say that X causes Y. A directed path is a collection of one or more edges. When a directed path leads from X to Y, then X is upstream of Y, and Y is downstream of X.

Note SymCure's module name is `cdg`, which stands for causal directed graph.

The following figure illustrates the most basic forms of event propagation for the simplest kind of event. In this causal directed graph, E1, E2, and E3 represent events. The edges between them specify that E1 *causes* E2 and E3. If E2 (or E3) becomes true, then SymCure propagates a value of true to E1, then concludes that E3 (or E2) is also true. In general, if E1 is true, then both E2 and E3 must be true.



In this causal directed graph, E4 or E5 *cause* E6. If either E4 or E5 is true, SymCure concludes that E6 is true. If E6 becomes true, SymCure concludes that at least one of E4 and E5 must be true, or that both are suspect. If E6 becomes false, both E4 and E5 must be false.



In addition to the simple event propagation logic described in these diagrams, SymCure supports other forms of event propagation logic that allow you to describe complex relationships between events.

Guidelines for Building a SymCure Application

Building a SymCure application consists of the following high-level steps:

- Obtain prerequisite information about your domain.
- Develop the domain map.
- Create the generic fault models.
- Perform debugging and analysis of the generic fault models.
- Configure the SymCure application.

The following sections explain these steps in more detail.

Prerequisite Information

Prior to developing your SymCure application, you need to identify essential information about the domain, including:

- Major components in the domain.
- The crucial problems faced in the domain and the specific scenarios that result.
- How these problems are detected and how their unique root cause(s) are currently determined.
- Corrective actions taken in response to the scenarios.
- Interfaces that might be needed between SymCure and external systems that send in events.

Identify the information you need by identifying and interviewing key resources that are knowledgeable about the domain. Examples of key resources include operations staff, engineers, systems administrators, facilities support personnel, and managers concerned about specific problems relating to the domain and its operation.

Creating Domain Maps

The next step in the design of a SymCure application is to develop the specific domain model, which involves these steps:

1. Obtain a thorough understanding of the domain, including its functionality, operation, and current management.
2. Use this information to create class and relation definitions, which reflect the desired level of detail to be used in the domain map.
3. Create a domain model, which requires:
   a. Creating the object definitions and relationships to be represented in the domain map.
   b. Constructing the domain map.
4. Obtain external data through an interface.

Creating Class Definitions and Relations

You create the domain map by creating instances of class definitions that inherit from the grtl-domain-object class. This class provides all the mechanisms required for creating intelligent domain objects that know how to respond to SymCure events.

To determine which managed object classes to represent in the domain model, you need to:

- Identify the major entities of the domain you are modeling.
- Determine the level of detail for representing the domain entities.
- Identify the relationships between domain entities that are important for fault propagation.

For example, in a manufacturing environment, the domain model might represent a manufacturing process, where the domain objects might include heaters, pumps, valves, and sensors. Similarly, in a network environment, the domain model might represent a company's computer network, where the domain objects might include hardware, network infrastructure, operating system software, and other software applications running on the system.

As a guideline for determining the level of detail, consider representing only those domain entities with events of interest. This practice limits the representation of domain entities to those that significantly participate in the fault analysis scenarios.

Creating the Domain Map

To create a domain map, first you must create class definitions and relationship definitions between domain object classes that describe your process.

Once you have defined the domain object classes and relationships, you construct the domain map by creating instances of the class definitions and by relating those instances through containment, connectivity, and other G2 relations.

Obtaining External Data

Finally, you must obtain data from the external system that the domain map represents. You obtain external data by configuring interfaces that communicate with external systems via a bridge.

Creating Generic Fault Models

Each domain object class defined in the domain model should have an associated **generic fault model**, which describes the failure propagation within an object of that class and to other related objects. To define a generic fault model for a domain object class, you need to identify:

- Root cause events.
- Events that are caused by the root causes.
- Causal relations between pairs of events.

SymCure typically responds to symptoms of problems by initiating:

- Root cause analysis.
- Tests for suspected root causes.
- Repair actions on detected root causes.

Tests

You might need to define test objects that are associated with certain events. These tests can include observations that the operator performs on request, and a set of actions and analysis of the measured data.

Repair Actions

You can associate repair actions with events that SymCure can invoke to recover from the occurrence of the event.

Debugging and Analyzing Generic Fault Models

As you build generic fault models for a SymCure application, you must compile the models to eliminate any modeling and configuration errors.

For detailed descriptions of these generic fault model debugging and analysis features, see Creating Generic Fault Models.

Configuring the SymCure Application

User-defined configurations for a SymCure application include parameters that:

- Control the size and rate for building the specific fault model at run time.
- Influence when the diagnosis algorithm terminates.
- Determine when older diagnostic problems are deleted from memory.
- Determine how SymCure handles events that have not been changed over long periods of time.
- Specify user-defined methods to be executed at various points of SymCure processing.

For more information, see Configuring SymCure Applications.

Testing the SymCure Application

To ensure that your SymCure application and the related generic fault models reflect the correct fault propagation behavior, they must be reviewed, tested, and verified against various fault occurrence and diagnosis scenarios.

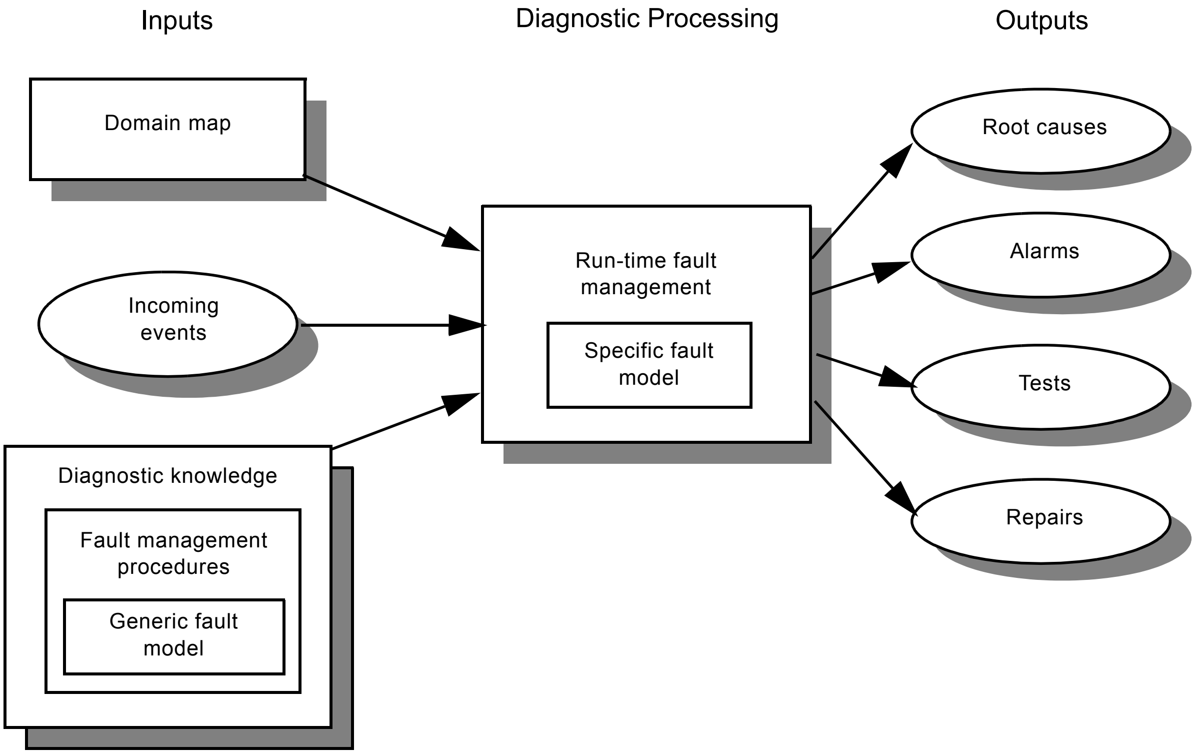
To help debug your SymCure application, you can use the SymCure debugger to build a specific fault model and step through the specific events.

For information related to testing your generic fault models, see Running SymCure Applications.

For information on run-time debugging, see Debugging SymCure Fault Models.

Architecture of a SymCure Application

This diagram shows the architecture of a SymCure application:



Diagnostic knowledge consists of generic fault models that allow SymCure to perform fault diagnosis. Fault management procedures built around generic fault models specify the procedural knowledge required for testing diagnostic processes and running corrective procedures to recover from failures. Incoming specific events trigger SymCure's run time fault management and diagnostic reasoning. During diagnosis, SymCure performs the following tasks:

- Builds a specific fault model, using the generic fault model and domain map specification for the managed system as inputs.
- Correlates incoming events by propagating their values across the specific fault model.
- Identifies root causes and predicts alarms.
- Integrates diagnostic knowledge and intelligent testing by selecting and executing appropriate tests to resolve suspected root causes.
- When SymCure has determined the root cause(s), it runs repair actions to recover from known faults or it presents such procedures to the operator for manual execution.

SymCure provides a run-time diagnostic console that enables you to view the current status of diagnosis and run user-initiated tests and repair actions. For detailed information on run-time processing, see Running SymCure Applications.

Getting Started

Describes the steps for setting up, creating, running, and configuring a SymCure application, and describes how to load the SymCure demos.

Introduction

SymCure is distributed as a set of modularized KB files, with `symcure.kb` as the top-level module. Typically, you create and run SymCure applications within the Optegrity or Integrity environment. You can also create and run SymCure applications within any G2 application, independent of Optegrity and Integrity.

Before you can use SymCure for diagnosis, you must create a domain map, which defines the specific domain objects that SymCure uses at run-time when it constructs specific fault models. You must also set up interfaces to provide external data from the managed system to G2 and SymCure. You can then use the external data to generate SymCure events.

If you are running SymCure within a stand-alone G2 environment, you can use GSI interfaces to import external data, G2 class definitions to create domain object classes, and G2 procedures to monitor data and generate events. You can use the SymCure Application Programmer's Interface (API) to generate SymCure events and obtain diagnosis information at run time.

If you are running SymCure within the Optegrity and Integrity environments, you can use the built-in modules that these products provide to facilitate building domain maps, managing data sources, and monitoring data and generating events.

This chapter assumes you are familiar with the environment in which you will run SymCure: G2, Optegrity, or Integrity.

SymCure is distributed as part of the G2 Developers' Utilities (`g2i`), which is part of the G2 Bundle.

SymCure is packaged with two simple demos in a single KB, `cdg-modguide.kb`, which you can run to become familiar with SymCure.

Creating a SymCure Application

We recommend that you create a new top-level module that requires the top-level SymCure module.

To create a SymCure application:

1. Start the environment in which you plan to run SymCure, typically, Optegrity or Integrity.

   Optegrity and Integrity provide all the SymCure KBs that you need to build a SymCure application.

   If you are running in a pure G2 environment, merge the KB named `g2i\kbs\symcure.kb` into your application and make it a required module of your module.

2. Create a new top-level module with a prefix that you will use to identify all items in your application, for example, `myapp`.

3. Configure the new module to require symcure.

   For details, see the *G2 Reference Manual*.

4. Save your application to a file name that matches the top-level module name, for example, `myapp.kb`.

Setting up the Application

This section describes, at a very high level, how to set up your application so it can use SymCure for diagnostic reasoning. The techniques you use depend on the environment in which you are running SymCure.

To set up the application

1. Create a domain map.

   A domain map consists of domain objects that are subclasses of the grtl-domain-object class. If you are running SymCure within an Optegrity environment, the top-level domain object class is opt-domain-object-with-key.

   You can create your own domain object classes by subclassing this class, or you can use the built-in classes that your environment provides for this purpose. When using SymCure within Integrity, you can also import domain maps.

   For background information, see Creating Domain Maps.

2. Create one or more interfaces to obtain external data.

   If you are running within a G2 environment, you create your own gsi-interface objects to import data through a bridge.

   If you are running Optegrity, you can use the Gensym Data Point Management (GDPM) module to manage external data sources. This module automatically configures external datapoints from CSV files and links those datapoints with internal datapoints within the domain map.

3. Set up a system that monitors external data and generates SymCure events, based on the data.

   If you are running within a stand-alone G2 environment, you can use procedures and methods to monitor external data and generate SymCure events. To generate events programmatically, you can use the SymCure API procedures, which are described in Application Programmer's Interface.

   If you are running Optegrity, you can use the Gensym Event and Data Processing (GEDP) module to monitor intelligent domain object datapoints, which obtain their values from the monitored system, and generate events, based on the data. GEDP provides a graphical language for creating event-detection diagrams.

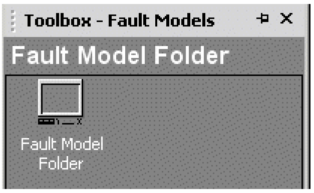
Creating Generic Fault Models

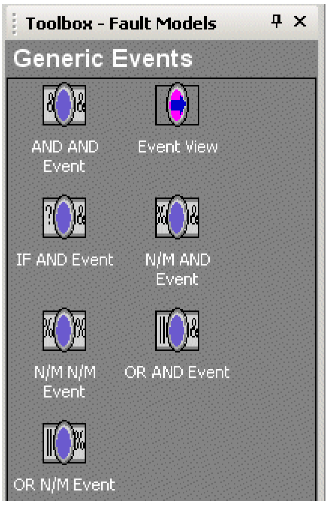
Once you have set up the application, you create generic fault models for domain object classes. These fault models describe causal relationships between events. The models allow SymCure to diagnose faults, based on observed symptoms, and to predict effects. The fault models also describe external actions that SymCure can execute to test its diagnoses and to repair and recover from faults.

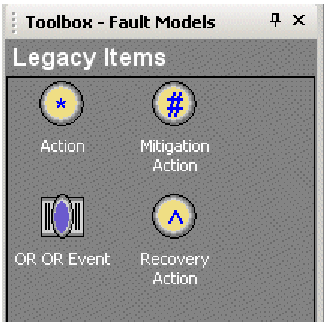
SymCure provides a graphical modeling language to create generic fault models. You clone generic fault model events from a palette and connect them to form a causal fault model. You create causal fault models within containers called Fault Model folders.

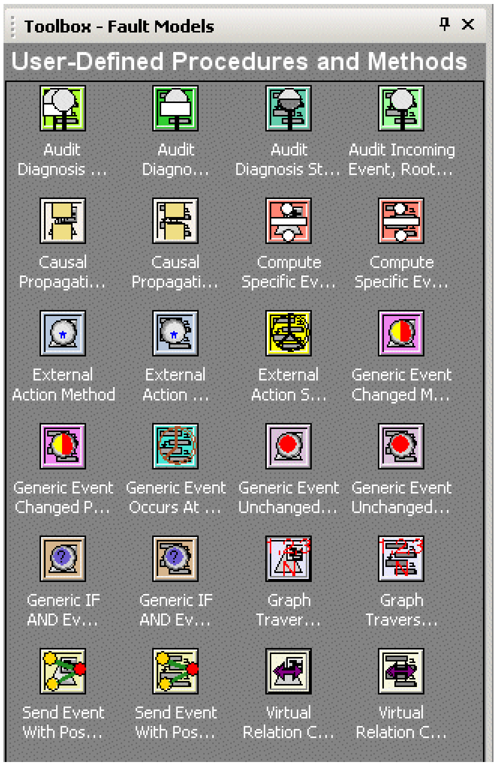
To display the SymCure palettes:

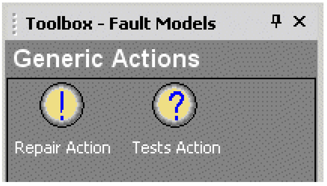
➔ Choose View > Toolbox - Fault Models to display these palettes:











For a description of the SymCure modeling language for representing causal relationships between events, see SymCure Modeling Language.

For details on how to create generic fault models, using generic events and generic actions, see Creating Generic Fault Models.

Running the Application

Once you have created generic fault models for your domain object classes, you can run a diagnostic application. Running the application involves monitoring external data and generating fault model events. When an event is generated, SymCure creates a specific fault model for specific domain objects, which is derived from generic fault models defined for domain object classes. The specific fault model consists of specific events, including alarms and root causes, and specific actions to run tests and repair faults. You can interact with alarms, root causes, test actions, and repair actions through various browsers.

SymCure also provides a mechanism for simulating events, which allows you to test generic fault model logic. For details, see Running SymCure Applications.

Configuring Initialization Parameters

You can configure a number of initialization parameters that affect the behavior of your application. These include parameters that affect the behavior of SymCure's diagnosis algorithm and user-defined procedures that SymCure can run at various points during diagnosis for auditing purposes.

For details, see Configuring SymCure Applications.

Running the SymCure Demos

SymCure is packaged with two demos, which are available in the `cdg-modguide.kb` module:

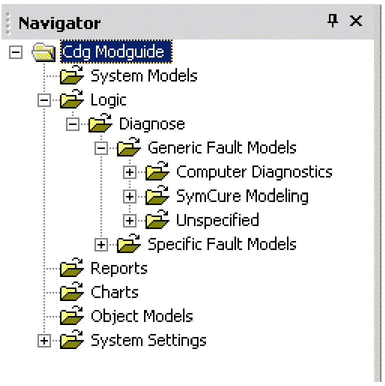
- SymCure application diagnostics — Diagnoses faults when building a SymCure application.
- Computer diagnostics — Diagnoses faults in a computer network.

For more information on running the SymCure application diagnostics demo, see Running SymCure Applications.

To run the SymCure demos:

1. Start G2.
2. Load `cdg-modguide.kb`, which is located in the `g2i\examples` directory of your installation directory.

3 Choose View > Navigator to view the fault models in the demo application:



The demo provides two generic fault models, one for computer diagnostics and the other for SymCure application diagnostics. The generic fault models describe causal relationships between events and external actions that the model can run during diagnosis. SymCure creates specific fault models for domain objects at run time to diagnose faults.

You can also access items in the demo through the cdg-modguide-demo top-level workspace, which also provides access to domain objects and their associated class definitions.

This document frequently uses examples from the SymCure application diagnostics demo.

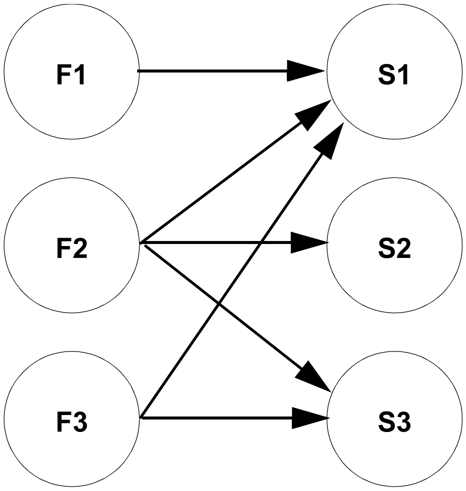
SymCure Modeling Language

Describes the SymCure modeling language, which provides graphical generic events that are used to create fault models in the form of causal directed graphs.

Introduction

SymCure has been used for diagnosing a wide variety of real world problems across different industries from telecommunication and satellite networks, to petroleum refineries. Real world applications often require modeling sophisticated interactions amongst events and have motivated the development of SymCure's powerful causal modeling framework.

To understand the sophisticated interactions amongst causal relationships between discrete events, consider the following causal directed graph, which is composed of root causes F1, F2, and F3, and their corresponding effects S1, S2, and S3.



Consider the following examples.

Example 1. Assume that F2 is true. Consider the set of causal relations at the "output" of event F2:

F2 -> S1

F2 -> S2

F2 -> S3

If these relations are "equal" in every respect, propagation occurs through all three relations identically, and all three of S1, S2, and S3 must occur. We refer to this type of causal relationship as *AND logic at the output* of F2.

In the real world, propagation might not occur uniformly across all pathways of causal interaction. In the presence of propagation delays, threshold errors, and noisy sensors, a symptom that is downstream of a root cause is sometimes incorrectly reported as false when in fact, it should be reported as true, or vice versa. Thus, it is possible that not all of S1, S2, and S3 will be manifested when F2 is true. This situation motivates an alternative interpretation of causal propagation that supports *OR logic at the output* of F2.

Finally, in the real world, the value of an event might depend on the state of the system rather than on any of the causes of the event. This situation is known as *context dependent propagation logic*, referred to as IF logic. An example appears later in this section. Thus, it is possible that none of S1, S2, and S3 will be manifested unless certain external conditions are met.

Example 2. Assume that S1 is true. Consider the set of causal relations at the "input" of event S1:

F1 -> S1

F2 -> S1

F3 -> S1

Suppose there are dependencies among the causal relations. For example, F1 and F2 and F3 might have to be true for S1 to be true. Such a dependency gives rise to the notion of *AND logic at the input* of an event, where the event is true if and only if all of its inputs are true.

On the other hand, causal relations can also be independent of each other, so the occurrence of any one of F1 or F2 or F3 may cause S1. This interpretation is known as *OR logic at the input* of an event.

In real world scenarios, there can also be partial dependencies among causal relations, for example, 50% or more of the causes of S1 must become true for S1 to be true. We characterize this type of causal relationship as *fractional event propagation logic*, referred to as N/M logic.

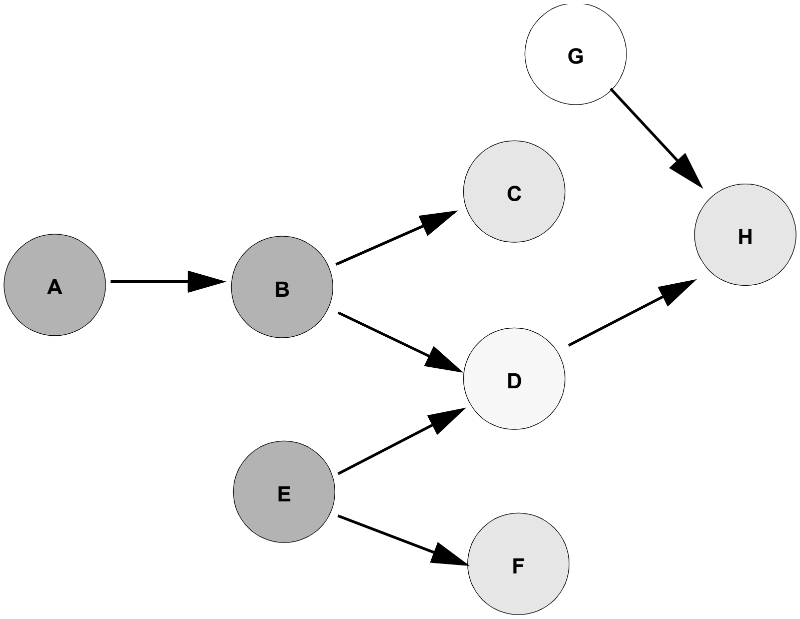
Introduction to SymCure Event Propagation

This section describes some of the basic concepts for event propagation that are crucial to understanding the behaviors of SymCure events.

At a high level, SymCure performs event propagation in a fault model, as follows:

```
for any incoming event e do
    propagate upstream to causes = all causes of e;
    propagate downstream to effects = all effects of causes + e;
end for
```

The following example illustrates the high level algorithm:



In the figure above, suppose that the incoming event is D. In this example:

causes = {A, B, E}

effects = {C, F, H}

Note that the event G is not affected in response to incoming event D.

The values computed for the events depend on SymCure's event propagation logic, which is described in detail below.

SymCure Event Logic

Whenever an event *e* gets a new value, SymCure must propagate that value upstream and downstream to the causes and effects of the event. This requires the following computations:

- Based on its propagation logic, its new value and its previous state, *e* must compute what values to propose for its inputs and outputs.
- Each cause and effect of *e* that receives the proposed values from *e* must decide whether or not it should accept these changes.

Let's look at how an event can get a new value. The value of an event can be obtained by observation. In this document, the observed value for an event is frequently referred to as an incoming value or a specified value. The following table shows how the value of any SymCure event (independent of its propagation logic) is affected by an observation.

Old Value	**Incoming Value**		
	true	**false**	**suspect**
true	true	false	true
false	true	false	suspect
suspect	true	false	suspect
unknown	true	false	suspect

Typically, an observation provides a value of true or false for the event. When the value of an event is attained by observation, SymCure associates that status "specified" with the event. SymCure also allows you to specify that an event is suspect. This is not strictly an observation, but may be useful for diagnostic reasoning. SymCure ignores values that are specified as unknown because they do not provide any meaningful information.

A triggering event is any event that causes the propagation of a value to another event. The value of an event may also be changed by upstream or downstream propagation as a consequence of a change in a triggering event.

A fundamental principle of causal modeling is that any event, unless it is a root cause, is a consequence of its causes. This is valid regardless of whether the event is observed or inferred. A change in the value of a downstream event can only be explained by changes in one or more of its causes, which can in turn, be traced back to one or more root causes. This impacts SymCure's propagation logic in the following ways:

- In general, when the triggering event is downstream of an event *e*, *e*'s value is changed by the triggering event. This is captured by AND logic at the output of the event.
- When the triggering event is upstream of an event *e*, *e*'s value is computed by analyzing all events that could cause *e*, as specified by the input logic for *e*, that is, OR, AND, N/M, IF.

You can use the following rules of thumb to compute proposed values for the causes and effects of any event. Keep in mind that these are only guidelines; the computation of event values depends on a number of factors, such as its propagation logic, the topology of the fault model in its neighborhood, and the nature of neighboring events, which may override the rules of thumb.

- If an event has only one unknown cause (input), then if the event is true, we can conclude that its sole cause is also true. (Events with OR and N/M logic and multiple effects may be exceptions to this rule.)

- Whenever possible, SymCure tries to reconcile new information with existing knowledge. Thus, it ignores unknown values in favor of what is already known about an event. However, existing values are overridden when they are not consistent with new information.
- In general, SymCure processes an event only when the value of the event changes. The only time that SymCure will process an event whose value has not changed is if the status of the event changes to specified.

Generic Events

SymCure defines generic event objects that are equipped with powerful causal propagation logic, which facilitates modeling a wide range of diagnostic situations. SymCure defines seven different types of generic events, which use different types of propagation logic to compute their values.

The icons identify the propagation logic that the event uses at the input and output of the event, as follows:

- Parallel lines (||) implies OR logic.
- Ampersand (&) implies AND logic.
- Percent (%) implies a fraction or N/M (read as "N out of M").
- Question mark (?) implies state or context-dependent logic, using IF logic.

Events share common behavior depending on their input and output logic, that is, N/M-AND, OR-AND, AND-AND, and IF-AND events all use AND logic at the output, while the N/M-AND and N/M-OR use N/M logic at their input.

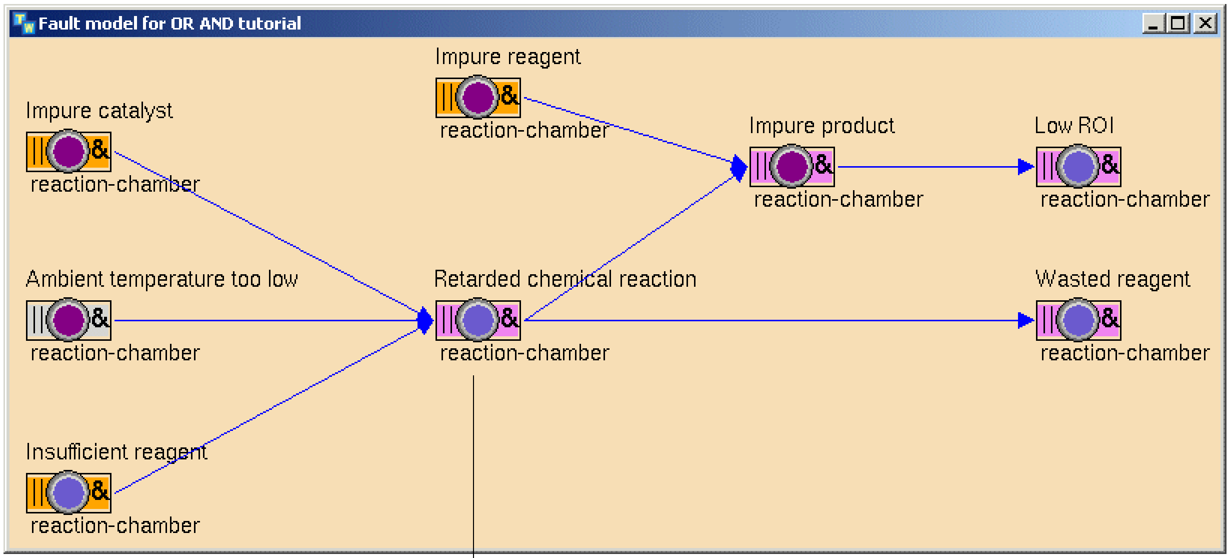
Note Currently, SymCure provides all types of events to avoid harming existing applications; however future versions might provide just the N/M-N/M event and the IF-AND event to build models of arbitrary complexity.

OR-AND Event

An OR-AND event uses OR logic at its input and AND logic at its output. The value of an OR-AND event is true if one or more of its causes are true.

Note The OR-AND event is the most common event for creating causal models.

In the following example, “Retarded chemical reaction” is caused by one or more of the upstream events:



OR-AND event

Upstream Propagation

The following table specifies the logic for computing the value of an OR-AND event when a triggering event downstream of the OR-AND event changes value:

When the value proposed by the trigger is...	Conclude that event is...
true	true
false	false
suspect	suspect
unknown	unknown

This table describes the rules that propose values for the inputs of an OR-AND event when the value of the OR-AND event changes during upstream propagation:

Old Value	**New Value**			
	true	**false**	**suspect**	**unknown**
true	If there is no true input, but there is a single unknown or suspect input, change it to true.	Change all inputs to false.	Change all true inputs to suspect.	Change all true and suspect inputs to unknown.
false	If there is only one input, change it to true, else change all inputs to suspect.	Change all inputs to false.	Change all inputs to suspect.	Change all inputs to unknown.
suspect	If there is only one input, change it to true, else if there is a single suspect input, change it to true and change all unknown inputs to suspect.	Change all inputs to false.	Change all unknown inputs to suspect.	Change all suspect inputs to unknown.
unknown	If there is a single unknown, false, or suspect input, change it to true, else change all unknown inputs to suspect.	Change all inputs to false.	Change all unknown inputs to suspect. If there are no unknown and no suspect inputs, then change all false inputs to suspect.	No change.

Downstream Propagation

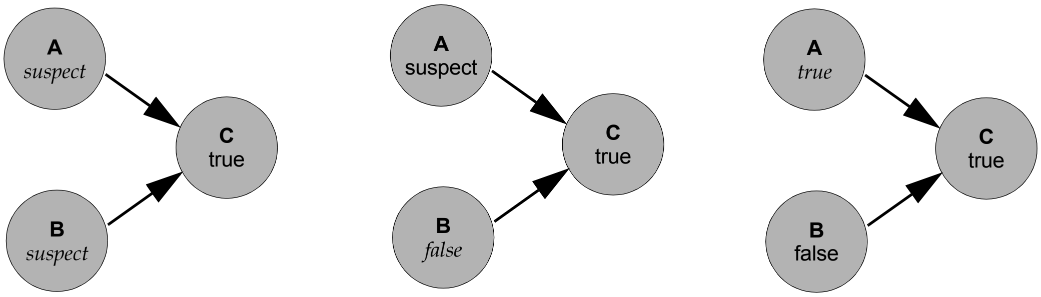
SymCure computes the value of an OR-AND event during downstream propagation when a triggering event upstream of the OR-AND event changes value.

In general, SymCure uses traditional OR logic for computing the value of the event, as follows:

> If there is at least one true input, conclude that the event is true,
> Else if there is at least one suspect input, conclude that the event is suspect,
> Else if there is at least one unknown input, conclude that the event is unknown,
> Else conclude that the event is false.

There is an exception to the general rule for computing the value of an event using OR logic during a downstream propagation.

The causal graph shown below illustrates the situation when an OR-AND event suspends the use of OR logic. Each event in the graph is an OR-AND event and is shown with a value. Event values that have changed most recently are shown in italics.

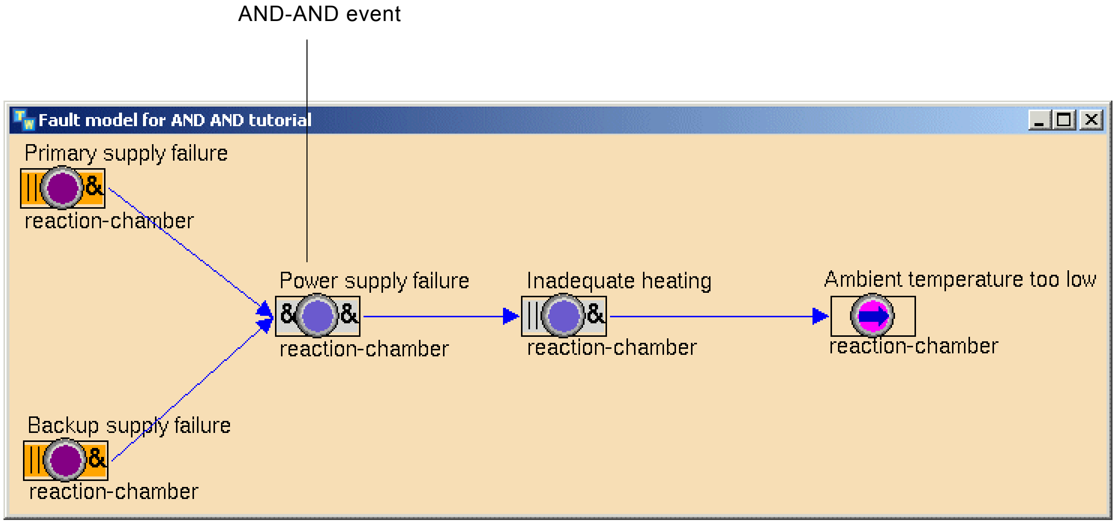


In this example, initially C becomes true, which causes A and B to be suspect. When B changes from suspect to false, A is the last remaining suspect event. Thus, for C to be true, A must be true. Hence, instead of making C suspect according to the OR logic rule above, SymCure sets A to be true.

AND-AND Event

An AND-AND event uses AND logic at its input and AND logic at its output. The value of an AND-AND event is true if all of its causes are true.

In this example, “Power supply failure” is caused by the conjunction of “Primary supply failure” and “Backup supply failure”:



Upstream Propagation

The following table specifies the logic for computing the value of an AND-AND event during upstream propagation (same as OR-AND event) when a triggering event downstream of the OR-AND event changes value:

When the value proposed by the trigger is...	Conclude that the event is...
true	true
false	false
suspect	suspect
unknown	unknown

This table shows proposed values for the inputs of an AND-AND event when the value of the AND-AND event changes during upstream propagation:

Old Value	New Value			
	true	false	suspect	unknown
true	Change all inputs to true.	If there are any unknown or suspect inputs, change them to false, else change all true inputs to suspect.	Change all inputs to suspect.	Change all inputs to suspect.
false	Change all inputs to true.	Change all inputs to false.	Change all false and unknown inputs to suspect.	Change all false inputs to unknown.
suspect	Change all inputs to true.	Change all suspect and unknown inputs to false.	Change all unknown inputs to suspect.	Change all inputs to unknown.
unknown	Change all inputs to true.	Change all inputs to false.	Change all unknown inputs to suspect.	No change.

Downstream Propagation

SymCure computes the value of an AND-AND event during downstream propagation when a triggering event upstream of the AND-AND event changes value.

SymCure uses traditional AND Logic for computing the value of the event as follows:

```
If every input is true, conclude that the event is true,
Else if there is at least one suspect input and the rest are true, conclude that
the event is suspect,
Else if there is at least one unknown input and the rest are suspect or true,
conclude that the event is unknown,
Else conclude that the event is false.
```

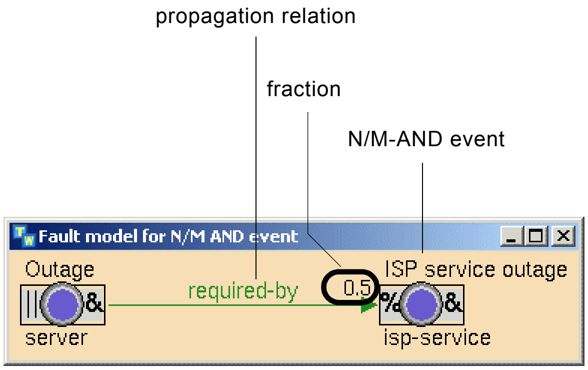
N/M-AND Event

SymCure provides N/M logic at the input of an event, which is a numerical generalization of OR logic at the input of an event. The value of an N/M event is true if at least N/M of its causes are true, where N/M is a real number ranging from 0.0 to 1.0. The N/M AND event uses AND logic at its output.

Consider the case of an IT service provider that uses a number of server objects to provide a service, for example, data communications and telephony, to its customers. Typically, individual servers can be shutdown without any significant impact on the service provided to customers; however, if the number of servers suffering outages exceeds a certain threshold, the quality of service starts to deteriorate.

Note The N/M-AND event is not a probabilistic event, because SymCure only calculates qualitative truth values (true, false, suspect, or unknown) for an event, not degrees of confidence.

In the following example, “ISP service outage” becomes true when at least 50% of the servers that are required-by an ISP service undergo “Outage”. The required-by relation describes the causal relationship between the ISP service that is experiencing “ISP service outage” and the servers it relies on that are undergoing “Outage”. The events in the fault model are generic; at run time, the domain representation provides SymCure with the number of specific servers required by a specific service and the number of specific “Outage” events that are true.



Upstream Propagation

The following table specifies the logic for computing the value of an N/M AND event during upstream propagation (same as OR AND event) when a triggering event downstream of the N/M AND event changes value:

When the value proposed by the trigger is...	Conclude that the event is...
true	true
false	false
suspect	suspect
unknown	unknown

The table below describes the rules that propose values for the inputs of an N/M-AND event when the value of the N/M-AND event changes during upstream propagation:

Old Value	**New Value**			
	true	**false**	**suspect**	**unknown**
true	Change all unknown inputs to suspect.	Change all true inputs to suspect.	Change all true inputs to suspect.	Change all true and suspect inputs to unknown.
false	If there only one input, change it to true, else change all inputs to suspect.	Change all true inputs to suspect.	Change all inputs to suspect.	Change all inputs to unknown.
suspect	If there only one input, change it to true, else change all unknown inputs to suspect.	Change all true inputs to suspect.	Change all unknown inputs to suspect.	Change all suspect inputs to unknown.
unknown	If there is a single unknown, false, or suspect input, change it to true, else change all unknown inputs to suspect.	Change all inputs to false.	Change all unknown inputs to suspect. If there are no unknown and no suspect inputs, then change all false inputs to suspect.	No change.

Downstream Propagation

The following rule describes the behavior of an N/M-AND event during downstream propagation, that is, when the triggering event is upstream of the N/M-AND event:

> If the number of causes that are true/total number of causes >= N/M then the event is true,
> Else if (the number of causes that are true + the number of causes that are suspect)/total number of causes >= N/M then the event is suspect,
> Else if (the number of events that are unknown + the number of events that are suspect + the number of events that are true)/total number of causes >= N/M then the event is unknown,
> Else the event is false.

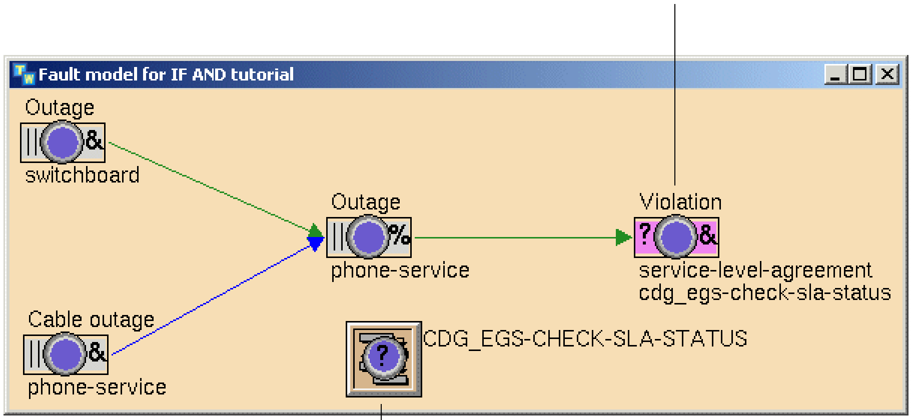
Note The N/M-AND does not provide any exceptions to this rule, unlike the exception case described for the OR-AND event.

IF-AND Event

In rare cases, the value of an event depends on the state of its target object, rather than on any of the causes of the event. SymCure provides context-dependent propagation in the form of an IF-AND event. You associate with an IF-AND event a user-defined procedure that dynamically determines the value of the event. The state-dependent procedure is invoked each time SymCure attempts to propagate a value to the event from its causes. The procedure returns a value for the IF-AND event by examining the state of its target object. The event uses AND logic at its output.

In the following example, “Outage” on the phone service is a potential cause for “Violation” on a related service-level-agreement. You represent “Violation” in the fault model to enable diagnosis of its causes whenever it occurs. However, you cannot conclude that “Violation” has occurred whenever there is a phone-service “Outage”. Violations of service level agreements might depend on a number of factors that are not captured by the fault model, such as the time of the day the outage occurs, the history of all outages in a corresponding period, and the duration of the present outage.

IF-AND event



state-dependent procedure

The only difference between this event and an OR-AND event is as follows:

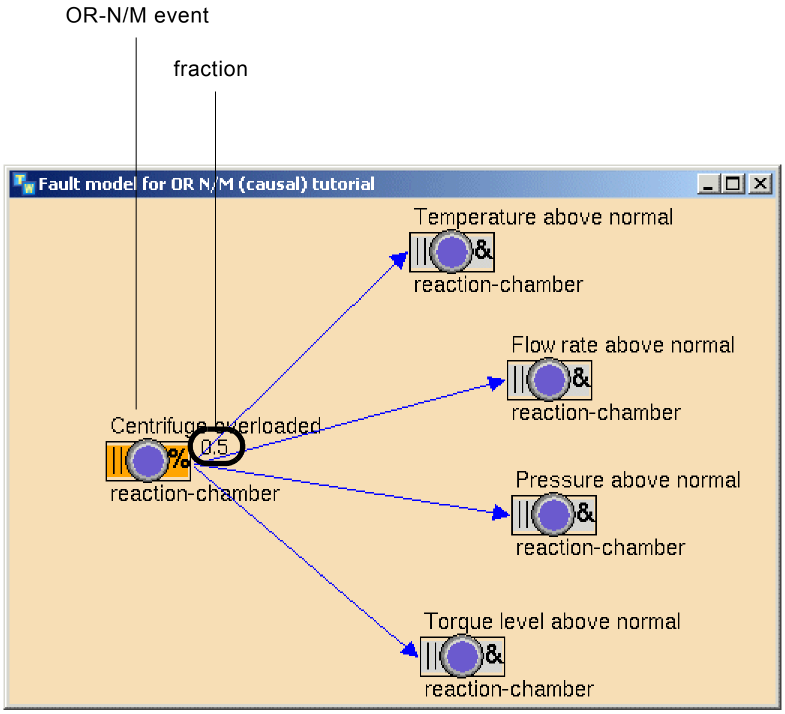
> SymCure does not define the behavior for concluding a value for the event from its inputs during downstream propagation. This behavior is defined by the user-defined procedure that will be associated with this event.

SymCure propagates to an IF-AND event whenever any of its upstream events gets a value, even if the value has not changed. This is required because an IF-AND event depends not only on its causal events, but on the state of the managed system. Triggering the event whenever an upstream event gets a new value ensures that the state-dependent procedure is run each time an event occurs, regardless of whether the value has changed.

OR-N/M Event

The OR-N/M event is a generalization of an OR-OR event. An OR-N/M event is true whenever there are at least N/M downstream effects that are true.

In the following example, the fraction defines the behavior of the event "Centrifuge overloaded". When the centrifuge of a reaction chamber is overloaded, it causes above-normal temperature, flow rate, pressure and torque levels. However, because of noise and sensor errors, the temperature, flow rate, pressure, and torque symptoms might or might not be manifested. In this model, as long as at least 50% of its symptoms are true, the event is believed to be true.



Upstream Propagation

SymCure computes the value of an OR-N/M event during upstream propagation when a triggering event downstream of the OR-N/M event changes value.

SymCure uses traditional OR Logic for computing the value of the event from its outputs independently of the value of the triggering event, as follows:

```
If the number of true outputs/total number of outputs >= N/M, conclude that the event is true,
Else if (the sum of true and suspect outputs)/total number of outputs >= N/M, conclude that the event is suspect,
Else if (the sum of true & suspect & unknown outputs)/total number of outputs >= N/M, conclude that the event is unknown,
Else conclude that the event is false.
```

The table for proposing values for the inputs of an OR-N/M event during upstream propagation is identical to that of the OR-AND event.

Downstream Propagation

SymCure computes the value of an OR-N/M event during downstream propagation when a triggering event upstream of the OR-N/M event changes value.

SymCure uses the same logic for computing the value of an OR-N/M event during downstream propagation that is does for an OR-AND event.

Consider the values propagated by an OR-N/M event to its effects when its value changes:

> If Fraction = 1.0, the OR-N/M event behaves like an OR-AND event.
>
> If Fraction = 0.0, the OR-N/M event behaves like an OR-OR event.
>
> If 0 < Fraction < 1.0, the behavior of the OR-N/M event is described by the table below.

Old Value	New Value			
	true	false	suspect	unknown
true	Change all unknown outputs to suspect.	Change all true outputs to suspect.	Change all true outputs to suspect.	Change all true outputs to unknown.
false	Change all outputs to suspect.	Change all unknown outputs to false.	Change all outputs to suspect.	Change all false outputs to unknown.
suspect	Change all unknown outputs to suspect.	No change.	Change all unknown outputs to suspect.	No change.
unknown	Change all outputs to suspect.	Change all unknown outputs to false.	Change all unknown outputs to suspect.	No change.

N/M-N/M Event

The N/M-N/M event is a generalization of all event types, which allows you to configure the input and output event logic. You can use this event in place of any of any of these events:

- OR-AND
- AND-AND
- N/M-AND
- OR-N/M
- OR-OR

The N/M-N/M event provides two fractions, the Input Fraction and Output Fraction, whose event logic is configured as follows:

- Input logic:
  - Input Fraction = 0.0 -> OR logic
  - 0.0 < Input Fraction < 1.0 -> N/M logic
  - Input Fraction = 1.0 -> AND logic
- Output logic:
  - Output Fraction = 0.0 -> OR logic
  - 0.0 < Output Fraction < 1.0 -> N/M logic
  - Output Fraction = 1.0 -> AND logic

OR logic at the input (output) implies that if the event is true, at least one of its causes (effects) must be true; it is possible for some causes (effects) to be false, while the event is true. When the event with OR logic at the input (output) becomes true, if there is a single cause (effect), that cause (effect) must also be true. If there are multiple causes (effects), at least one of them must be true; if SymCure cannot identify at least one cause (effect) that might be true or suspect, it treats all causes (effects) as suspect.

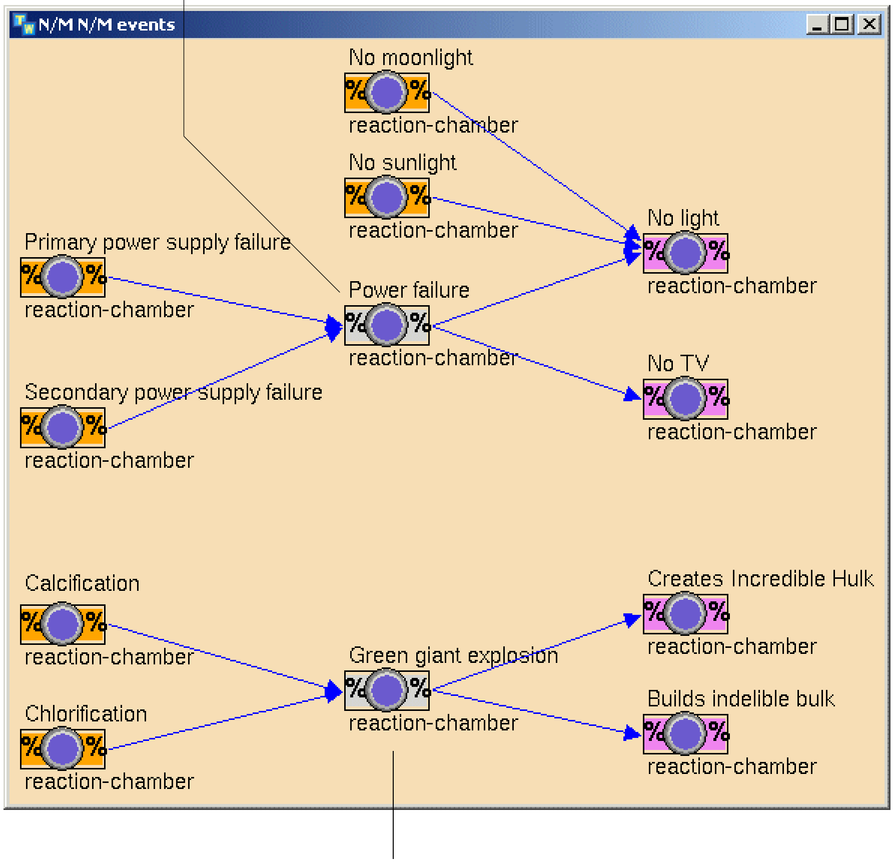
AND logic at the input (output) implies that when the event is true, all of its causes (effects) must be true. AND logic at the output is "all-or-nothing" when an event is true, i.e., if a single effect is inferred or observed to be true, both the event and all other effects must be true. When any of the effects of the event become false, by applying AND logic, the event itself becomes false; however, this does not affect the remaining effects of the event, which may continue to be true.

For all event types except the N/M-N/M event, SymCure makes the assumption that when all of the effects of the event become false, the event itself must also be false, that is, its underlying root cause must have been fixed. However, there is no fundamental reason to prefer this assumption over another, for example, a propagation delay may preclude the manifestation of any of the effects of the event.

The N/M-N/M event with output OR logic, that is, the output fraction = 0.0, may retain a value of true or suspect if its status is "specified" or "downstream inferred", even when each of its effects is false. To achieve this, the N/M-N/M event provides an option called Independent Of Effects. If enabled, this option permits a specified or downstream inferred N/M-N/M event with OR logic to retain a true or suspect value, even when all its effects are observed or inferred to be false. By default, this option is disabled.

In the following example, all the events are N/M-N/M events, which behave differently, depending on the Input and Output Fractions. For example, the "Power Failure" event has an Input Fraction of 1 and an Output Fraction of 0, which means it behaves like an AND-OR event. The "Green giant explosion" event, on the other hand, has an Input Fraction of 0 and an Output Fraction of 1, which means it behaves like an OR-AND event.

N/M-N/M event behaves like an AND-OR event.



N/M-N/M event behaves like an OR-AND event.

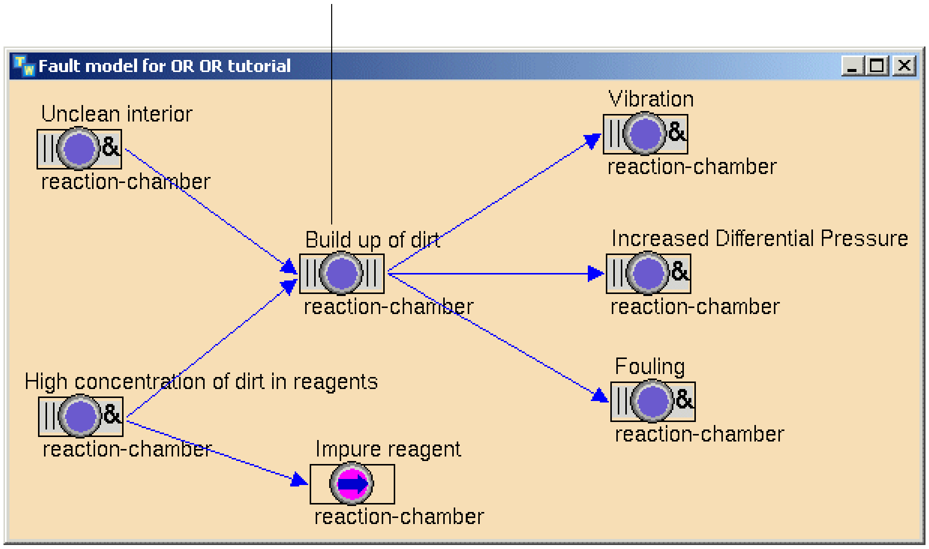
OR-OR Event

An OR-OR event uses OR logic at its output. When the event is true, this propagation logic allows the event to retain its value even after one or more of its downstream effects become false, as long as there is at least one downstream effect that is either suspect or true. Thus, SymCure can model situations where some of the effects of a true event might not be manifested.

Note This event is a legacy event that exists for compatibility with older applications.

In the following example, “Build up of dirt” causes “Vibration”, “Increased Differential Pressure” and “Fouling” in a reaction chamber. Assume that “Vibration” is not easy to detect. Suppose further that “Build up of dirt”, “Vibration”, “Increased Differential Pressure”, and “Fouling” are initially suspect. If “Build up of dirt” is modeled as an OR-AND event, the absence of “Vibration” could lead SymCure to prematurely conclude that “Build up of dirt” is no longer true and that the problems that cause “Build up of dirt” have been fixed. SymCure would then conclude “Increased Differential Pressure” and “Fouling” to be false, because the event that causes them is false. Using an OR-OR event for “Build up of dirt” allows SymCure to persist in the belief that “Build up of dirt” is suspect, as long as “Increased Differential Pressure” and “Fouling” are suspect. The OR-OR event uses OR logic at its input, thus “Build up of dirt” is true if any one or both of its causes are true.

OR-OR event



Upstream Propagation

SymCure computes the value of an OR-OR event during upstream propagation when a triggering event downstream of the OR-OR event changes value.

SymCure uses traditional OR Logic for computing the value of the event from its outputs independently of the value of the triggering event as follows:

> If there is at least one true output, conclude that the event is true,
> Else if there is at least one suspect output, conclude that the event is suspect,
> Else if there is at least one unknown output, conclude that the event is unknown,
> Else conclude that the event is false.

This is the only difference between an OR-OR event and an OR-AND event. The rest of the behavior of the OR-OR event is identical to the OR-AND event. This behavior allows an OR-OR event to retain a value of true or suspect despite one or more of its outputs being false which distinguishes it from an OR-AND event

Comparing OR-N/M and OR-OR Events

Setting the Fraction of an OR-N/M event to 0% models genuine OR logic behavior, while the OR-OR event is an expedient mixture of OR and AND logic. For this reason, we recommend that you use an OR-N/M event with a Fraction of 0 as opposed to an OR-OR event. The following description captures the differences between the two events for upstream and downstream propagation:

- Upstream propagation

  Suppose that the OR-OR or OR-N/M event with a Fraction of 0.0 is true, and that one or more of its effects are true. Now suppose that one by one, its effects become false. As long as at least one of its effects is true, the event remains true. This behavior is consistent with OR logic at its output.

  The OR-N/M event behaves differently from the OR-OR event in one other respect during upstream propagation. With the OR-N/M event, it is possible for the event to be true and for all of its effects to be suspect, as described in "Downstream Propagation" below. Now, suppose that one after the other, its effects become false, until there is just one suspect effect remaining. At this point, SymCure concludes that the last remaining effect is true. This behavior is identical to SymCure's OR logic at the input of an OR-AND event, where the last unknown cause becomes true when all other causes have been ruled out. If the last effect now also becomes false, then at that point, SymCure concludes that the OR-N/M event is false.

- Downstream propagation

  When an OR-N/M event has a Fraction of 0.0 becomes specified as true or downstream inferred as true, applying OR logic at its output, any one of its effects must be true. Therefore, SymCure tries to propagate suspect to each effect.

  When an OR-OR event becomes specified as true or downstream inferred as true, applying AND logic, SymCure tries to make all the effects of the event true, even if some are already true. In other words, the OR-OR event behaves like an OR-AND event when its value is specified as true or downstream inferred as true.

Creating Generic Fault Models

Describes how to build a generic fault model, using fault model folders, generic events, event views, and external actions.

Introduction

To build a generic fault model, you need to:

- Understand elements that make up a generic fault model.
- Create a SymCure fault model folder.
- Create and configure the appropriate events and event views in the generic fault model.
- Create and configure causal connections between the events and event views in the generic fault model.
- Compile the generic fault model.

This chapter describes the SymCure modeling language used to create, configure, and compile generic fault models, and to create tests and repair actions, configure them, and relate them to generic events.

This chapter uses examples from the `cdg-modguide.kb` application. For more information, see Running the SymCure Demos.

Elements of a Generic Fault Model

A generic fault model defines the propagation of the effects of root causes for a particular class of domain object. Generic fault models can be inherited from parent classes in a class hierarchy in accordance with object-oriented programming principles.

Generic fault models are independent of any specific collection of domain objects and relationships that exist at a particular site. Thus, diagnosis knowledge is impervious to changes in system topology and operating modes. This allows you to reuse generic fault models across different applications.

You can develop generic fault models from “first principles” models, expert knowledge, or Failure Mode Effects Analysis (FMEA) results.

Prerequisites

Before developing class-level generic fault models, you must:

- Create an appropriate domain-object class hierarchy for the diagnostic problem.
- Understand the nature of root causes, their effects, tests, and recovery methods most suited to that application. Typically, this involves answering the following questions:
  - What are the classes of objects and types of faults for which you want to receive messages and the simplest set of classes that you can use to reasonably model propagation of failures?
  - How do domain objects fail?
  - What are the most common failures?
  - What are the most significant or costly failures?
  - What are the symptoms and tests you would see in these failures?
  - From the top 20 or so messages that operators get, can you determine additional failure modes?
  - How are events related to each other?
  - What are the specific relationship definitions required for the above associations, for example, is-upstream-of, is-downstream-of?

Diagnostic Knowledge

SymCure integrates two kinds of diagnostic knowledge for fault management:

- *Fault propagation knowledge* comprises generic fault models in the form of causal relations among *generic events* defined over generic classes. This knowledge is used for diagnosis, correlation, and impact prediction.
- *Procedural knowledge* supplements the fault models by specifying processes for responding to diagnostic conclusions and predictions. It includes test mechanisms for resolving suspected root causes and repair actions for recovering from identified, predicted, and suspected failures.

Generic Events

A generic fault model for a class of domain objects defines the propagation of events within its instances and to instances of other classes via generic domain relationships. Events can represent:

- Observable symptoms.
- **Alarms** that are used to alert operators of potential problems.

- Underlying failures, that is, **root causes** of the manifested alarms.
- Events that are neither alarms nor root causes but that exist for modeling convenience only.

Fault Model Folders and Generic Event Views

Generic fault models are stored in containers called **fault model folders**. You can distribute generic events for a class definition across different fault model folders, but we recommend that you create one fault model folder for each domain object class definition.

You can use a **generic event view** to act as a bridge between events in different fault model folders or within the same folder.

Generic Tests and Repair Actions

SymCure supports two types of fault management procedures to supplement generic fault models:

- **Test actions** are used to verify the occurrence of an associated event.
- **Repair actions** are used to recover from failures.

Fault management procedures have a set of associated actions that can be applied to the managed system. The actions can be automated, they can simply be a request to an operator, or they can be a combination of the two. Such actions include extracting a datapoint from a database, “pinging” an IP address to test for network connectivity, and even sending a repair technician to a remote site to conduct manual tests and repairs. Upon completion, a test action must send a "true" or "false" event to SymCure.

The specification for a fault management procedure has three parts:

- An attribute specification that includes its name, its target domain object, the conditions for activating the procedure, and the resources, costs, and other information necessary for scheduling and executing the action.
- A relation linking it to an event in the generic fault model. The action is invoked in response to suitable state changes of an associated event.
- A procedural specification, which can be written in any computer programming language, that lays down the sequence of steps that must be performed to obtain the result for a test or to fix a problem.

Creating Fault Model Folders

You must build generic fault models within a generic fault model fault model folder. A fault model folder is a container that provides a way of organizing generic fault models. You can compile a fault model folder to view errors and warnings about the fault models.

We recommend that you organize fault model folders in an object-oriented class hierarchy with one folder per class and with inheritance according to the class hierarchy. Fault model folders can also have subfolders that correspond with subclasses in the class hierarchy.

Creating and Configuring Generic Fault Model Folders

You create and configure generic fault model folders by using the Project menu or the SymCure toolbox. The generic fault model folders appear in the Navigator.

For information on using the Project menu and Navigator, see the *Optegrity User's Guide*.

To create and configure a generic fault model folder:

1 Create a Generic Fault Model Folder from the SymCure palette and place it on any workspace.

You can also create the folder by using the Project > Logic > Diagnose > Generic Fault Models > Manage menu choice.

2 Choose Properties on the fault model folder and configure the Folder Name.

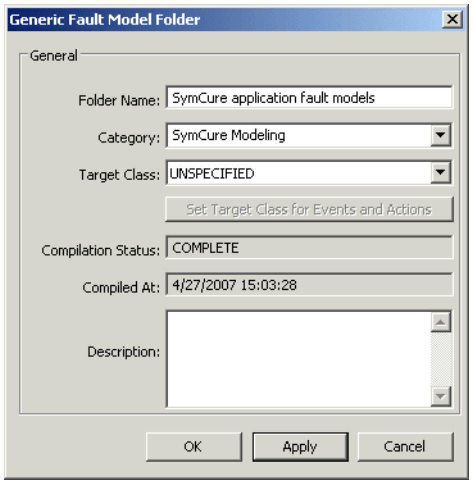
The folder name appears below the folder's icon.

3 Configure the Category to be any user-defined text used for organizing fault models in the Project > Logic > Diagnose > Generic Fault Models menu or Navigator.

If you do not specify a category, the generic fault model diagram appears under the category Unspecified.

4 Configure the Description to be a general description of the fault model.

Here is the top-level fault model folder associated with the SymCure application diagnostics example and its properties dialog:


Generic Fault Model Folder

General

Folder Name: SymCure application fault models

Category: SymCure Modeling

Target Class: UNSPECIFIED

Set Target Class for Events and Actions

Compilation Status: COMPLETE

Compiled At: 4/27/2007 15:03:28

Description:

Note In Optegrity, if you change the name of a class definition in the Domain Object Definition dialog and that class has been configured as the Target Class of a generic fault model folder, generic event, generic event view, or generic action, SymCure automatically updates the Target Class. Changing the name of a class definition that is assigned as the target class for a generic fault model does not impact existing specific fault models. However, we do not recommend changing the name of a class definition that is the target class for any fault model during an ongoing diagnostic process.

The Target Class is optional. For more information, see Asserting the Target Class.

For information about the Compilation Status and Compiled At, see Compiling a Generic Fault Model.

Creating a Fault Model Hierarchy

You can create a fault model folder hierarchy to organize generic fault models. One way to organize generic fault models is by class. By organizing diagrams by class, you can assign the target class of the fault model folder to all events in the folder.

To create a fault model folder hierarchy:

1. Create and configure a fault model folder.

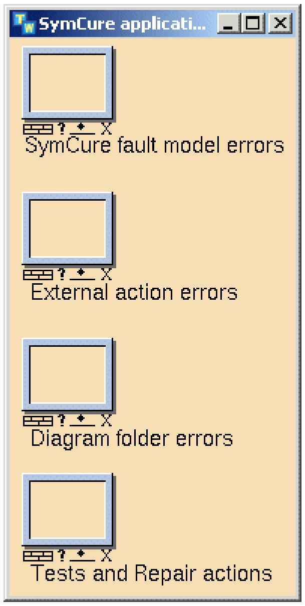
   For details, see Creating and Configuring Generic Fault Model Folders.

2. Choose Show Details on the fault model folder.

   SymCure creates a subworkspace, which is called the detail.

3. Create and configure additional fault model folders on the detail.
4. Repeat steps 2 and 3 to create as many nested fault model folders as you need.

For example, here is the top-level fault model folder for the SymCure application diagnostics example:



Asserting the Target Class

For modeling convenience, you can configure the target class of all events contained in a fault model folder, using a menu choice on the folder. This menu choice automatically configures the target class of all generic events in the fault model folder; it does not affect any of its subfolders. The menu choice affects generic events only; it does not affect generic event views.

To assert the target class of all generic events in a fault model folder:

1. In the properties dialog for the generic fault model folder, configure the Target Class.

   This property is for modeling convenience only; it has no impact on processing.

2. Choose Assert Class on the fault model folder to configure the Target Class for each generic event in the folder, using the target class of the folder.

You can manually override the target class for particular events in the folder, as needed.

For information on configuring the target class of individual generic events, see Configuring General Properties of Generic Events.

Searching for Generic Fault Models

You can search for generic fault models by keyword, target class, keyword and target class, or keyword or target class.

To search for generic events:

1. Choose Tools > Search > Fault Models > Generic Fault Models or click the equivalent button in the Fault Modeling toolbar ( ).
2. Provide the Keyword and/or Target Class to search for.
3. Configure Search By to determine how to combine Keyword and Target Class in the search.
4. Click the Search button.

   A list of generic fault models that meet the search criteria appears; otherwise, No Matches Found appears.

5. Select a generic fault model and click the Go To button to go to the generic fault model.

Creating Generic Events

To create a causal fault model, you place generic events on the detail of a diagram folder and connect them to establish causal relations. In a causal model, the direction of the arrowhead on a connection determines causality. Thus, an upstream event in the model is said to cause a downstream event.

Note Generic events can only be placed in a fault model folder. Generic events placed anywhere else are automatically deleted.

At run time, SymCure derives a specific fault model for a collection of specific domain object from its generic fault model library.

When creating a generic event, you must configure its event name and target class.

You can also configure:

- The message that appears in the built-in message browsers when a specific event corresponding with the generic event occurs on a domain object.
- User-defined procedures that SymCure executes at run time when a specific event corresponding with the generic event either occurs or does not occur for a specified period of time.
- Whether the event can trigger diagnostic processing.
- Whether the event is to be treated as an alarm or a root cause.
- Additional properties for the various types of generic events.

Note If you change the name of a generic event, SymCure automatically updates any GEDP Send Fault Model Event blocks that refer to the event.

Note During an active deployment of a SymCure application, you should not delete generic events from a generic fault model; otherwise, errors will occur. If you need to delete a fault model, be sure to bring the application offline first.

For information on configuring the target class automatically for all events in a fault model folder, see Asserting the Target Class.

For information on connecting generic events when the target classes are related, such as “connected to” or “contained in,” see Configuring Causal Connections.

For information on specific fault models, see Running SymCure Applications.

Creating and Connecting Generic Events

By default, causal connections assume that the connected events have the same target class.

To create and connect generic events:

1. Show the fault model folder detail on which you want to place a generic event.

   For details, see Creating Fault Model Folders.

2. Create a generic event from the SymCure palette and place it on the fault model folder detail.

3. Choose Properties on the generic event and configure the Event Name to be any user-defined text.

   The event name is displayed above the icon for the generic event.

4. Configure the Target Class to be the domain object class to which the generic event applies.

   The target class is displayed below the icon for the generic event.

5. Create as many generic events as you need to describe the causal relationships in the generic event model.

6. Drag the downstream connection stub from one event into the upstream connection stub of another event.

   By default, the causal connection establishes a causal relationship between generic events on the same object. Such connections appear blue in the diagram.

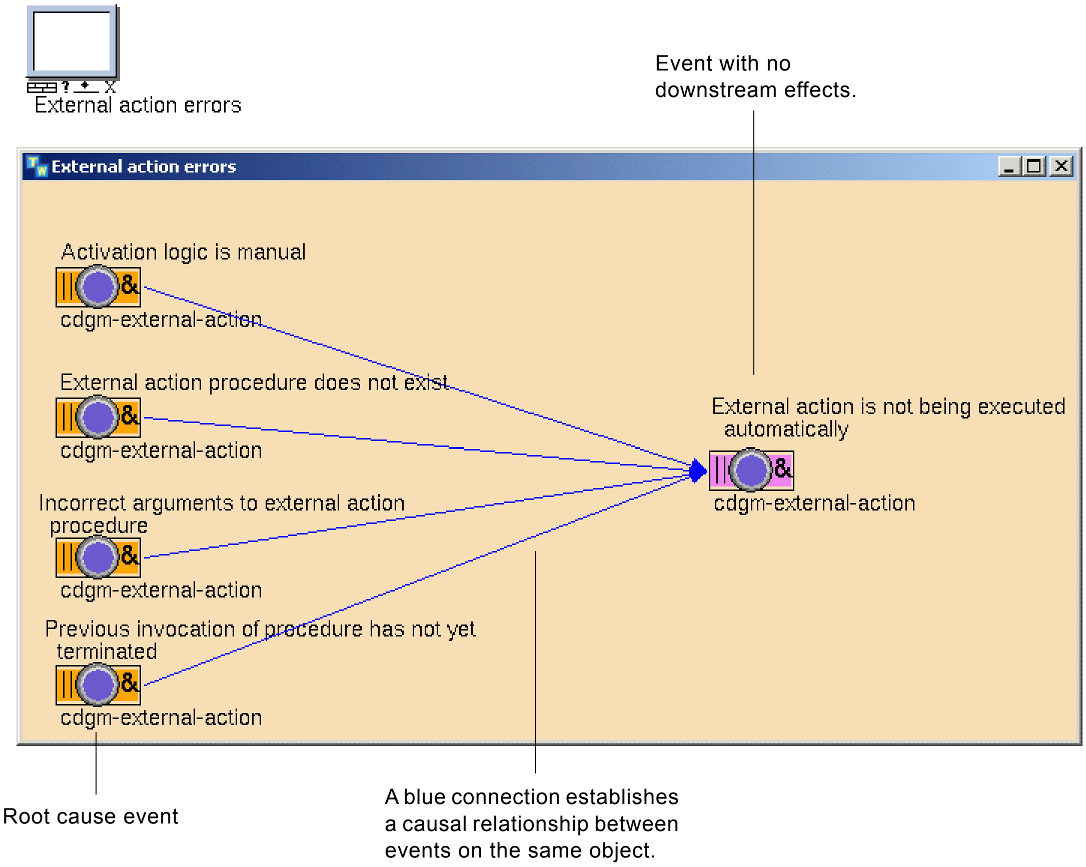
7. For root cause events, drag the upstream connection stub into the event object to remove it, indicating it has no upstream causes.

8. For events with no downstream effects, drag the downstream connection stub into the event object to remove it, indicating it has no downstream effects.

Tip Use the Add Stubs menu choice on the generic event to add input and output stubs, as needed.

Tip Compiling a fault model folder automatically removes extraneous stubs.

For example, here is the detail of the "External action errors" fault model folder in the SymCure application diagnostics example. Notice that the target class of each connected generic event is external-action, which means it uses the default directed connection, which is blue.

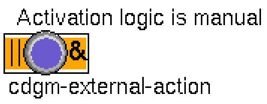

External action errors
External action errors
Activation logic is manual
cdgm-external-action
External action procedure does not exist
cdgm-external-action
Incorrect arguments to external action procedure
cdgm-external-action
Previous invocation of procedure has not yet terminated
cdgm-external-action
External action is not being executed automatically
cdgm-external-action

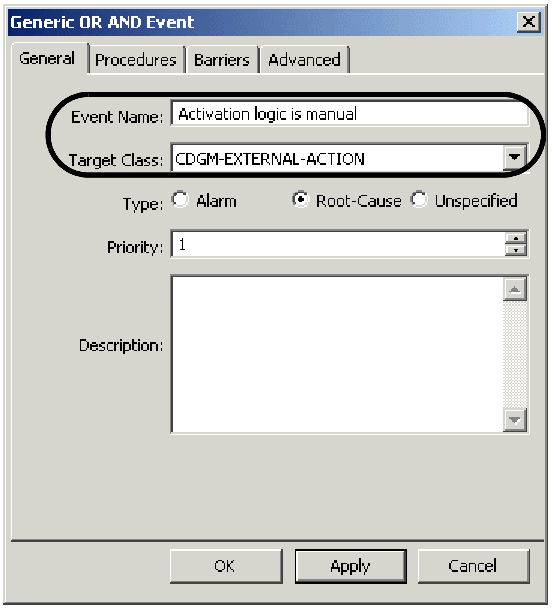
Event with no downstream effects.

Root cause event

A blue connection establishes a causal relationship between events on the same object.

Here is the “Activation logic is manual” event and its properties dialog, which applies to the cdgm-external-action class:





Event Name: Activation logic is manual

Target Class: CDGM-EXTERNAL-ACTION

Type: ○ Alarm ● Root-Cause ○ Unspecified

Priority: 1

Description:

Configuring General Properties of Generic Events

In addition to configuring the event name and target class of a generic event, which are required, you can configure these additional general properties:

- Type — Whether the event is an alarm, root cause, or unspecified. SymCure uses the event type for reporting events in one of the built-in message browsers for operator notification and intervention. The event type does not impact diagnostic reasoning in any way.

  By default, the event type is unspecified, which means it does not appear in any message browser. Alarm events appear in the Alarms Browser, and root cause events appear in the Root Causes browser.

  The background color of the generic event icon indicates the event type, as follows:

  - Root cause is orange.
  - Alarm is violet.
  - Unspecified is light-gray.

- Priority — An integer that represents the level of importance of the event, the likelihood that the event has occurred, or any other numerical measure. SymCure uses the priority to prioritize suspected root causes. The priority appears in the built-in browsers as the priority, which you can use for sorting and filtering messages.

By default, all events can initiate diagnostic processing.

For information on interacting with alarms and root causes in the built-in message browsers, see Interacting with Specific Events and Actions through Diagnostic Console Browsers.

You can configure the default priority of all generic events with the same target object in the configuration file. You can also configure a procedure to compute the priority for all specific events or a particular specific event, based on the generic event priority and the domain object on which the root cause event occurs. For more information, see Priority in Configuring SymCure Applications.

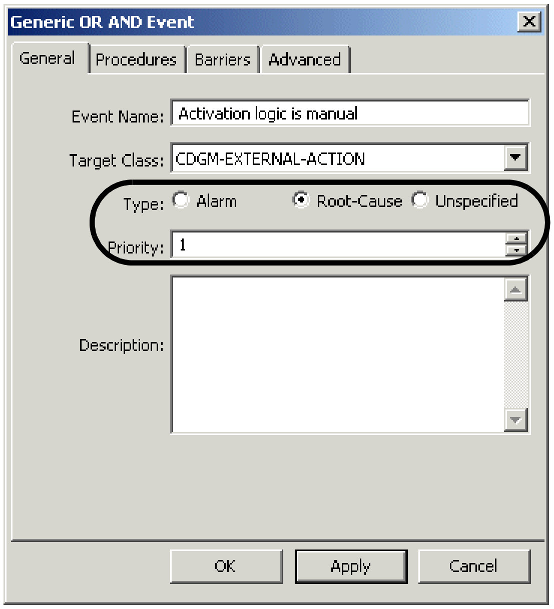
To configure general properties of a generic event:

1 Display the properties dialog for the generic event and configure the Type to be alarm, root-cause, or unspecified.

If the event is neither an alarm nor a root cause, use the default, which is unspecified.

2 Configure the Priority to be any integer that prioritizes the event, as needed.

The “Activation logic is manual” event type is root-cause and the priority is 1:



Event Name: Activation logic is manual

Target Class: CDGM-EXTERNAL-ACTION

Type: ○ Alarm ● Root-Cause ○ Unspecified

Priority: 1

Description:

Configuring User-Defined Procedures for Generic Events

You can configure generic events with these user-defined procedures that can execute at run time under different conditions:

- Event Changed procedure — A user-defined procedure that is invoked when the state of a specific event associated with the generic event changes.
- Event Unchanged procedure — A user-defined procedure that is invoked when the state of a specific event does not change over a specified time interval.
- Occurs At procedure — A user-defined procedure that is invoked to compute the inferred occurrence time of an event, for example, based on the fraction of true inputs over a time period.

When using the Event Changed and Event Unchanged procedures, a change in state includes a change in event value or status. The event value can be true, false, unknown, or suspect, and the event status can be specified, upstream inferred, or downstream inferred. For details, see Event Propagation.

For example, you might use the Event Changed procedure to invoke some type of audit procedure when the event value changes to "suspect" or whenever the status is "upstream inferred" or "downstream inferred."

You might use the Event Unchanged procedure when an event is true or suspect and, for some reason, it is not attended to for a long time. For fault management applications, it might be necessary to alert the operator or perform some action when the event state does not change after a period of time. You can use the event unchanged procedure to alert operators that a fault has not been repaired or that a symptom remains unexplored, even after the passage of a considerable period of time.

You might use the Occurs At procedure to determine when an event is true, based on the fraction of true inputs over a period of time. For example, an event might become true in 5 days if only 1/5th of its inputs are true, but in 1 day if 4/5th of its inputs are true. Aircrafts are often equipped with multiple engines. Often a plane can fly even if one or more of its engines are malfunctioning, but as the number of malfunctioning engines increase, the amount of time that the plane can safely stay in the air is reduced.

You can create the procedures from the palettes by choosing View > Toolbox - Fault Models > User-Defined Procedures and Methods and creating one of the following, which have these signatures:

- Generic Event Changed Procedure or Generic Event Changed Method.

  my-event-changed-proc
  (*Target*: class grtl-domain-object,
  SpecificEvent: cdg-specific-event,
  TimeStamp: integer, *Client*: class object)

  where:

  Target is the target class of the generic event.

  SpecificEvent is the specific event associated with an instance of the target class.

  TimeStamp is the timestamp at which the procedure executes, in seconds.

  Client is the G2 window on which the procedure should execute.

- Generic Event Unchanged Procedure or Generic Event Unchanged Method.

  my-event-unchanged-proc
  (*Target*: class grtl-domain-object,
  SpecificEvent: cdg-specific-event,
  EventUnchangedDuration: integer)

  where:

  Target is the target class of the generic event.

  SpecificEvent is the specific event associated with an instance of the target class.

  EventUnchangedDuration is the duration of time, in seconds, that the event has not changed state, that is, the time for which neither the value nor the status of the event has changed. The procedure executes when this time period has passed.

- Generic Event Occurs At Procedure

  my-compute-inferred-occurrence-time-proc
  (*SpecificEvent*: class cdg-specific-event)
  -> *inferred-occurrence-time*: quantity

  where:

  SpecificEvent is the specific event associated with an instance of the target class.

  inferred-occurrence-time is the inferred occurrence time of *SpecificEvent*.

The procedure's signature is automatically populated from its signature attribute. You can configure the procedure for the generic event by choosing from a list of such procedures in the properties dialog for the generic event.

If you use SymCure's specialized procedures to implement one of an event's user-defined procedures, these procedures appear in the dropdown list for the event's Event Changed, Event Unchanged, and Occurs At attributes. If you use a G2 procedure for this purpose, the procedure will not appear in the dropdown list.

You can control the behavior of the event unchanged procedure mechanism. For details, see Event Unchanged Procedure in Configuring SymCure Applications,

The procedures are invoked by SymCure's diagnostic algorithm in separate threads.

To configure user-defined procedures for a generic event:

1. Clone one of the appropriate types of procedures or methods from the User-Defined Procedures and Methods palette of the Fault Modeling toolbox and configure the procedure or method, as desired.
2. Display the properties dialog for a generic event.
3. On the Procedures tab, configure the Event Changed, Event Unchanged, and/or Occurs At to be your user-defined procedure.

To go to the user-defined procedure from a generic event:

➔ Choose Go to Event Changed Procedure, Go to Event Unchanged Procedure, or Go To Occurs At Procedure on a generic event.

If the specified procedure exists, SymCure displays the procedure with a red arrow pointing to it.

Note that these menu choices are enabled only if the corresponding attribute is specified.

Controlling the Size of the Specific Fault Model

Typically, if an event is false or unknown, constructing the specific fault model upstream of that event does not contribute towards isolating the root causes responsible for other observed true events. Similarly, constructing the specific fault model downstream of that event does not contribute toward identifying predicted alarms.

To preserve memory and increase efficiency of diagnostic processing, you can control the size of the specific fault model to construct only the events that are relevant to diagnostic problem solving by configuring the following attributes on generic events:

- Upstream Barrier — The set of event values for which upstream construction of the specific fault model does not occur.

- Downstream Barrier — The set of event values for which downstream construction of the specific fault model does not occur.

By default, SymCure blocks upstream and downstream construction of the specific fault model when the value of a specific event is false or unknown. To block upstream or downstream construction completely, include every possible event value in the set.

Note For accurate propagation, the upstream barrier of an event that is connected downstream of an N/M-N/M or an OR-N/M event must be empty.

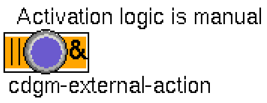
The upstream and downstream barriers do not preclude propagation to events that have already been constructed upstream and downstream of an event.

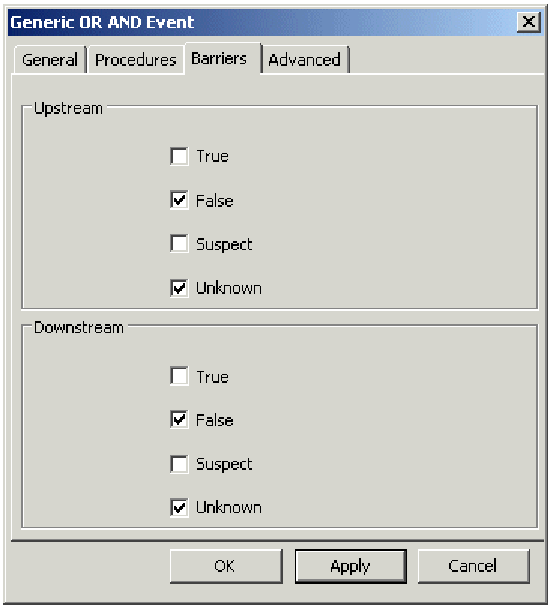
You can also configure various parameters to control the rate at which the events are constructed upstream and downstream. For details, see Specific Fault Model Creation.

To control the size of the specific fault model:

1. Display the properties dialog for a generic event and click the Barriers tab.
2. Configure the Upstream and/or Downstream barrier to include event values for which upstream and downstream specific event construction does not occur.

Here is the Barriers tab of the “Activation logic is manual” event, which uses the default Upstream and Downstream barrier:





Upstream

☐ True

☑ False

☐ Suspect

☑ Unknown

Downstream

☐ True

☑ False

☐ Suspect

☑ Unknown

Configuring Operator Messages for Generic Events

You can configure a generic event so that when an alarm or root cause event occurs, a suitable message is sent to the default operator message browser.

To customize the operator message and to provide additional information to the operator, you can configure the event to generate messages when the value of the event is true, false, suspect, and/or unknown. By default, operator messages are disabled for all event values. You must explicitly enable message generation for one or more event values.

You can only configure operator messages for events of type Alarm or Root Cause; you cannot configure messages for events of type Unspecified.

You can configure these properties:

- Message — The message text to display in the Alarms Browser or Root Causes Browser when the event has the specified value.
- Detail — Detailed information about the event, which the operator can view by showing details for the event.

- Advice — Advice about how to recover from the event, which the operator can also view in the message details.
- Priority — A priority from 1 to 9 for the message. The default priorities are 1 when the value is true, 5 when the value is false, 3 when the value is suspect, and 7 when the value is unknown.

The message text and message details can include text substitutions, which are references to any attribute of the specific event, using this syntax:

$*event-attribute*

By default, the message that appears has the following format:

```
$EVENT-NAME on $TARGET-OBJECT is $EVENT-VALUE
```

The corresponding message looks similar to this:

Activation logic is manual on TEST-1 is TRUE

You can also refer to any attribute of the underlying target object, using the same syntax. For example, if the domain object defines an attribute named `process-temp`, you could refer to the value of this attribute in the message by using `$PROCESS-TEMP`.

The Detail attribute contains a complete list of available text substitutions that you can use in either the message text or detail text. You configure the message text, using the desired substitutions and delete the rest.

Here is a list of text substitutions that you can use:

Text Substitution	**Description**
$EVENT-NAME	The event name.
$TARGET-OBJECT	The name of the domain object on which the event occurred.
$EVENT-VALUE	The value of the event (true, false, suspect, or unknown).
$EVENT-STATUS	The status of the event (specified, upstream inferred, downstream inferred, or mutually exclusive)
$HISTORY	The history of event values.
$TIME-STAMP	The time at which the event occurred.

Text Substitution	Description
$INFERRED-OCCURRENCE-TIME	The time at which the event value was inferred.
$OCCURS $BECOMES	See Describing When an Event Occurs

By default, any message about an event depends only on changes to its value; thus, when an event is inferred to be true (false), the message it generates (retracts) overrides any previously generated messages for the event, regardless of whether the previously generated message is a consequence of a specified event.

You might want a message generated by an inferred event not to override a message generated when that event is specified. Consider that a symptom event is specified to be true. Now suppose that after due processing, SymCure infers that the symptom event is false, perhaps because it ran a repair action on the underlying root cause for the symptom. From the fault model's perspective, it is quite reasonable to infer that the symptom is no longer true. However, from an operator's perspective, particularly in the process world where events are monitored continuously, until the symptom is reported to be false, the operator should not be told that it is false. In this example, while the underlying event may be inferred to be false, the message displayed to an operator must continue to treat the event as true. To accomplish this, you disable the Override Specified Event Messages option.

For general information on interacting with messages in the built-in message browsers, see Interacting with Specific Events and Actions through Diagnostic Console Browsers. For information on viewing message details and advice, see Showing Event Properties.

For information about the properties of specific events, see Showing Specific Event Properties.

To configure operator messages for generic events:

1. Choose Configure Messages on the generic event.

   You can also click the Configure Messages button on the Advanced tab of the generic event properties dialog.

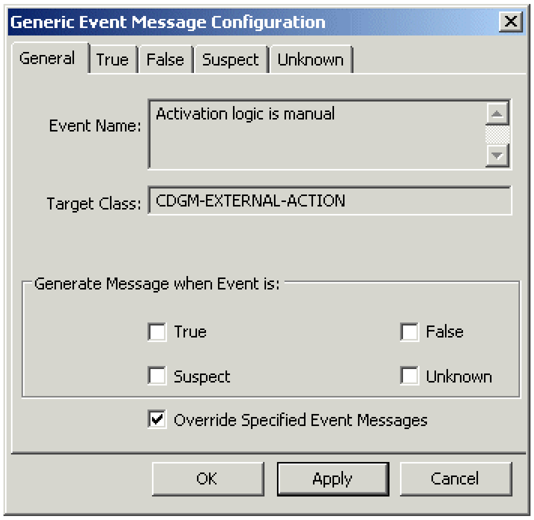
2. On the General tab, configure the Generate Message When Event Is to be true, false, suspect, and/or unknown.

   SymCure generates messages when the value of the specific event becomes any one of the enabled values.

3 Configure the Override Specified Event Messages, as needed.

4 Click the True, False, Unknown, and/or Suspect tab for the values whose message you want to configure, and configure the Message, Details, and/or Advice to be any text, including text substitutions.

Here is the General tab for a generic event that generates messages only when the value of the underlying event is true and uses the default behavior for overriding specified event messages:


Generic Event Message Configuration

Event Name: Activation logic is manual

Target Class: CDGM-EXTERNAL-ACTION

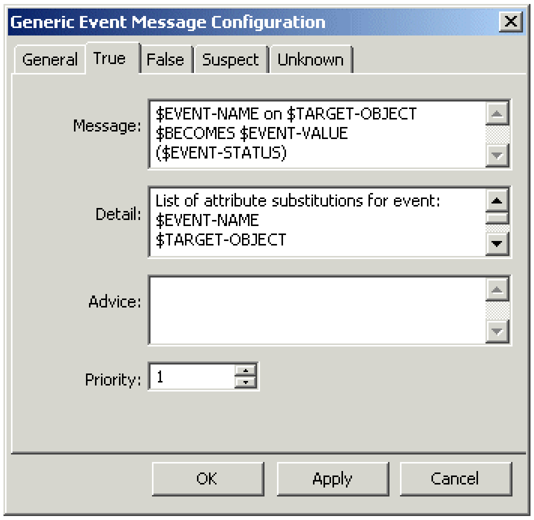
Generate Message when Event is:

☐ True ☐ False

☐ Suspect ☐ Unknown

☑ Override Specified Event Messages

Here is the default message configuration of the True tab for the generic event:


Generic Event Message Configuration

Message: $EVENT-NAME on $TARGET-OBJECT $BECOMES $EVENT-VALUE ($EVENT-STATUS)

Detail: List of attribute substitutions for event: $EVENT-NAME $TARGET-OBJECT

Advice:

Priority: 1

Describing When an Event Occurs

A message may describe a past event, that is, one that has already occurred, a present event, that is, one that is in process, or a future event, that is, one that may happen in time. For many applications, predicting future events in order to prevent them from ever occurring is paramount.

SymCure recognizes whether an event has occurred, is occurring, or will occur, and can construct the message text and set its priority accordingly. To do this, you use the text substitution $OCCURS in an event message configuration, for which SymCure substitutes “has occurred,” “is occurring,” or “will occur,” depending on whether the event described by the message is in the past, present, or future.

You can also use the text substitution $BECOMES, for which SymCure substitutes “became”, “has become”, or “will become”, as required.

You can also extend the set of verbs that you can use to configure the text of a message, as follows.

To extend the set of verbs that you can use to configure the text of a message:

1 Add the tags to the parameter `CDG-MESSAGE-SUBSTITUTION-VERB-TAGS` in the `config.txt` file, located in the `g2i\kbs` directory.

The default value for this parameter is:

```
CDG-MESSAGE-SUBSTITUTION-VERB-TAGS=$BECOMES $OCCURS
```

For example, you can add verb tags for the verbs “explodes” and “implodes,” as follows:

```
CDG-MESSAGE-SUBSTITUTION-VERB-TAGS=$BECOMES $OCCURS $EXPLODES
$IMPLODES
```

2 For each verb tag, add the following lines to `resources-english.txt`:

```
CDG-MESSAGE-[verb]-PAST-TENSE, "your text"

CDG-MESSAGE-[verb]-PRESENT-TENSE, "your text"

CDG-MESSAGE-[verb]-FUTURE-TENSE, "your text"
```

For example, to use the correct tense for the verb “explodes” in your message, add the following lines to your `resources-english.txt` file:

```
CDG-MESSAGE-EXPLODES-PAST-TENSE, "has exploded"

CDG-MESSAGE-EXPLODES-PRESENT-TENSE, "is exploding"

CDG-MESSAGE-EXPLODES-FUTURE-TENSE, "will explode"
```

3 Use your new verb tag while configuring the text, detail, or advice of the message.

For example:

```
$EVENT-NAME $EXPLODES at $TIME-STAMP
```

Configuring Generic OR-AND, AND-AND, and OR-OR Events

To configure generic OR-AND, AND-AND, and OR-OR events, configure the properties described in these sections:

- Creating and Connecting Generic Events.
- Configuring General Properties of Generic Events.
- Configuring User-Defined Procedures for Generic Events.
- Controlling the Size of the Specific Fault Model.
- Configuring Operator Messages for Generic Events.

For descriptions of these generic events, see:

- OR-AND Event.
- AND-AND Event.
- OR-OR Event.

Configuring Generic N/M-AND Events

To configure generic N/M-AND events, you configure the same properties as the OR-AND event. In addition, configure the following property on the Properties tab:

> fraction — The percentage of inputs that must be true during downstream propagation for the event to be true. When the value is 0.0, this event behaves like an OR-AND event. When the value is 1.0, the event behaves like an AND-AND event.

For a description of this event, see N/M-AND Event.

For information on configuring generic OR-AND events, see Configuring Generic OR-AND, AND-AND, and OR-OR Events.

Configuring Generic IF-AND Events

To configure generic IF-AND events, you configure the same properties as the OR-AND event. In addition, configure the following property on the Properties tab:

> State Dependent Procedure — The name of a G2 procedure or method that the generic IF-AND event executes to get the value of the event. The procedure must return a value for the IF-AND event.

You can create the procedure from the palettes by choosing View > Toolbox - Fault Modeling > User-Defined Procedures and Methods and creating a Generic IF-AND Event State Dependent Procedure or Method. The procedure's signature is automatically populated from its signature attribute. You can configure the procedure for the generic event by choosing from a list of such procedures in the properties dialog for the generic event.

The signature of the procedure is:

```
my-state-dependent-proc
  (target: class grtl-domain-object,
   specific-event: class cdg-specific-event,
   specific-cause: class cdg-specific-event)
   -> event-value: text
```

where:

target is the target domain object of the specific event.

specific-event is the specific event.

specific-cause is the upstream specific event that is responsible for the propagation to the IF-AND event, if there are multiple events upstream of the IF-AND event.

The procedure returns the value of the event as a text.

You can go to the state dependent procedure from a generic event, using a menu choice.

To go to the state dependent procedure:

➔ Choose Go to State Dependent Procedure on a generic event.

SymCure places an arrow next to the specified procedure, if one is specified.

For a description of this event, see IF-AND Event

For information on configuring generic OR-AND events, see Configuring Generic OR-AND, AND-AND, and OR-OR Events.

Configuring Generic OR-N/M Events

To configure generic OR-N/M events, you configure the same properties as the OR-AND event. In addition, configure the following property on the Properties tab:

> Fraction — The percentage of directly connected downstream effects that must be true during upstream propagation for the event to be true.

For information on this event, see OR-N/M Event.

Configuring Generic N/M-N/M Events

To configure generic N/M-N/M events, you configure the same properties as the OR-AND event. In addition, configure the following properties on the Properties tab:

- Input fraction **—** The percentage of inputs that must be true during downstream propagation for the event to be true, where:
  - Input Fraction = 0.0 -> OR logic
  - 0.0 < Input Fraction < 1.0 -> N/M logic
  - Input Fraction = 1.0 -> AND logic
- Output Fraction — The percentage of directly connected downstream effects that must be true during upstream propagation for the event to be true, where:
  - Output Fraction = 0.0 -> OR logic
  - 0.0 < Output Fraction < 1.0 -> N/M logic
  - Output Fraction = 1.0 -> AND logic
- Independent Of Effects — When Output Fraction = 0.0 (OR logic) and this option is enabled, the event retains a value of true or suspect if its status is "specified" or "downstream inferred", even when each of its effects is false. By default, this option is disabled.

For information on this event, see N/M-N/M Event.

For information on tuning the input and output properties of a generic N/M-N/M event from a specific event, see Learning Generic Models from Specific Events.

Converting Generic Event Logic

Each type of generic event defines menu choices for converting the upstream and downstream event logic that the event uses. Note that the N/M-N/M event does not define menu choices for converting event logic, because you can obtain most desired behaviors by configuring the input and output fractions of the event.

Generic OR-AND Event Menu Choices

Generic OR-AND events define these menu choices:

Menu Choice	Description
Convert Output Logic to NM	Converts the output logic to NM, causing the event to become an OR-N/M event.
Convert Output Logic to Or	Converts the output logic to OR, causing the event to become an OR-OR event.
Convert Input Logic to NM	Converts the input logic to NM, causing the event to become an N/M-AND event.
Convert Input Logic to If	Converts the input logic to IF, causing the event to become an IF-AND event.
Convert Input Logic to And	Converts the input logic to AND, causing the event to become an AND-AND event.

Generic AND-AND Event Menu Choices

Generic AND-AND events define these menu choices:

Menu Choice	Description
Convert Input Logic to NM	Converts the input logic to NM, causing the event to become an N/M-AND event.
Convert Input Logic to If	Converts the input logic to IF, causing the event to become an IF-AND event.
Convert Input Logic to Or	Converts the input logic to OR, causing the event to become an OR-AND event.

Generic OR-OR Event Menu Choices

Generic OR-OR events define these menu choices:

Menu Choice	Description
Convert Output Logic to NM	Converts the output logic to N/M, causing the event to become an OR-N/M event.
Convert Output Logic to And	Converts the output logic to AND, causing the event to become an OR-AND event.

Generic N/M-AND Event Menu Choices

Generic N/M-AND events define these menu choices:

Menu Choice	Description
Convert Input Logic to Or	Converts the input logic to OR, causing the event to become an N/M-OR event.
Convert Input Logic to And	Converts the input logic to AND, causing the event to become an N/M-AND event.

Generic OR-N/M Event Menu Choices

Generic OR-N/M events define these menu choices:

Menu Choice	Description
Convert Output Logic to And	Converts the output logic to AND, causing the event to become an OR-AND event.
Convert Output Logic to Or	Converts the input logic to OR, causing the event to become an OR-OR event.

Generic IF-AND Event Menu Choices

Generic IF-AND events define these menu choices:

Menu Choice	Description
Convert Input Logic to NM	Converts the input logic to N/M, causing the event to become an IF-N/M event.
Convert Input Logic to Or	Converts the input logic to OR, causing the event to become an IF-OR event.
Convert Input Logic to And	Converts the input logic to AND, causing the event to become an IF-AND event.

Going to Generic Event-Detection Diagrams

SymCure generic events allow you to navigate to generic event-detection diagrams that contain a Send Fault Model Event block that refers to the generic event. Similarly, the GEDP Send Fault Model Event block provides the Show Fault Model Event menu choice for navigating to the generic event in the fault model.

For more information, see the *G2 Event and Data Processing User's Guide*.

To go to generic event-detection diagrams:

➔ Choose Show Event Detection Diagrams on a generic event.

This menu choice only appears if the generic event is specified in a generic event-detection diagram.

Showing Detailed Explanations of Generic Events

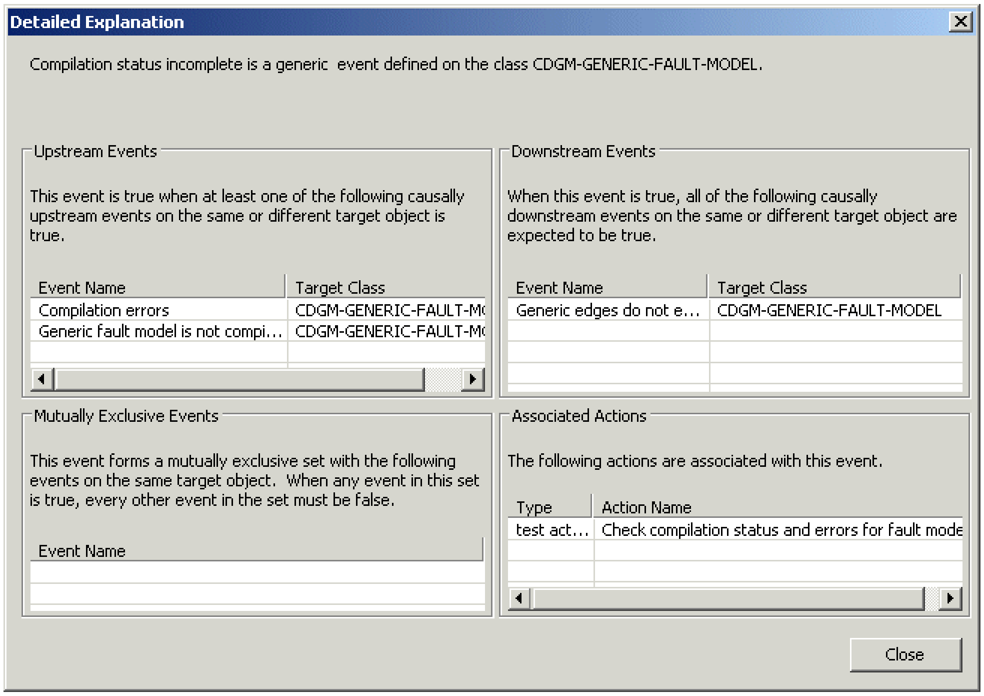
You can view a detailed explanation about a generic event, which includes information about:

- Logical relationship with upstream events.
- Logical relationship with downstream events.
- Associated actions.
- Mutually exclusive events.

To show a detailed explanation about a generic event:

➔ Choose Detailed Explanation on the generic event.

For example, here is the detailed explanation for a generic event with two upstream events, a downstream event, and an associated action:



Compilation status incomplete is a generic event defined on the class CDGM-GENERIC-FAULT-MODEL.

Upstream Events

This event is true when at least one of the following causally upstream events on the same or different target object is true.

Event Name	Target Class
Compilation errors	CDGM-GENERIC-FAULT-M(
Generic fault model is not compi…	CDGM-GENERIC-FAULT-M(

Downstream Events

When this event is true, all of the following causally downstream events on the same or different target object are expected to be true.

Event Name	Target Class
Generic edges do not e…	CDGM-GENERIC-FAULT-MODEL

Mutually Exclusive Events

This event forms a mutually exclusive set with the following events on the same target object. When any event in this set is true, every other event in the set must be false.

Event Name

Associated Actions

The following actions are associated with this event.

Type	Action Name
test act…	Check compilation status and errors for fault mode

Searching for Generic Events

You can search for generic events by keyword, target class, keyword and target class, or keyword or target class.

To search for generic events:

1 Choose Tools > Search > Fault Models > Generic Events or click the equivalent button in the Fault Modeling toolbar ( ).

2 Configure the Keyword and/or Target Class.

3 Configure Search By to determine how to search.

4 Click the Search button.

A list of generic events that meet the search criteria appears; otherwise, No Matches Found appears.

5 Select a generic event and click the Go To button to go to the generic event.

Creating Generic Event Views

You can establish a causal relationship between events in separate fault model folders by using event views, which act as bridges between different fault model folders. Event views are typically used when generic fault models are organized by class to trigger events on related classes defined in separate folders.

Instances of the classes can be related, using any built-in G2 relation, such as containment or connection, or any user-defined relation. For more information on how events can be related, see Configuring Causal Connections.

You can also use event views to provide a bridge between events in the same fault model folder, which can help to organize the fault model to make it easier to read.

Using Generic Event Views to Bridge Events in Separate Fault Model Folders

A single event view can map to as many generic events that match its event name and target class.

To use a generic event view to bridge two diagrams:

1. Create and configure a generic event on the detail of a fault model folder.

   For details, see Creating Generic Events.

2. Create a generic event view from the SymCure palette and place it on the detail of another fault model folder.

3. Choose Properties on the generic event view and configure the Event Name to be the same as the event name of a generic event in another diagram.

   The event name is displayed above the icon for the generic event view.

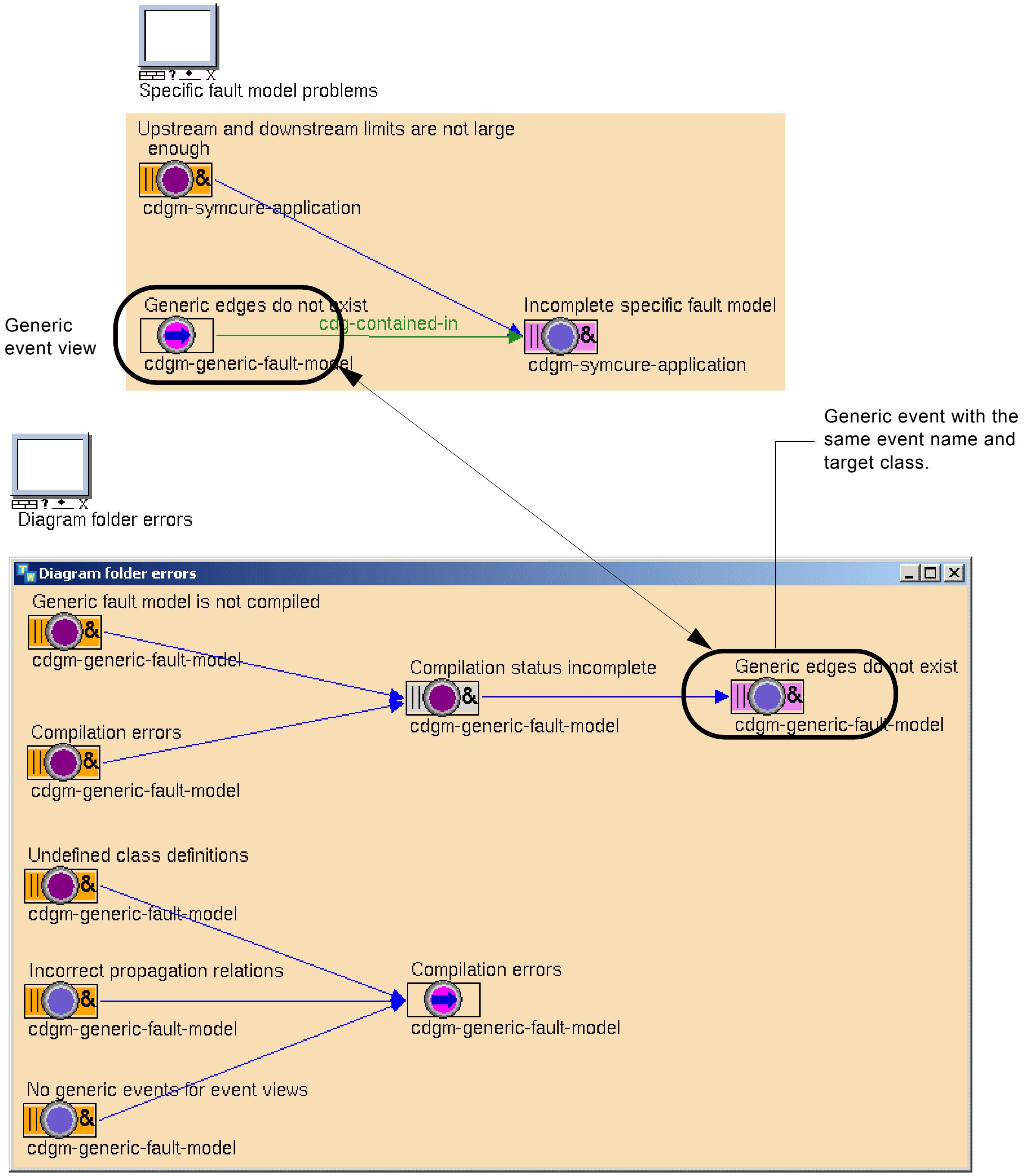
4. Configure the Target Class to be the domain object class to which the associated generic event applies.

   The target class for a generic event view maps to the target class of the generic event or any subclass of the target class. The target class is displayed below the icon for the generic event view.

5. Use the event view in the diagram as if it were a generic event by connecting upstream or downstream events to the event view.

Note You cannot connect two event views together; an event view must connect to a generic event.

Here is part of the subworkspace of the “SymCure application errors” fault model folder with a generic event view named “Generic edges do not exist”. The event view propagates events to the generic event with the same name in the “Diagram folder errors” fault model folder.



Generic event view

Generic event with the same event name and target class.

Using Generic Event Views in the Same Fault Model Folder

To use generic event views in the same diagram:

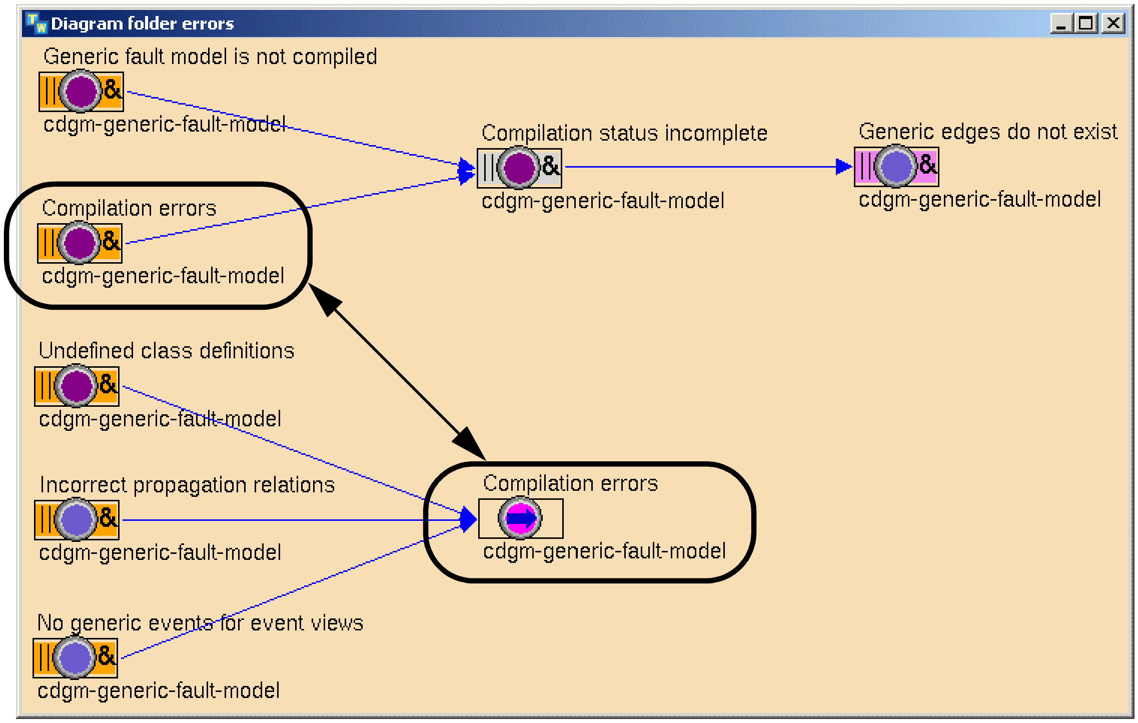
1 Create and configure a generic event on the detail of a fault model folder.

For details, see Creating Generic Events.

2 Create, configure, and connect a generic event view on the same fault model folder.

For details, see Using Generic Event Views to Bridge Events in Separate Fault Model Folders.

Here is part of the "Diagram folder errors" fault model in which the "Compilation errors" generic event and generic event view appear in the same fault model folder:



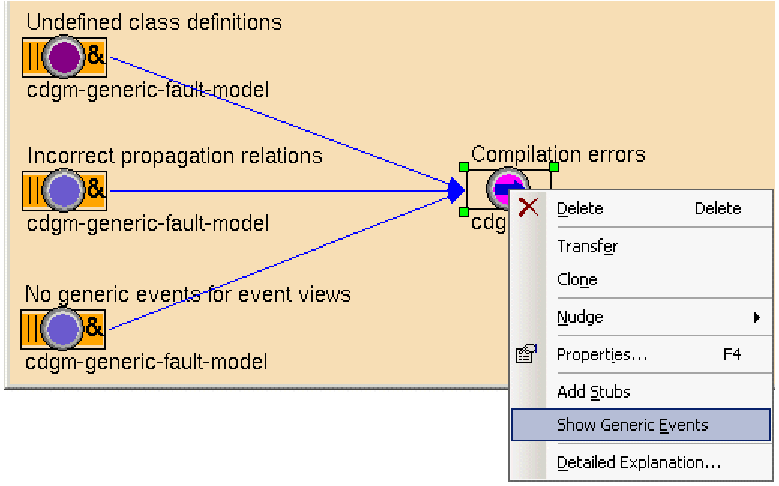
Going to the Associated Generic Event

You can show the generic events associated with a generic event view, then go to an associated generic event.

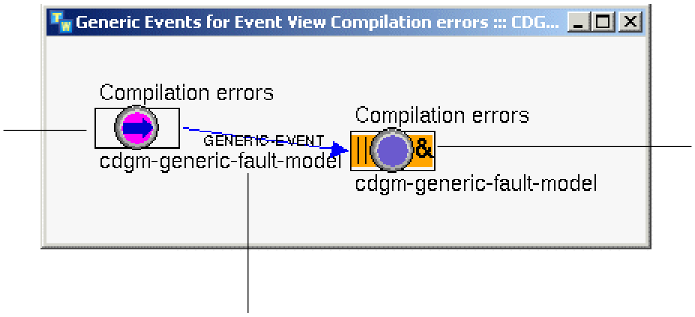
To go to the associated generic event:

1 Choose Show Generic Events on a generic event view.

For example:



SymCure displays a workspace with objects that represent the generic event view and its associated generic event. The generic-event label describes the relation between the generic event view and the generic event.



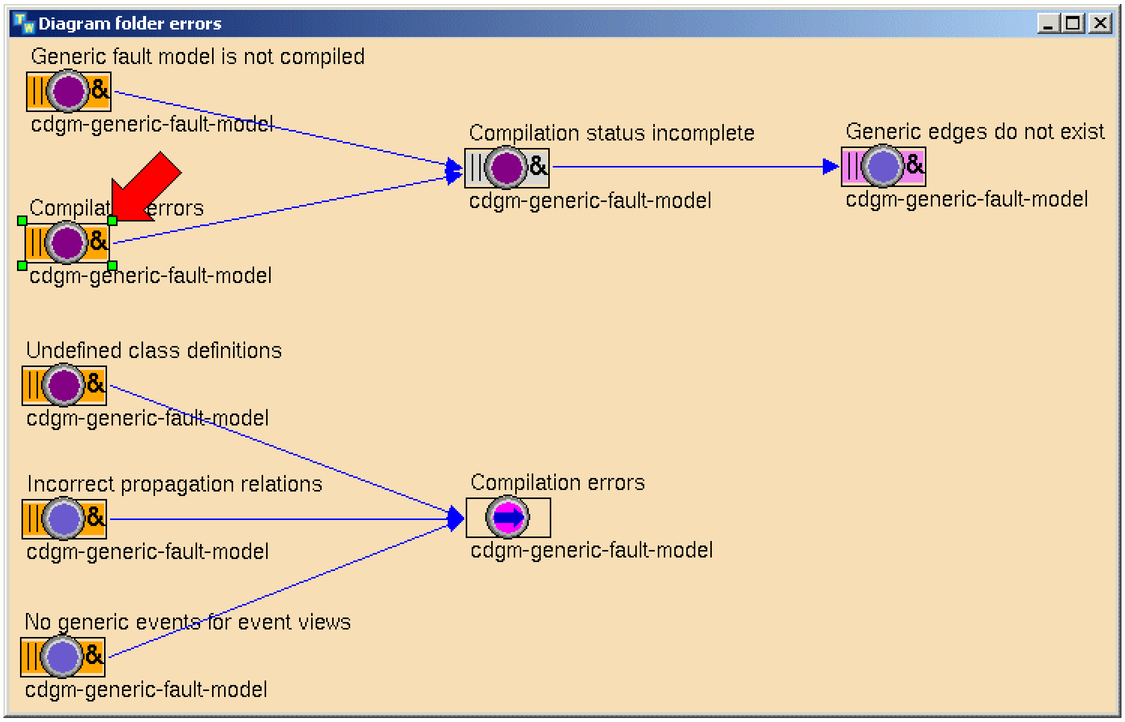
Representation of the generic event view

Representation of the generic event

Relation type between events

2 Choose Go To Generic Event on the representation of the generic event.

SymCure places an arrow next to the generic event in its associated fault model folder:



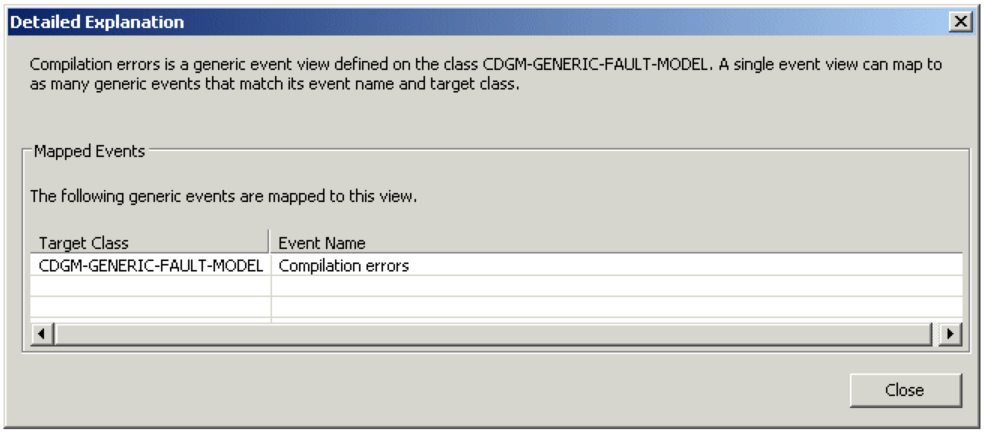
Showing Detailed Explanations of Generic Event Views

You can show detailed explanations about a generic event view, which includes information about mapped events.

To show a detailed explanation about a generic event view:

➔ Choose Detailed Explanation on the generic event view.

For example, here is the detailed explanation for a generic event view:



Compilation errors is a generic event view defined on the class CDGM-GENERIC-FAULT-MODEL. A single event view can map to as many generic events that match its event name and target class.

The following generic events are mapped to this view.

Target Class	Event Name
CDGM-GENERIC-FAULT-MODEL	Compilation errors

Configuring Causal Connections

Causal connections between events in a generic fault model define a **propagation relation**, which qualifies the causal connection. The default propagation relation self specifies propagation among events on the same domain object. The propagation relation can also be the name of any relation that exists between two domain objects in a domain map. If the specified relation exists, at run time, SymCure propagates events between the domain objects.

The propagation relation can be one of a set of built-in G2 relations. These relations define connectivity and containment relations between domain objects in a domain map. The propagation relation can also be any user-defined relation that you define between domain objects or a dynamically computed relation.

The color of the connection between two generic events or event events indicates whether the causal relationship uses the default propagation relation or a user-specified relation, as follows:

This connection color...	Indicates the propagation relation is...
Blue	The default, which is self.
Green	A built-in, user-defined, or dynamically created propagation relation.

You can create a label that displays the propagation relation next to the event. You might want to do this when configuring the propagation relation to be a value other than self, the default. The color of the label matches the color of the causal link. If you delete a connection, the label is automatically deleted. If you change the propagation relation, the label is automatically updated.

Built-In Propagation Relations

You can configure the propagation relation to be any of these built-in relations:

Propagation Relation Name	Description
self	The default propagation relation, which propagates events within the same object.
cdg-connected-upstream	A directed connection from the target class of the upstream event to the target class of the downstream event.
cdg-connected-downstream	A directed connection from the target class of the downstream event to the target class of the upstream event.
cdg-connected-to	An undirected connection between the target classes of the upstream and downstream events.
cdg-contained-in	The containment of the target class of the upstream event on the subworkspace of the target class of the downstream event.
cdg-the-container-of	The containment of the target class of the downstream event on the subworkspace of the target class of the upstream event.
cdg-the-embedded-object-of	An attribute of an object that is a subobject.
cdg-the-embedding-object-of	An object that contains an attribute that is a subobject.
cdg-virtual-relation	A dynamically computed relation.

Thus, the following relations exists between domain objects in a domain map:

- An upstream domain object is cdg-connected-upstream of a downstream domain object.
- A downstream domain object is cdg-connected-downstream of an upstream domain object.
- An upstream domain object is cdg-connected-to a downstream domain object, and a downstream domain object is also cdg-connected-to an upstream domain object.

- A domain object that exists on the subworkspace of another domain object is cdg-contained-in that domain object.
- A domain object that contains another domain object on its subworkspace is cdg-the-container-of that domain object.

To use the embedded object relations, you must either define the embedded object to be an instance of grtl-domain-object-with-key (or opt-domain-object-with-key for Optegrity applications) or a subclass, or you must provide each embedded object with a name. For more information on this class, see the *G2 Developers' Utilities Runtime Library User's Guide*.

Configuring Causal Connections by using a Built-In Propagation Relation

You use the built-in propagation relations when domain objects in a domain map are either not related or are related by either connectivity or containment:

Relation Type	Description
Self	The domain objects are not related.
Connectivity	One domain object is connected to another domain object by either a directed or an undirected connection.
Containment	One domain object is contained on the subworkspace of another domain object or one domain object is a subobject of another domain object and does not appear on a domain map.

To configure causal connections by using a built-in propagation relation:

1. Create a domain map in which one domain object is related to another by either connectivity or containment.

   For more information, see Creating Domain Maps. In this scenario, it is not necessary to define any relations between domain object classes, because containment and connectivity relations already exist.

2. Create, configure, and connect generic events and generic event views, as needed, in which the target classes of the connected events are related by either connectivity or containment.

   For details, see Creating Generic Events and Creating Generic Event Views.

3. Choose Properties on the directed connection between two events or between and event and an event view.

   The default propagation relation is self.

4 Configure the Type of Relation to be Self, Connectivity, or Containment, depending on the type of built-in relation that exists between the two domain object instances of the target classes defined for the connected events.

5 Depending on the Type of Relation, configure these additional properties:

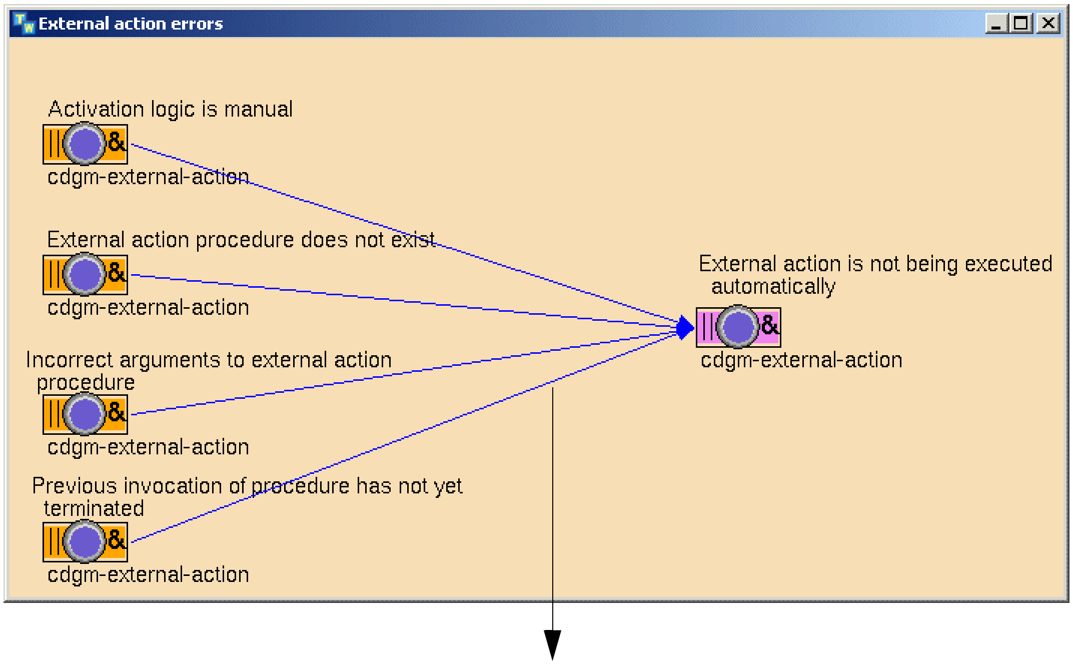
Type of Relation	Property	Value
Connectivity	Connection Direction	• cdg-connected-downstream • cdg-connected-upstream • cdg-connected-to
	Connection Class	The class of connection, whose superior class is connection.
Containment	Containment Relation	• cdg-contained-in • cdg-contained-in • cdg-an-embedded-object-of • cdg-the-embedding-object-of

6 Optionally, enable the Show Label option to display the propagation relation next to the connection.

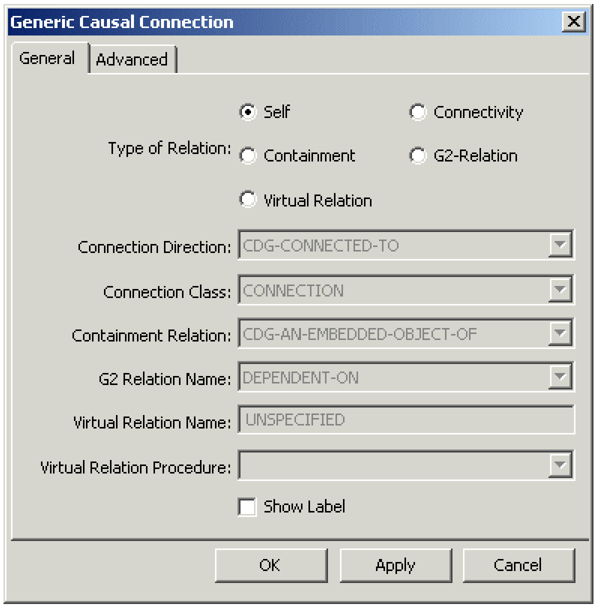
The causal relationship between the events is now defined, based on this relation. When SymCure builds the specific fault model, it propagates events if the specified propagation relation exists between specific instances of the target classes.

Example: Relation Type is Self

This example shows a connection between two generic events that use the default propagation relation, which is self:


External action errors
Activation logic is manual
cdgm-external-action
External action procedure does not exist
cdgm-external-action
Incorrect arguments to external action procedure
cdgm-external-action
Previous invocation of procedure has not yet terminated
cdgm-external-action
External action is not being executed automatically
cdgm-external-action

Type of Relation is self, the default, which propagates events within the same target class.



Type of Relation: ● Self ○ Connectivity ○ Containment ○ G2-Relation ○ Virtual Relation

Connection Direction: CDG-CONNECTED-TO

Connection Class: CONNECTION

Containment Relation: CDG-AN-EMBEDDED-OBJECT-OF

G2 Relation Name: DEPENDENT-ON

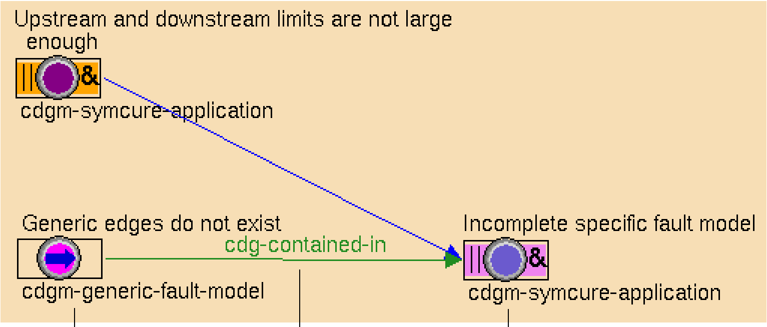
Virtual Relation Name: UNSPECIFIED

Virtual Relation Procedure:

□ Show Label

Example: Relation Type is Containment

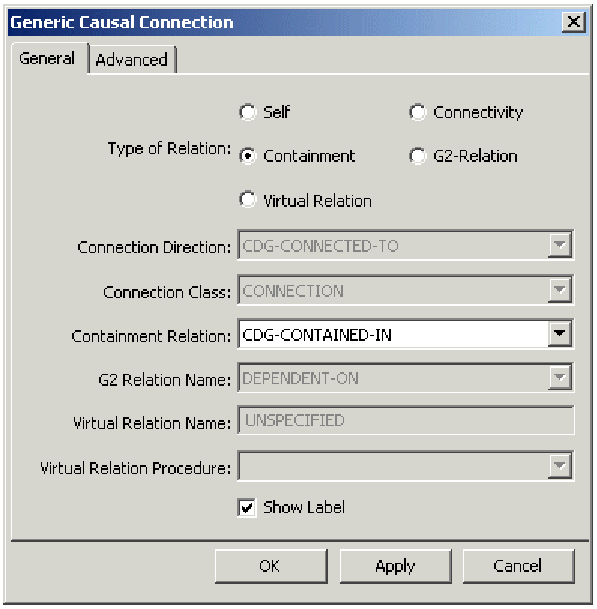
This example shows a connection between a generic event view and a generic event that uses one of the built-in propagation relations, which is cdg-contained-in. The generic event view named “Generic edges do not exist” is defined for the cdgm-generic-fault-model class. The generic event named “Incomplete specific fault model” is defined for the cdgm-symcure-application class. Notice that the connection with the built-in propagation relation is green, as opposed to blue, and it’s propagation relation is labeled.


Upstream and downstream limits are not large enough
cdgm-symcure-application
Generic edges do not exist
cdg-contained-in
cdgm-generic-fault-model
Incomplete specific fault model
cdgm-symcure-application

The generic event view target class is cdgm-generic-fault-model

The generic event target class is cdgm-symcure-application.

Propagation-relation is cdg-contained-in, which propagates events whenever an instance of cdgm-generic-fault-model is "contained in" an instance of cdgm-symcure-application.



Type of Relation: ○ Self ○ Connectivity ● Containment ○ G2-Relation ○ Virtual Relation

Connection Direction: CDG-CONNECTED-TO

Connection Class: CONNECTION

Containment Relation: CDG-CONTAINED-IN

G2 Relation Name: DEPENDENT-ON

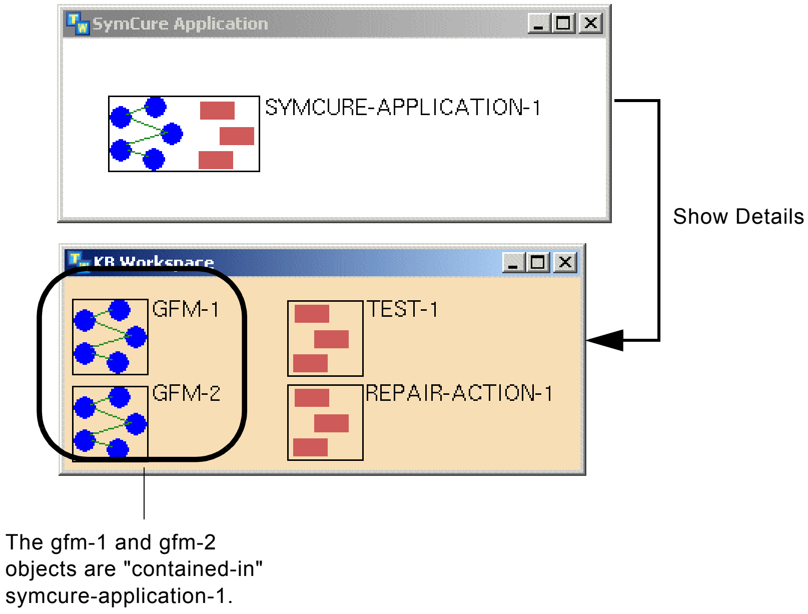
Virtual Relation Name: UNSPECIFIED

Virtual Relation Procedure:

☑ Show Label

In the domain map for the application, symcure-application-1 contains two generic fault models on its detail, gfm-1 and gfm-2.

Because gfm-1 and gfm-2 are on the detail of symcure-application-1, the propagation relation is cdg-contained-in. During diagnosis, the “Generic edges do not exist” event for both gfm-1 and gfm-2 propagates to the “Incomplete specific fault model” event for symcure-application-1.



The gfm-1 and gfm-2 objects are "contained-in" symcure-application-1.

Configuring Causal Connections by using a User-Defined Propagation Relation

In this scenario, first, you must create a relation definition between two domain object classes, then you must programmatically conclude the relation between instances of those classes.

Note Configuring user-defined propagation relations requires knowledge of G2.

To configure causal connections by using a user-defined propagation relation:

1. Create a relation definition between two classes.

   For details, see the *G2 Reference Manual*.

2. Create a domain map that uses instances of these domain object classes.

   For more information, see Creating Domain Maps.

3 Create, configure, and connect generic events and generic event views, as needed, in which the target class of the connected events are related, based on the user-defined relation.

For details, see Creating Generic Events and Creating Generic Event Views.

4 In the properties dialog for the causal link between connected generic events, configure the Type of Relation to be g2-relation.

5 Configure the G2 Relation Name to be user-defined relation name.

6 Before performing SymCure diagnosis, programmatically conclude a relation between specific domain object instances for which the relation has been defined.

For details, see the *G2 Reference Manual*.

The causal relationship between the events is now defined, based on the user-defined relation. When SymCure builds the specific fault model, it propagates events if the specified propagation relation exists between specific instances of the target classes.

Configuring Virtual Propagation Relations

SymCure allows you to specify causal connections by using a virtual propagation relation. A virtual relation can be used in highly dynamic domains for determining “on the fly” what objects are related to a target domain object, without requiring the establishment of any G2 relations, connectivity, or containment.

To use a virtual propagation relations, you provide a name for the virtual relation and you write a procedure or method to compute related objects at run time.

SymCure provides the following classes of user-defined procedures/methods for computing virtual relations, which are available from the User-Defined Methods and Procedures palette:

- cdg-virtual-relation-computation-procedure
- cdg-virtual-relation-computation-method

The signatures for the procedure or method are:

my-virtual-relation-computation-procedure
(*Target*: class grtl-domain-object, *VirualRelationName*: symbol,
DirectionOfPropagation: symbol)
-> *RelatedDomainObjects*: sequence

Note Configuring virtual propagation relations requires knowledge of G2.

To configure causal connections by using a virtual propagation relation:

1 Create a domain map that uses instances of these domain object classes.

For more information, see Creating Domain Maps.

2 Create, configure, and connect generic events and generic event views, as needed, in which the target class of the connected events are related, based on the virtual relation.

For details, see Creating Generic Events and Creating Generic Event Views.

3 Create a Virtual Relation Computation Method or Procedure from the User-Defined Procedures and Methods palette of the Fault Models toolbox, and configure the text of the method or procedure to compute the virtual relation.

4 In the properties dialog for the causal link between connected generic events, configure the Type of Relation to be virtual relation.

5 Configure the Virtual Relation Name to be a symbol to use as the virtual relation name.

6 Configure the Virtual Relation Procedure to be the method or procedure you created above.

The causal relationship between the events is now defined, based on the virtual relation. When SymCure builds the specific fault model, it computes the virtual relation and propagates events if the virtual propagation relation exists between specific instances of the target classes.

Configuring Propagation Delays

Causal connections define the Propagation Delay attribute, which allows you to model the delay between each generic cause and its effect. Propagation delays provide a foundation for computing the inferred time of an occurrence while propagating events in the specific fault model. The use of a propagation delay enables SymCure to construct a suitable message that will provide meaningful advance notice to the operator about predicted events.

In addition, SymCure calculates the Inferred Occurrence Time of specific events, which represents the time at which an event is inferred to be true. This is distinct from the time stamp of the event.

For example, consider the following scenario. If the flame of a furnace is extinguished, then oxygen builds up in the furnace. If oxygen is allowed to build up indefinitely, it can cause a furnace explosion. This scenario is modeled as follows: "Flameout" -> "Oxygen buildup" -> "Explosion". For this model to hold true, we must implicitly assume a propagation delay for the causal relationship

"Oxygen buildup" -> "Explosion". Now suppose that we infer that "Flameout" is true. What useful information can we provide to an operator? The model will conclude that "Flameout" ultimately leads to "Explosion", but without explicitly representing the propagation delay along the causal relationships, it simply cannot say whether the explosion has already occurred or, more importantly, when the explosion is likely to occur. This lack of information diminishes the utility of the model's predictive capabilities and could be misunderstood as "crying wolf", namely, predicting that an event has occurred when in fact it may be hours away and, if the underlying problem is treated in time, will never happen.

To provide another example, consider a root cause event that becomes suspect at time t0 because some of its symptoms are manifested. Suppose that it takes a finite amount of time to test the root cause, and an affirmative test result is returned at time t_1 ($t_1 > t_0$). The value of the event is set to true and its timestamp is set to t_1 — the time at which the event becomes true. However, in all likelihood, the event must have been true before t_1 to cause the symptoms at t_0. The timestamp can lead to the misinterpretation that the root cause became true only at t_1. In this example, the inferred occurrence time of the root cause event would be t_0 while its timestamp is t_1.

Note We believe that propagation delays are inherently approximations and should be used for informational purposes only. They do not in any manner impact the values computed by diagnostic propagation algorithm and are not required by the model. If, however, such delays can be modeled with desired accuracy, not only will they provide vital information to operators about predicted events, but they may aid the diagnostic process by pinpointing the respective times at which the set of candidate root causes must have occurred to explain the known symptoms. Such information may be invaluable in validating the candidate root causes.

When specifying a propagation delay, you can specify a procedure for computing the propagation delay. For example, consider a simple model where "Leak" in TANK causes "Empty" on TANK, that is, a leak in the tank causes the tank to become empty. The propagation delay between these two events depends on the volume of fluid in the tank and the rate at which fluid flows out of the tank.

SymCure allows you to specify the name of a cdg-causal-propagation-delay-computation procedure or method with the following signature:

cdg-compute-causal-propagation-delay-procedure
(*UpstreamEventTarget*: class opt-domain-object,
DownstreamEventTarget: class opt-domain-object,
DefaultPropagationDelay: quantity
-> *propagation-delay*: quantity

Returns the propagation delay between *UpstreamEventTarget* and *DownstreamEventTarget*.

If such a procedure is specified on a causal link, it is used to compute the propagation delay; otherwise, by default, it uses the propagation delay specified in the causal connection.

Configuring a Propagation Delay

To configure a propagation delay:

1. Display the properties dialog for a causal connection and click the Advanced tab.
2. Configure the Propagation Delay or configure the Compute Delay Procedure to be a user-defined cdg-causal-propagation-delay-computation procedure or method.

You can create the procedures from the palettes by choosing View > Toolbox - Fault Models > User-Defined Procedures and Methods and creating a Causal Propagation Delay Computation Method or Procedure. The procedure’s signature is automatically populated from its signature attribute. You can configure the procedure for the causal connection by choosing from a list of such procedures in the properties dialog for the causal connection.

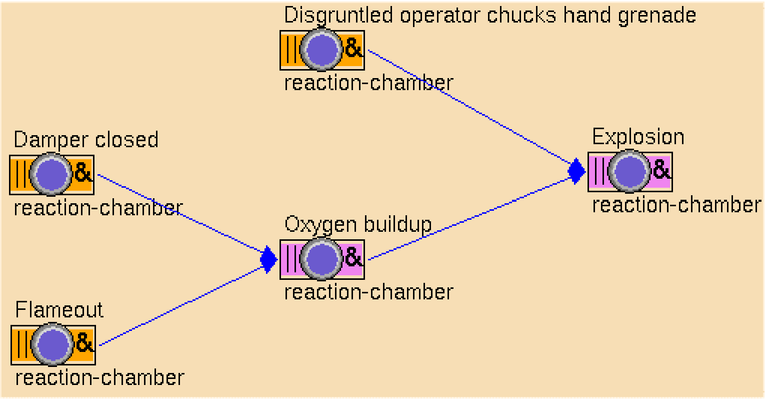
Example: Specifying Generic Propagation Delays

The following hypothetical generic fault model has the following propagation delays:

- 30 seconds

  Disgruntled operator chucks hand grenade -> Explosion

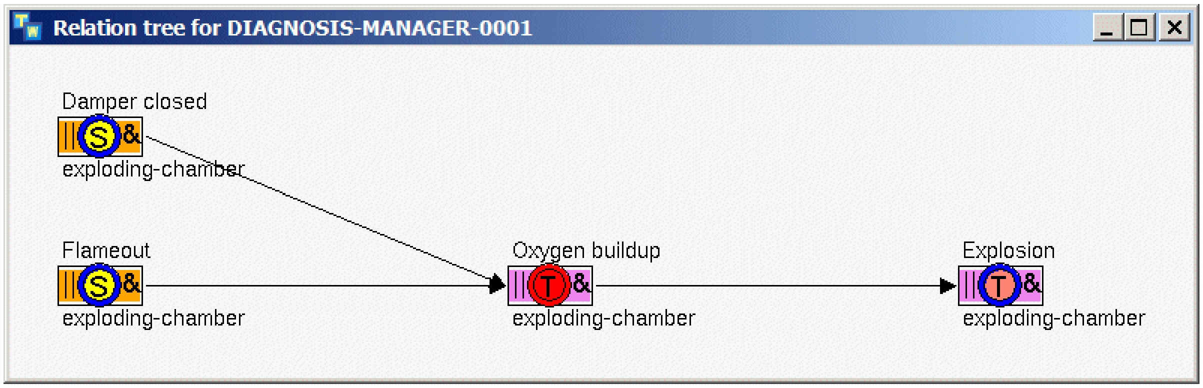
- I hour
  - Oxygen buildup -> Explosion
  - Damper closed -> Oxygen buildup
  - Flameout -> Oxygen buildup



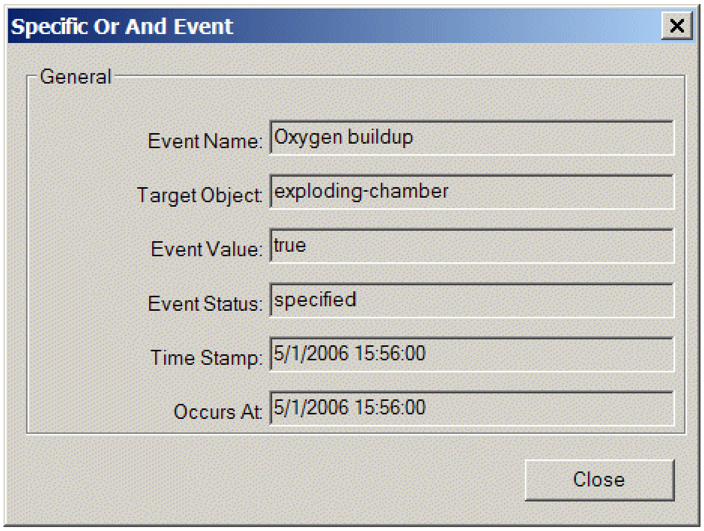
Example: Calculating Inferred Time of Occurrence During Propagation

During upstream propagation, SymCure computes the Inferred Occurrence Time for a specific event by subtracting the propagation delay of the causal link from the effect's inferred time of occurrence. During downstream propagation, it computes the inferred time of occurrence for an event by adding the propagation delay of the causal link to the cause's inferred time of occurrence.

Consider the following example. When "Oxygen buildup" is specified to be true, SymCure generates the following specific fault model:



The timestamp of "Oxygen buildup" is same as its inferred occurrence time ("Occurs At"):

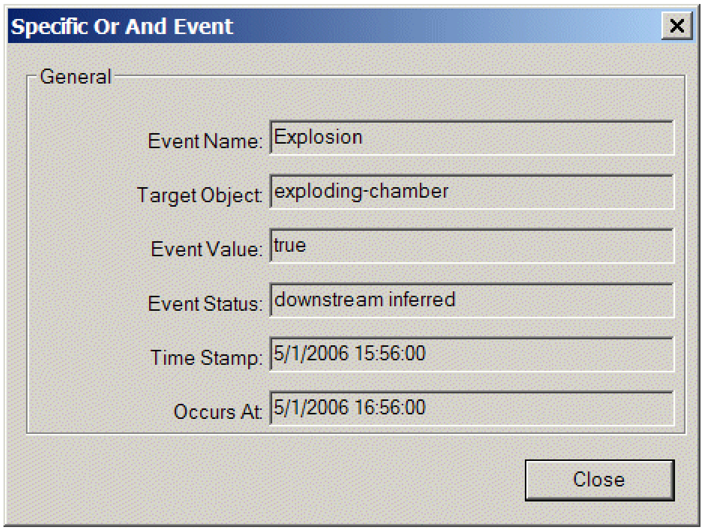


Specific Or And Event

General

Event Name:	Oxygen buildup
Target Object:	exploding-chamber
Event Value:	true
Event Status:	specified
Time Stamp:	5/1/2006 15:56:00
Occurs At:	5/1/2006 15:56:00

"Explosion" is expected to occur 30 minutes after "Oxygen buildup" as shown below. Note the difference between Time Stamp and Occurs At for "Explosion".


Specific Or And Event

General

Event Name: Explosion

Target Object: exploding-chamber

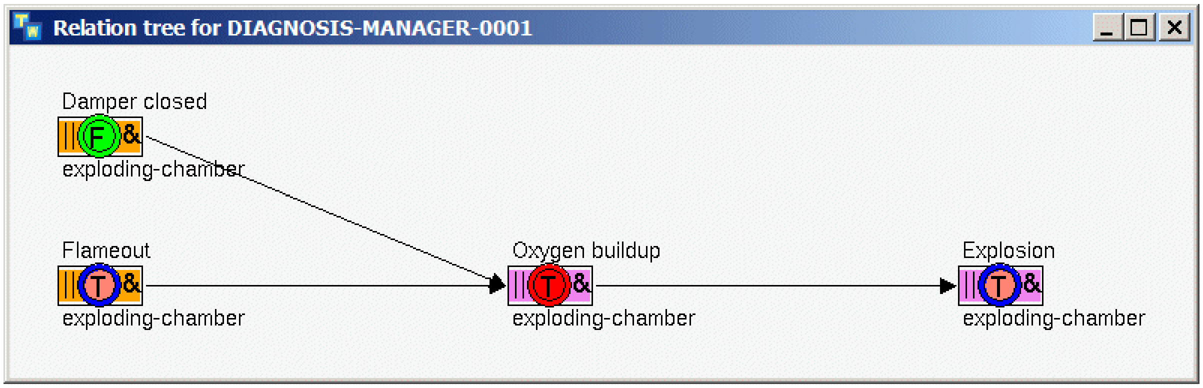
Event Value: true

Event Status: downstream inferred

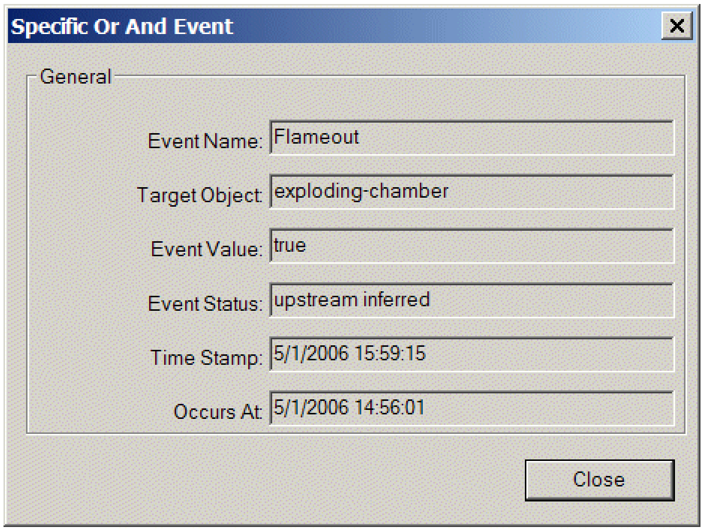
Time Stamp: 5/1/2006 15:56:00

Occurs At: 5/1/2006 16:56:00

After "Damper closed" is reported to be false, SymCure concludes that "Flameout" is the root cause of "Oxygen buildup":



Note that "Flameout" is deemed to have occurred one hour prior to "Oxygen buildup", and again note the difference between the time stamp and the time of occurrence.



Specific Or And Event

General

Event Name: Flameout

Target Object: exploding-chamber

Event Value: true

Event Status: upstream inferred

Time Stamp: 5/1/2006 15:59:15

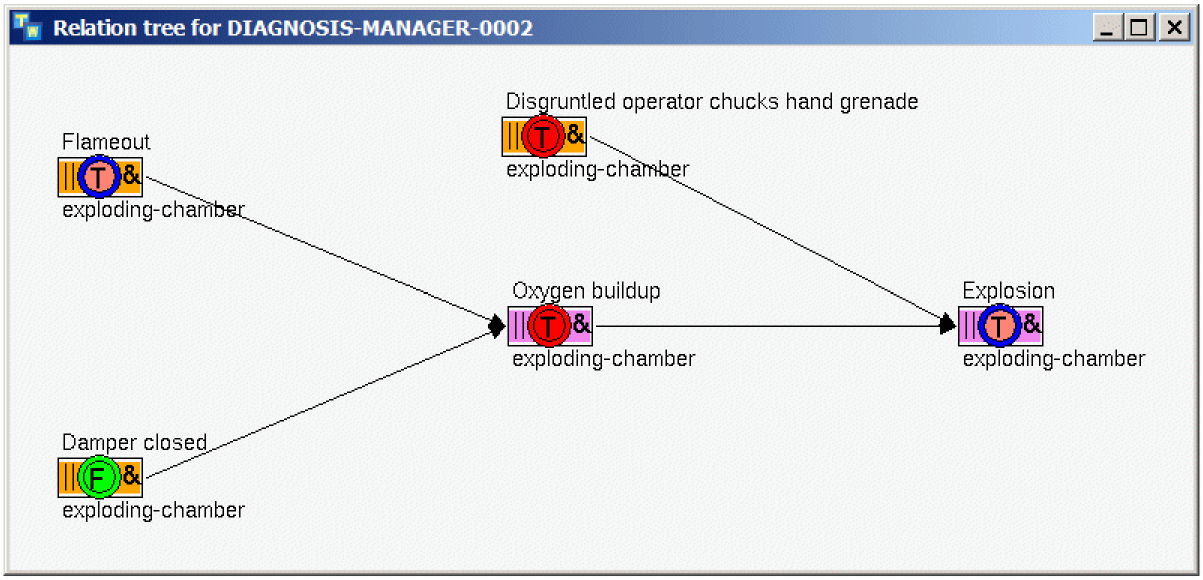
Occurs At: 5/1/2006 14:56:01

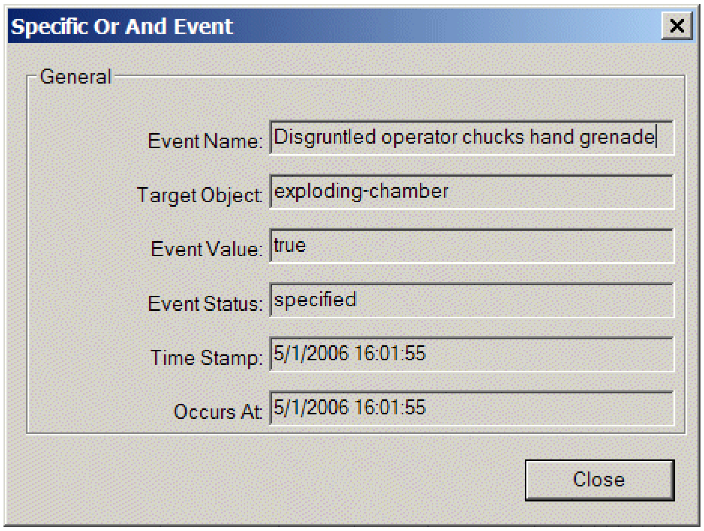
Note that the Occurs At field is shown only for specific events that are true. By definition, a false, suspect, or unknown event is not considered to have "occurred".

To use the inferred occurrence time in a message configuration, use $INFERRED-OCCURRENCE-TIME as the substitution pattern. SymCure formats the inferred occurrence time as specified by the date and time format in `config.txt` while constructing messages displayed in the operator message browser.

Limitations

Inferred time of occurrence is not updated unless the event’s value is changed. This is necessary to avoid expensive recalculation of the occurrence times of events. As a consequence, it is possible that on occasion the occurrence times of events may not accurately reflect the propagation delays along the causal edges. Consider, for example, that while the above diagnosis is in progress, a disgruntled operator is seen to drop a grenade into the reaction chamber, resulting in the following specific fault model.




Specific Or And Event

General

Event Name: Disgruntled operator chucks hand grenade

Target Object: exploding-chamber

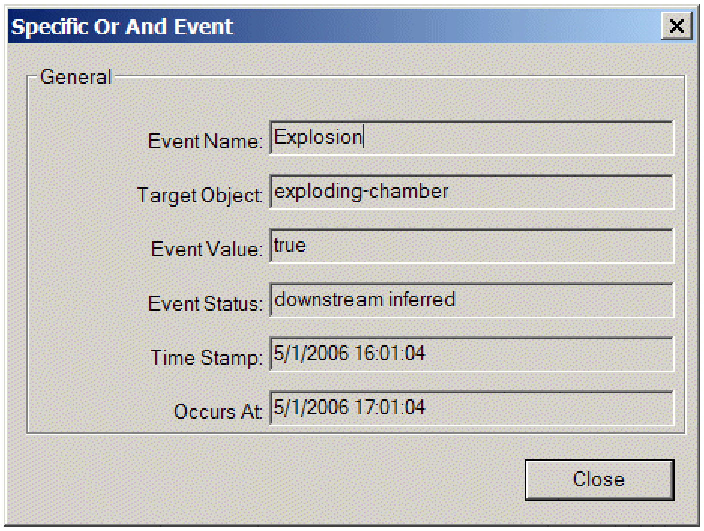
Event Value: true

Event Status: specified

Time Stamp: 5/1/2006 16:01:55

Occurs At: 5/1/2006 16:01:55

Because "Explosion" stays true, there is no downstream propagation of "Disgruntled operator chucks hand grenade". Now even though a grenade can cause an explosion much faster than the "Oxygen buildup" (in 30 seconds according to our model), the time of occurrence for "Explosion" is unchanged.



Event Name: Explosion

Target Object: exploding-chamber

Event Value: true

Event Status: downstream inferred

Time Stamp: 5/1/2006 16:01:04

Occurs At: 5/1/2006 17:01:04

While this example clearly demonstrates a fundamental limitation of SymCure's temporal reasoning capability, we believe that in practice this is not a serious flaw. It is unlikely that while one set of root causes associated with an event is being explored, a new and completely independent set of root causes for the same event will arise simultaneously.

There is a practical problem with synchronizing the inferred occurrence time. Suppose we do recalculate the occurrence time for "Explosion" based on the occurrence of "Disgruntled operator chucks hand grenade". If there were any events downstream of "Explosion", we would need to update them as well. Now suppose that we learn that the object lobbed into the reaction chamber by the disgruntled operator was not a grenade, but an orange, so "Disgruntled operator chucks hand grenade" is false. This would require recalculating the occurrence time for "Explosion" and propagating it once more, by recognizing that it can still be caused by "Flameout". In general, such recalculations and repeated propagations will prove to be extremely expensive and will fundamentally compromise the efficiency and efficacy of the diagnostic propagation algorithm.

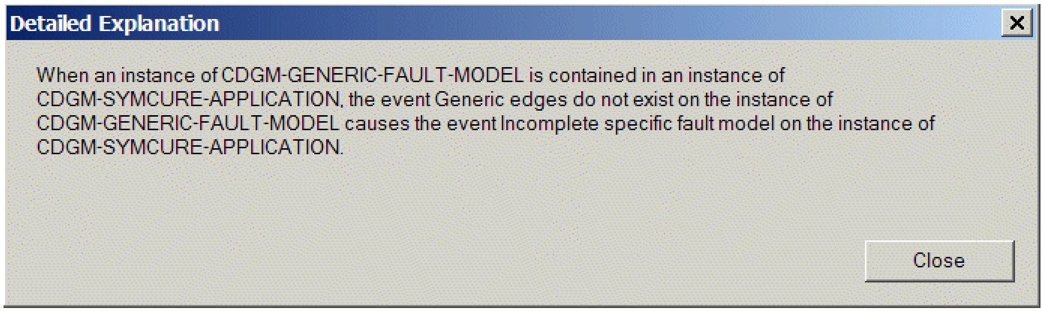
Showing Detailed Explanations of Causal Connections

You can show detailed explanations about a causal connection, which provides the meaning of the propagation relation.

To show a detailed explanation about a causal connection:

➔ Choose Detailed Explanation on the causal connection.

For example, here is the detailed explanation for a causal connection:



When an instance of CDGM-GENERIC-FAULT-MODEL is contained in an instance of CDGM-SYMCURE-APPLICATION, the event Generic edges do not exist on the instance of CDGM-GENERIC-FAULT-MODEL causes the event Incomplete specific fault model on the instance of CDGM-SYMCURE-APPLICATION.

Creating Generic External Actions

External actions, which include tests and repair procedures are procedural components of SymCure's fault management capabilities. As they are extrinsic to events that form the heart of the SymCure fault models, hence the term external actions. This distinguishes them from the intrinsic event changed and event unchanged procedures associated with fault model events.

External actions have properties that you can configure, such as durations and costs, and they might require resource allocation and scheduling for optimal performance. When the value of the underlying event changes, then an external action may be enabled or disabled according to its configuration. You can schedule external actions to execute either manually or automatically, based on event transitions.

You can interact with external actions through various built-in browsers. You can execute manual tests, obtain explanations about why the action was enabled or executed, and go to the specific event that caused the action to be scheduled.

Creating the Activation Procedure

Each generic action has an associated **activation procedure**, which is typically a G2 procedure that performs the external action. You can also implement the procedure by using one of Gensym's graphical block languages, for example, Gensym Event and Data Processing (GEDP), which is available with Optegrity, or Operator Actions (OPAC), which is available with Integrity.

You can call a number of SymCure API procedures in the activation procedure. These procedures provide programmatic access to numerous SymCure features, such as sending events, getting root causes, and so on.

The procedure for activating an external action is the same for all types of generic actions, except that the activation procedure of a test action must send a value for the underlying event, via an API call.

You can create the procedure from the palettes by choosing View > Toolbox - Fault Models > User-Defined Procedures and Methods and creating a Generic Action Activation Procedure. The procedure's signature is automatically populated from its signature attribute. You can configure the procedure for the generic action by choosing from a list of such procedures in the properties dialog for the generic action.

The activation procedure must have the following signature:

```
my-activation-proc(Target: class grtl-domain-object,
    SpecificAction: class cdg-specific-action,
    TriggeringEvent: item-or-value, AssociatedEvents: sequence,
    TimeStamp: integer, Client: class object)
```

where:

- *Target* is an instance of the target class defined for the generic action.
- *SpecificAction* is the specific action associated with the target object.
- *TriggeringEvent* represents the event that is responsible for triggering the action, when the action is activated automatically. If the action is activated manually, *TriggeringEvent* is the symbol none.
- *AssociatedEvents* is a sequence of all specific events associated with the specific action, when associating a single specific action with multiple specific events. The order of events in the sequence *AssociatedEvents* is completely arbitrary.
- *TimeStamp* is the time at which *TriggeringEvent* changes its value, in seconds. The timestamp might be necessary if the value of the event changes during the execution of the procedure. This can occur when many

events occur rapidly, because the activation procedure runs in a different thread.

- *Client* is the G2 window on which to display any data while executing the action.

A single specific action can be associated with multiple specific events. The events associated with the action are available from the *AssociatedEvents* sequence. If there is only one event associated with the action and the action is executed automatically, then *TriggeringEvent* = *AssociatedEvents*[0]. Otherwise, for any automatic invocation of the action, *TriggeringEvent* is a member of *AssociatedEvents*.

Note If an action is associated with multiple events, the activation procedure should perform the necessary steps on each specific event in the *AssociatedEvents* sequence.

Identifying the *TriggeringEvent* allows the activation procedure some flexibility. An automatic action that services multiple events may be triggered by any one of them. You can program the activation procedure to respond to all of its associated events, or just the triggering event, depending on your requirements.

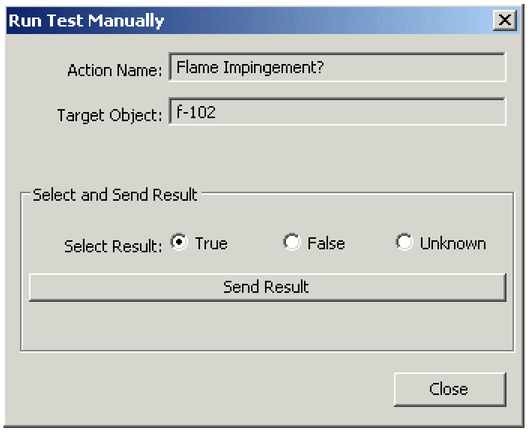
If the activation procedure of the generic event does not refer to an existing G2 procedure, the action looks for a GEDP diagram with the same name as the action and with the same target class. If such a diagram exists, the action uses that as the activation procedure. The GEDP diagram must use the Return block to return a value to its GEDP diagram folder. Also, ensure that the Activated option is disabled for the specific diagram or generic diagram template to deactivate the diagram after the event occurs. For more information, see the *G2 Event and Data Processing User's Guide*.

For more information about API procedures that you can call in activation procedures, see Application Programmer's Interface.

Running Tests Manually

SymCure provides a built-in activation procedure named cdg-default-run-test-manually-procedure, which displays a dialog that allows you to manually select "true", "false", or "unknown" as a result for the test. Once this procedure is assigned to a generic action, you can simply select the Execute Action button in the browser to display the dialog to assign a suitable result for the test.

For example, here is the Run Test Manually dialog that appears for the Flame Impingement? test action:



Run Test Manually

Action Name: Flame Impingement?

Target Object: f-102

Select and Send Result

Select Result: ● True ○ False ○ Unknown

Scheduling External Actions

You can associate generic actions with one or more generic events. An action is activated when the value of an associated event changes, depending on the type of external action. This change in the associated event's value, which enables the action to be activated, is called an **enabling transition** of the action.

For example, a test action activates when the value of the underlying event changes from any value to "suspect" or "unknown", indicating that the event requires further information to complete the diagnosis. By contrast, a repair action is activated when the value of the underlying event changes from any value to "true", indicating that the event, typically a root cause, is known to be true and, therefore, external intervention is required to repair the root cause.

When the action is activated, it executes its activation procedure either manually or automatically, depending on its **activation type**. Manual execution requires operator intervention, whereas automatic execution does not. The default activation type is automatic.

For information on customizing the scheduling of external actions, see Customizing the Scheduling of External Actions.

Types of Enabling Transitions

The enabling transition specifies when the action is activated, based on the change in value of the underlying event. The enabling transition can be one or more of the following options:

- Any to Any — Event changes from any value to any other value.
- Any to True — Event changes from any other value to "true".

- Any to False — Event changes from any other value to "false".
- Any to Suspect — Event changes from any other value to "suspect".
- Any to Unknown — Event changes from any other value to "unknown".
- Suspect to True — Event changes from "suspect" to "true".
- Unknown to True — Event changes from "unknown" to "true".
- True to False — Event changes from "true" to "false".
- Suspect to False — Event changes from "suspect" to "false".
- Unknown to False — Event changes from "unknown" to "false".
- True to Suspect — Event changes from "true" to "suspect".
- False to Suspect — Event changes from "false" to "suspect".
- Unknown to Suspect — Event changes from "unknown" to "suspect".
- True to Unknown — Event changes from "true" to "unknown".
- False to Unknown — Event changes from "false" to "unknown".
- Suspect to Unknown — Event changes from "suspect" to "unknown".

In most cases, you can use the default enabling transition for each type of generic action, as follows:

External Action Type	Default Enabling Transition
Generic test action	Any to Suspect, Any to Unknown
Generic repair action	Any to True
Generic action	Any to Any
Generic mitigation action	Any to Suspect
Generic recovery action	Any to False

Types of External Actions

The two basic types of external actions are:

- Test action, which tests for the occurrence of a specific event and returns the value of the event via an API call to SymCure. See the description of cdg-send-event in Sending Events.
- Repair action, which performs some type of repair to the domain objects.

In addition to test and repair actions, SymCure provides a generic action, which triggers whenever the value of an event changes. For backward compatibility, SymCure also provides mitigation actions, which are enabled when the underlying event becomes suspect, and recovery actions which are enabled when the underlying event changes from true to false. We recommend that you use the repair actions instead of mitigation and recovery actions by simply configuring the enabling transitions. These actions are located on the Legacy Items palette of the Fault Models toolbox.

This table shows the icon for each type of external action:

Icon	External Action
?	Test action
!	Repair action
*	Generic action
^	Recovery action
#	Mitigation action

Creating and Configuring Generic External Actions

To create an external action, you:

- Configure the properties of the external action.
- Associate the external action with a generic event.

You define generic external actions for a domain object class by placing them on the subworkspace of a generic fault model folder.

Note Generic actions that do not appear on the subworkspace of a fault model folder are automatically deleted.

You can associate one or more generic events with a single generic action. You choose from a list of available events, which include all generic events whose target class matches the target class of the generic action.

External actions and their associated generic events can reside in different fault model folders and even in different modules. This allows you to build a library of fault models for domain objects in one module and customize it with external actions relevant for particular applications in other modules.

Note If you change the name of a generic action, SymCure automatically updates any GEDP Send Fault Model Action Result blocks that refer to the event.

Creating an External Action and Configuring its Properties

To create an external action and configure its properties:

1 Create a generic fault model folder to contain the generic actions.

For details, see Creating Fault Model Folders.

2 Create a generic action, generic test action, generic repair action, generic mitigation action, or generic recovery action from the SymCure palette and place it on the fault model folder detail.

3 Choose Properties on the generic action and on the General tab, configure the Action Name to be a string, which appears next to the generic action.

4 Configure the Target Class to be the domain object class to which the action applies, which must be a subclass of grtl-domain-object.

5 Configure the Type to be manual or automatic, depending on whether the generic action should execute automatically upon activation or whether it should require operator intervention.

6 Configure the Procedure to be the name of a G2 procedure that specifies the action to perform.

You can create the procedure from the palettes by choosing View > Toolbox - Fault Models > User-Defined Procedures and Methods and creating a Generic Action Activation Procedure. The procedure’s signature is automatically populated from its signature attribute. You can configure the procedure for the generic action by choosing from a list of such procedures in the properties dialog for the generic action.

For more information, see Creating the Activation Procedure.

7 Configure the Estimated Duration to be the maximum duration of the action, in seconds.

The default is 0. Specify the duration to be a non-zero value to simulate a finite delay between the start and end of an external action, when the activation procedure for the external action has a wait statement.

8 Configure the Cost and Reliability to be a measure of the cost and reliability of executing the action, as needed.

For information on how to use the cost and reliability to schedule external actions, see Customizing the Scheduling of External Actions.

9. To override the default enabling transition for the action, click the Advanced tab, then click the Enabling Transitions button and click the enabling transitions that should activate the action.

   For details, see Scheduling External Actions and Types of Enabling Transitions.

Associating Generic External Actions with Generic Events

To associate a generic external action with a generic event:

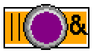
1. Choose Associate Events on a generic external action.

   You can also click the Associate Events button on the Advanced tab of the generic external actions properties dialog.

   A dialog that lists all generic events associated with the target class of the generic action appears. By default, all generic events for the target class are unrelated to the generic action.

2. Select one or more generic events from the list of unrelated events and move them to the list of related events for the generic action, then click OK.

The center of the generic event icon turns a darker shade of purple to indicate it has an associated external action:



For an example, see Example: Generic Repair Action.

Showing Related Generic Actions and Generic Events

You can show the related generic events of a generic action, and you can show the related generic actions of a generic event.

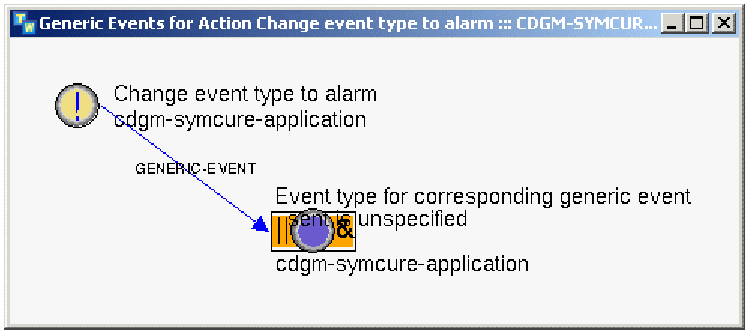
To show the related generic events of a generic action:

➔ Choose Show Generic Events on a generic action.

To show the related generic actions of a generic event:

➔ Choose Show Related Objects on a generic event.

Here is the result of either menu choice:


Generic Events for Action Change event type to alarm ::: CDGM-SYMCUR...
Change event type to alarm
cdgm-symcure-application
GENERIC-EVENT
Event type for corresponding generic event sent is unspecified
&
cdgm-symcure-application

Showing the Activation Procedure

You can go to the activation procedure from a generic action, using a menu choice.

To go to the activation procedure:

➔ Choose Go to Activation Procedure on a generic action.

This menu choice is only available if the activation procedure is specified for the generic action. SymCure places an arrow next to the specified procedure, if one is specified.

Configuring Operator Messages for Generic Actions

You might want to generate an operator message when a generic action occurs, just as you do with alarms and root causes.

To customize the operator message and to provide additional information to the operator, you can configure the generic action to generate messages when the action is created, enabled, running, and/or inactive. By default, operator messages are disabled for all generic action status values. You must explicitly enable message generation for one or more status values.

Messages for generic actions appear in the default Messages Browser.

You configure the same properties for generic action messages as you do for generic event messages. For details, see Configuring Operator Messages for Generic Events.

The default message priority for generic action messages is 5 for all generic action status values: created, enabled, running, and inactive.

You can use the following substitutions for configuring a generic action message:

- $ACTION-NAME
- $TARGET-OBJECT

- $ACTION-STATUS
- $TAG
- $START-TIME
- $END-TIME
- $UNDERLYING-ROOT-CAUSES
- $RESULT
- $COST
- $ESTIMATED-DURATION
- $HISTORY

To configure operator messages for generic actions:

1 Choose Configure Messages on the generic action.

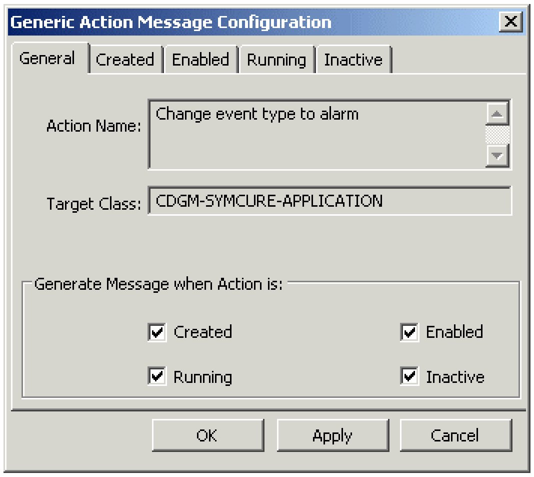
You can also click the Configure Messages button on the Advanced tab of the generic external actions properties dialog.

2 On the General tab, configure the Generate Message When Action Is to be created, enabled, running, and/or inactive.

SymCure generates messages when the value of the specific event becomes any one of the specified values.

3 Click the Created, Enabled, Running, and Inactive tabs for the status values whose message you want to configure, and configure the Message, Details, and/or Advice to be any text, including text substitutions.

Here is the General tab of the message configuration dialog for a generic action that generates messages only when the status of the underlying action is created, enabled, running, and inactive:



Action Name: Change event type to alarm

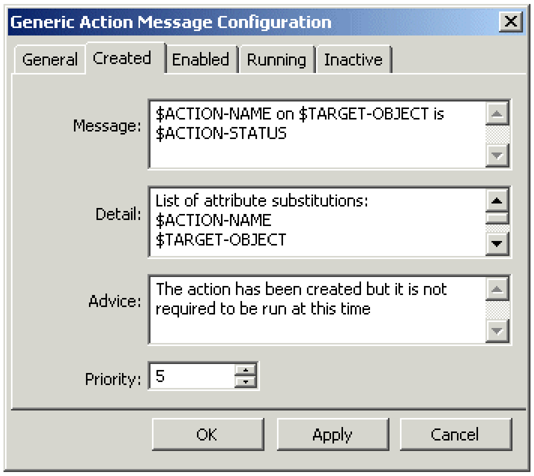
Target Class: CDGM-SYMCURE-APPLICATION

Generate Message when Action is:

☑ Created ☑ Enabled

☑ Running ☑ Inactive

Here is the default message configuration of the Created tab:



Message: $ACTION-NAME on $TARGET-OBJECT is $ACTION-STATUS

Detail: List of attribute substitutions:
$ACTION-NAME
$TARGET-OBJECT

Advice: The action has been created but it is not required to be run at this time

Priority: 5

Customizing the Scheduling of External Actions

Generic external actions provide attributes named Cost and Reliability, which you can use to schedule execution in some desirable order. To schedule actions, you create a user-defined procedure that implements a customized scheduling algorithm, using the cost and reliability attributes of the specific action, as needed.

You can create the procedure from the palettes by choosing View > Toolbox - Fault Models > User-Defined Procedures and Methods and creating a User-Defined Scheduling Procedure. The procedure's signature is automatically populated from its signature attribute.

The signature for this procedure is:

my-specific-action-scheduling-procedure
(*SpecificAction*: class cdg-specific-action,
SpecificEvent: class cdg-specific-event, *Client*: class ui-client-item)

SpecificAction is the specific action to schedule. *SpecificEvent* is the triggering event. *Client* is the client window.

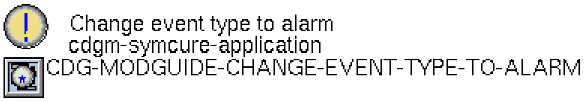
To use your procedure to schedule external actions, configure the cdg-user-defined-scheduling-procedure parameter in the configuration file, for example:

```
cdg-user-defined-scheduling-procedure=my-specific-action-scheduling-procedure
```

For more information, see Specific Action Scheduling.

Example: Generic Repair Action

Here is an example of a repair action named “Change event type to alarm”, which is associated with an activation procedure named cdg-modguide-change-event-type-to-alarm:

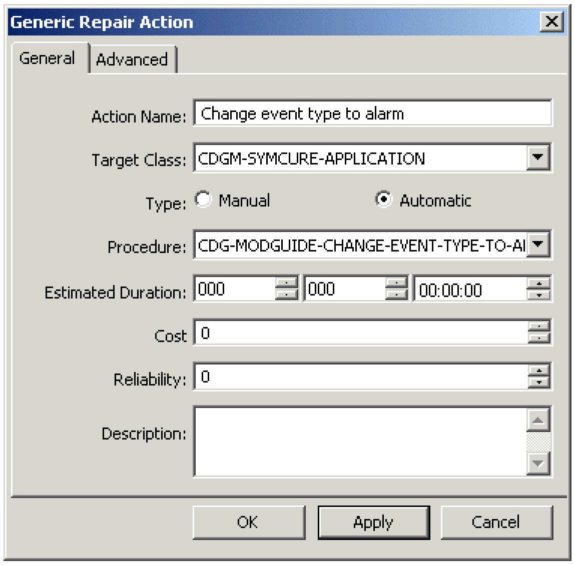


Change event type to alarm
cdgm-symcure-application
CDG-MODGUIDE-CHANGE-EVENT-TYPE-TO-ALARM

Here is the text of the procedure, which simply posts a message to the G2 Message Board:

```
cdg-modguide-change-event-type-to-alarm(Target: class cdgm-generic-fault-model,
SpecificAction: class cdg-specific-action, TriggeringEvent: item-or-value,
AssociatedEvents: sequence, TimeStamp: integer, Client: class object)
SpecificEvent: class cdg-specific-event;
begin
    for SpecificEvent = each cdg-specific-event in AssociatedEvents do
        post "REPAIR ACTION: Change the event type of [the event-name of
        SpecificEvent] on [the class of Target] to alarm.";
    end;
end
```

Here is the properties dialog for the repair action, which specifies the action name, activation procedure, and target class. It uses the default values for cost, reliability, activation type, and enabling transitions. This means the action executes automatically when the value of the underlying event changes from any value to true.



Action Name: Change event type to alarm

Target Class: CDGM-SYMCURE-APPLICATION

Type: ○ Manual ● Automatic

Procedure: CDG-MODGUIDE-CHANGE-EVENT-TYPE-TO-Al

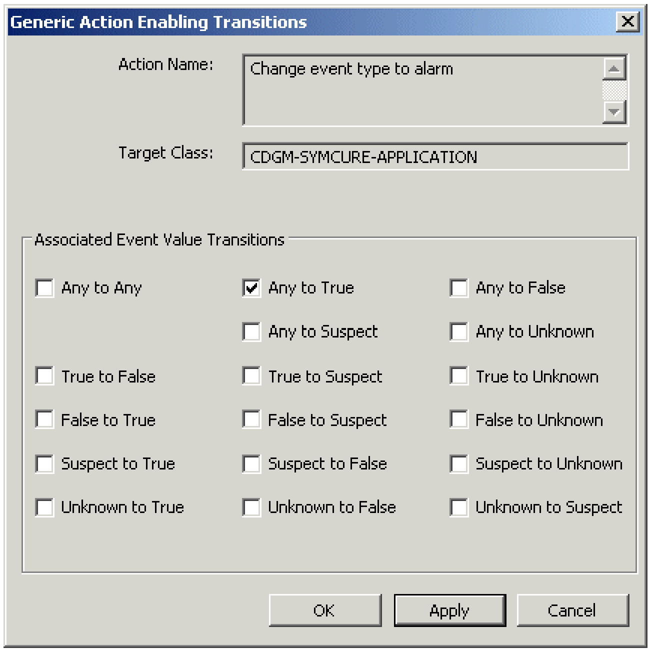
Estimated Duration: 000 000 00:00:00

Cost 0

Reliability: 0

Description:

The generic repair action uses the default enabling transitions, which is Any to True:



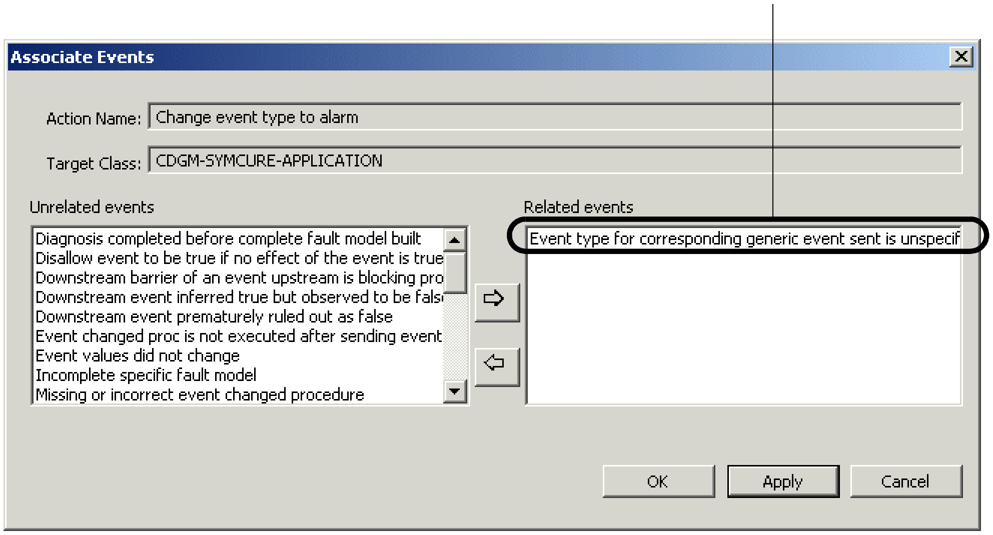
Action Name: Change event type to alarm

Target Class: CDGM-SYMCURE-APPLICATION

Associated Event Value Transitions

□ Any to Any	☑ Any to True	□ Any to False
	□ Any to Suspect	□ Any to Unknown
□ True to False	□ True to Suspect	□ True to Unknown
□ False to True	□ False to Suspect	□ False to Unknown
□ Suspect to True	□ Suspect to False	□ Suspect to Unknown
□ Unknown to True	□ Unknown to False	□ Unknown to Suspect

Here is the dialog that appears when associating events for the “Check compilation status and errors for fault model”, whose target class is cdgm-generic-fault-model. Notice that two generic events are associated with this generic action: “Compilation errors” and “Undefined class definitions”, both of which are defined for the cdgm-generic-fault-model class.



The action activates whenever the value of this event changes to true.

Action Name: Change event type to alarm

Target Class: CDGM-SYMCURE-APPLICATION

Unrelated events

- Diagnosis completed before complete fault model built
- Disallow event to be true if no effect of the event is true
- Downstream barrier of an event upstream is blocking pro
- Downstream event inferred true but observed to be fals
- Downstream event prematurely ruled out as false
- Event changed proc is not executed after sending event
- Event values did not change
- Incomplete specific fault model
- Missing or incorrect event changed procedure

Related events

- Event type for corresponding generic event sent is unspecif

Going to Generic Event-Detection Diagrams

SymCure generic events allow you to navigate to generic event-detection diagrams that contain a Send Fault Model Action Result block that refers to the generic action. Similarly, the GEDP Send Fault Model Action Result block provides the Show Fault Model Action menu choice for navigating to the generic action in the fault model.

For more information, see the *G2 Event and Data Processing User’s Guide*.

To go to generic event-detection diagrams:

➔ Choose Show Event Detection Diagrams on a generic event.

This menu choice only appears if the generic action is specified in a generic event-detection diagram.

Showing Detailed Explanations of Generic Actions

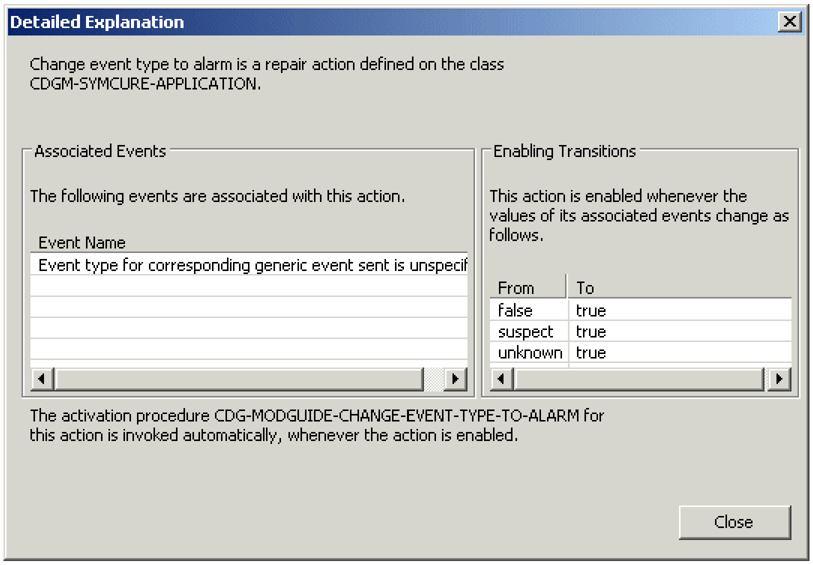
You can show detailed explanations about a generic action, which includes information about:

- Associated events.
- Enabling transitions.
- Activation procedure.
- Activation type.

To show a detailed explanation about a generic action:

➔ Choose Detailed Explanation on the generic action.

For example, here is the detailed explanation for a generic action:



Change event type to alarm is a repair action defined on the class CDGM-SYMCURE-APPLICATION.

The following events are associated with this action.

Event Name
Event type for corresponding generic event sent is unspecif

This action is enabled whenever the values of its associated events change as follows.

From	To
false	true
suspect	true
unknown	true

The activation procedure CDG-MODGUIDE-CHANGE-EVENT-TYPE-TO-ALARM for this action is invoked automatically, whenever the action is enabled.

Searching for Generic Actions

You can search for generic actions by keyword, target class, keyword and target class, or keyword or target class.

To search for generic actions:

1. Choose Tools > Search > Fault Models > Generic Actions or click the equivalent button in the Fault Modeling toolbar ( ).
2. Configure the Keyword and/or Target Class.
3. Configure Search By to determine how to search.

4 Click the Search button.

A list of generic actions that meet the search criteria appears; otherwise, No Matches Found appears.

5 Select a generic event and click the Go To button to go to the generic action.

Associating Mutually Exclusive Events

Sometimes, events that occur on the same target object are **mutually exclusive**, that is, they cannot occur simultaneously. During diagnosis, if one of the mutually exclusive events is true, then all the other events must be false. For example, a pump cannot have a temperature that is too high and too low at the same time. Note that mutually exclusive events can, however, all be false at the same time.

You can associate mutually exclusive relationships among events with the same target class and that reside in the same module. Mutually exclusive events can be defined in separate fault model folders.

To associate mutually exclusive events:

1 Show the generic fault model that includes the generic event that you want to be mutually exclusive of other events defined for the target class.

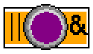
2 Choose Associate Mutex Events on the mutually exclusive event.

You can also click the Associate Mutually Exclusive Events button on the Advanced tab of the generic event properties dialog.

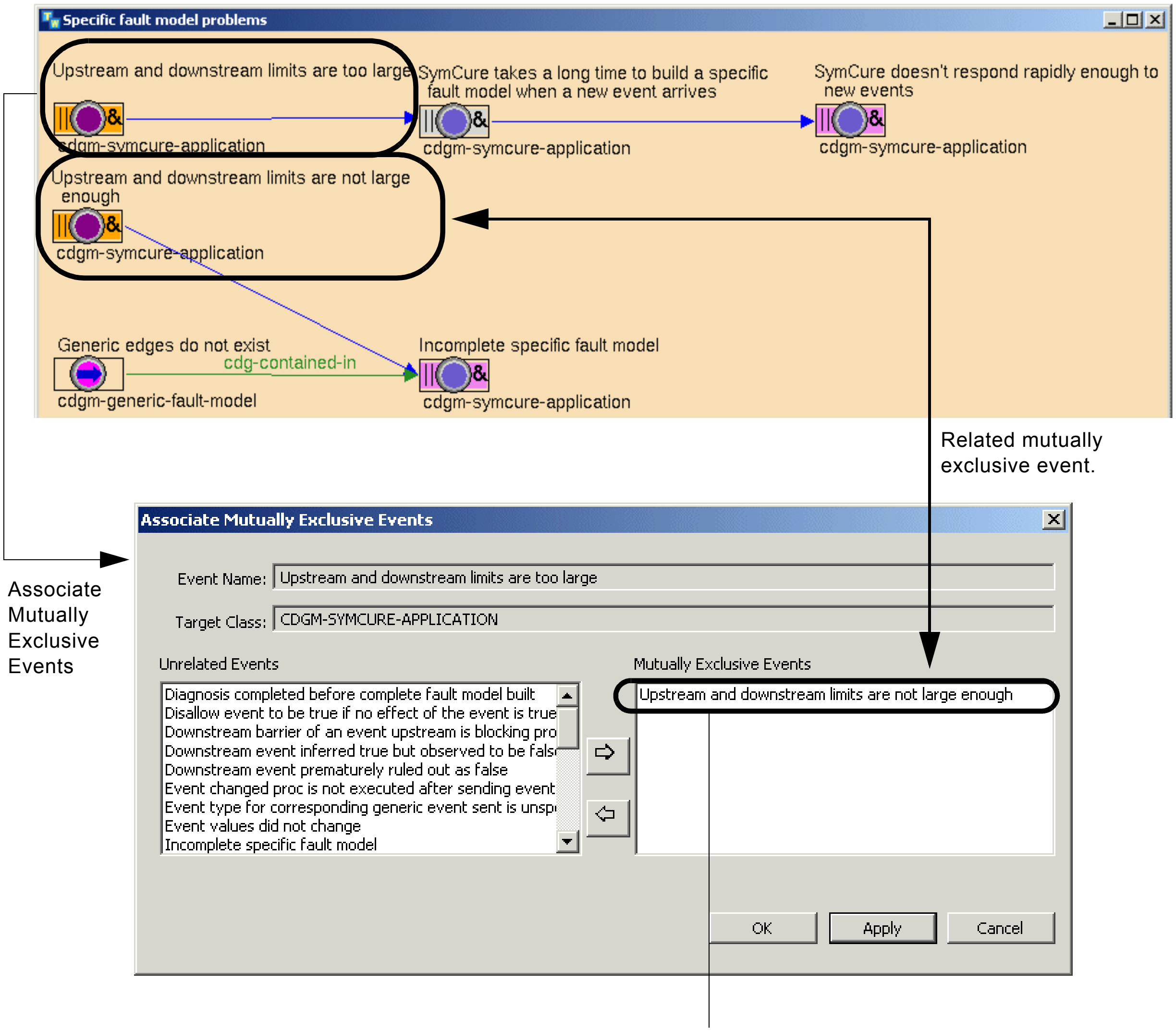
SymCure displays a dialog that includes all generic events defined for the target class, except the selected generic event.

3 Select one or more events from the Unrelated Events column that are mutually exclusive of the selected event, click the right arrow button to move them to the Mutually Exclusive column, then click OK.

The center of the generic event icon turns a darker shade of purple to indicate that it has an associated mutually exclusive event:



Here is part of the detail of the “SymCure application errors” fault model folder and the dialog for associating mutually exclusive events for the “Upstream and downstream limits are too large” event. The “Upstream and downstream limits are not large enough” event is mutually exclusive of the “Upstream and downstream limits are too large” event, which means that whenever the downstream effect of these events is true, only one of the mutually exclusive events can be true; the other event must be false.


Specific fault model problems
Upstream and downstream limits are too large
cdgm-symcure-application
SymCure takes a long time to build a specific fault model when a new event arrives
cdgm-symcure-application
SymCure doesn't respond rapidly enough to new events
cdgm-symcure-application
Upstream and downstream limits are not large enough
cdgm-symcure-application
Generic edges do not exist
cdg-contained-in
cdgm-generic-fault-model
Incomplete specific fault model
cdgm-symcure-application

Related mutually exclusive event.

Associate Mutually Exclusive Events

Associate Mutually Exclusive Events

Event Name: Upstream and downstream limits are too large

Target Class: CDGM-SYMCURE-APPLICATION

Unrelated Events

Diagnosis completed before complete fault model built
Disallow event to be true if no effect of the event is true
Downstream barrier of an event upstream is blocking pro
Downstream event inferred true but observed to be fals
Downstream event prematurely ruled out as false
Event changed proc is not executed after sending event
Event type for corresponding generic event sent is unsp
Event values did not change
Incomplete specific fault model

Mutually Exclusive Events

Upstream and downstream limits are not large enough

The "Upstream and downstream limits are not large enough" event is mutually exclusive of the "Upstream and downstream events are too large" event.

Asserting NOT Relations between Generic Events

Your fault model might require that certain events be inferred as *not* true, when some other events are known to be true. You can assert "not" relations between generic events to perform such an inference.

For example, consider two events, "Switch is ON" and "Switch is OFF". If one event is true (false) the other must necessarily be false (true).

To assert that a generic event is not true:

1. Choose Associate NOT Logic Event on a generic event in a generic fault model.

   You can also click the Associate NOT Logic Event button on the Advanced tab of generic event properties dialog.

   The dialog shows target events that can participate in the "not" logic relationship with the source event. It also shows a list of all target events in the generic fault model, from which you can select a single event to be false when the source event is true.

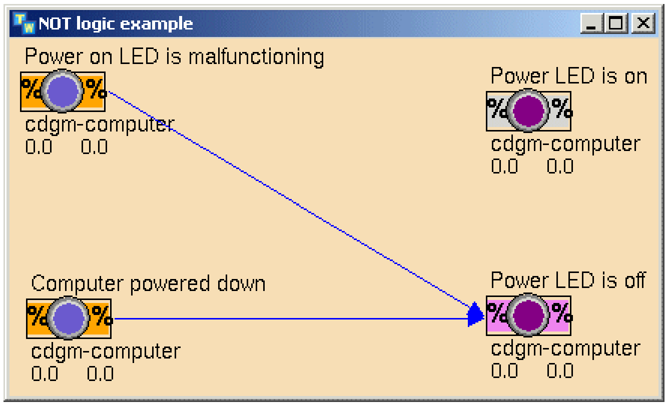
2. Select an event from the list that is not true when the source event is true, and click the Assert Event button.

   The selected event appears in the NOT Event field.

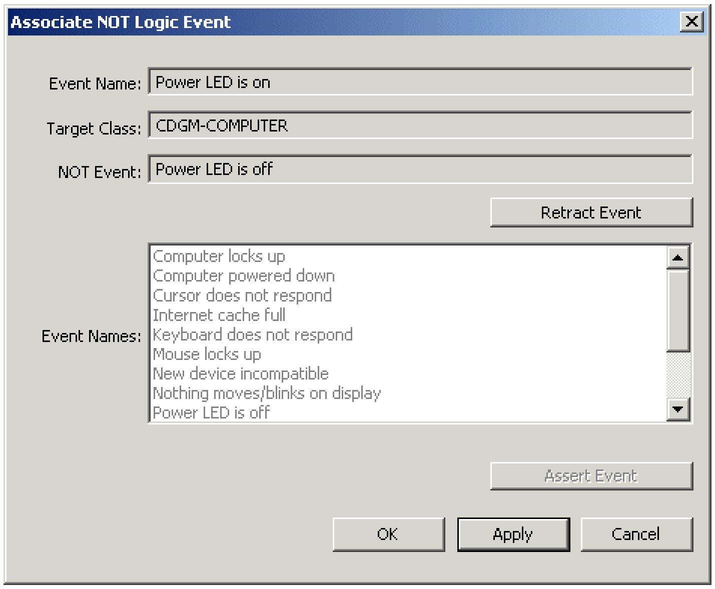
3. Click Assert Event to assert that the selected event is not true when the source event is true, then click Apply.

4. To undo the assertion, click the Retract Event button, then click Apply.

This example uses NOT logic. Using NOT logic when "Power LED is on" is true, "Power LED is off" is false. Likewise, when "Power LED is on" is false, "Power LED off" is true.



Here is the Associate NOT Logic Event dialog for the “Power LED is on” event, which asserts the “Power LED is off” event as its NOT Event:



Event Name: Power LED is on

Target Class: CDGM-COMPUTER

NOT Event: Power LED is off

Event Names:
Computer locks up
Computer powered down
Cursor does not respond
Internet cache full
Keyboard does not respond
Mouse locks up
New device incompatible
Nothing moves/blinks on display
Power LED is off

Compiling a Generic Fault Model

Before SymCure can perform its diagnosis, all generic fault models in the application must be compiled. Compiling a fault model folder identifies any errors and warnings that exist within the model. If errors exist, SymCure cannot use the generic fault model for diagnosis. SymCure ignores all warnings; they represent potential concerns about the fault model, but they are not serious enough to prevent SymCure from performing its diagnostic tasks.

SymCure reports the Compilation Status of the compilation in the properties dialog for the generic fault model folder. A status of incomplete means the fault model folder has errors. A status of complete means the fault model folder has no errors and can be used for diagnosis.

SymCure also reports the Last Compilation Time. This timestamp can help determine if it is necessary to recompile a model. It also serves as a visual indicator that compilation has actually occurred. If the model has never been compiled, no time stamp appears.

The color of the generic fault model folder indicates its compilation status. If the compilation status is complete, the folder is blue, and if the compilation status if incomplete, the folder is red.

If a fault model folder has subfolders, the value of the Compilation Status does not propagate from a subfolder to a parent folder. Thus, that status of a parent folder might be complete even though one or more subfolders are incomplete.

Compiling any fault model folder compiles all fault model folders in the application. SymCure compiles all fault model folders on start up. Initializing the application also compiles all fault model folders.

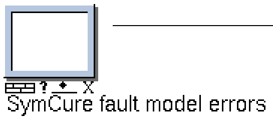
You can view errors and warnings associated with a generic fault model folder in a dialog.

Compiling a Fault Model Folder

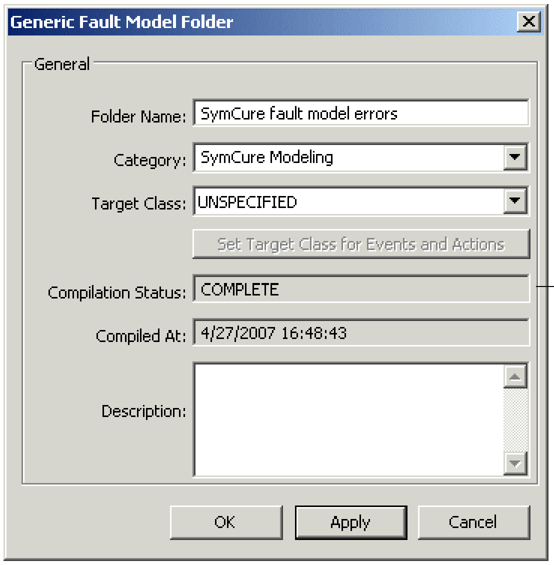
To compile a fault model folder:

➔ Choose Compile Folder on a fault model folder.

Here is a fault model folder and its associated properties dialog, which has been compiled and for which no errors exist:



The outline of the fault model folder is blue, indicating it has no errors.



Generic Fault Model Folder

General

Folder Name: SymCure fault model errors

Category: SymCure Modeling

Target Class: UNSPECIFIED

Set Target Class for Events and Actions

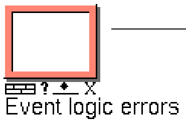
Compilation Status: COMPLETE

Compiled At: 4/27/2007 16:48:43

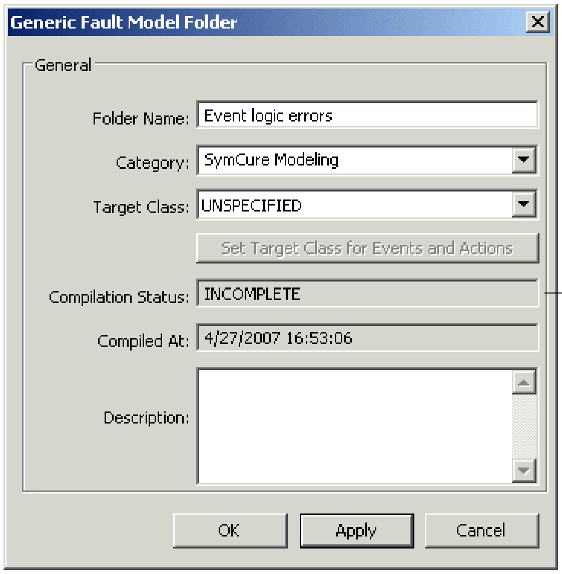
Description:

The compilation status of the fault model folder is complete.

Here is a fault model folder that has been compiled and has errors:



Fault model folders with warnings but no errors also have a tan outline.


Generic Fault Model Folder
General
Set Target Class for Events and Actions
OK
Apply
Cancel

Folder Name: Event logic errors

Category: SymCure Modeling

Target Class: UNSPECIFIED

Compilation Status: INCOMPLETE

Compiled At: 4/27/2007 16:53:06

Description:

The compilation status of the fault model folder is incomplete, indicating it has errors.

Viewing Errors

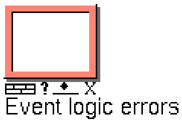
Here are the errors that can occur during compilation, organized by the object on which the error can occur:

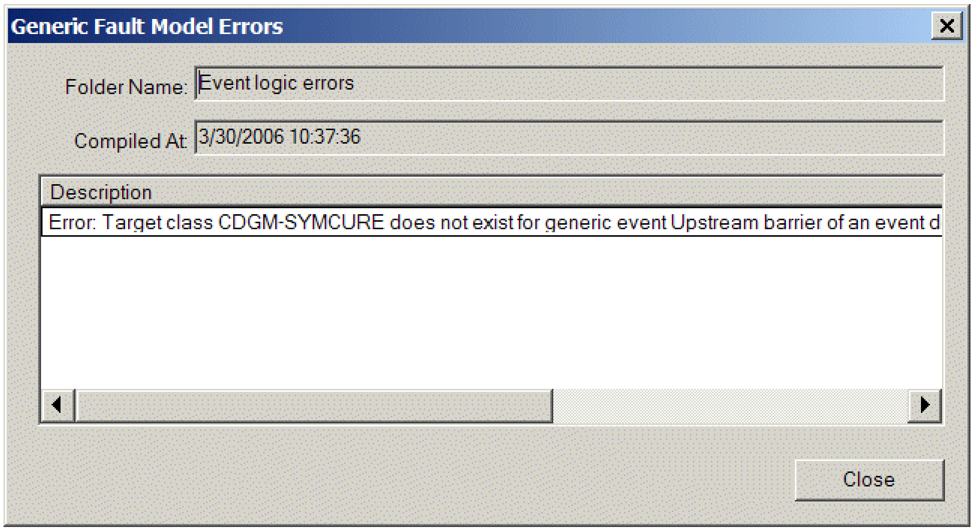
Object	Errors
Generic event	Target class does not exist. Duplicate event definition. Illegal event type. Mutex event errors: Target class of mutually exclusive events is different. Upstream barrier contains an illegal value (i.e., isn't true, false, suspect, unknown). Downstream barrier contains an illegal value (i.e., isn't true, false, suspect, unknown).
Generic IF event	State dependent procedure does not exist. Generic N/M-AND and OR-N/M event: Illegal fraction value, i.e., Fraction < 0.0 or Fraction > 1.0
Causal connection	Incorrect propagation relation (e.g., self when it should be something else). Undefined propagation relation, i.e., there is no g2-relation corresponding to the propagation-relation attribute of a causal connection connecting two generic events.
Generic event view	Target class does not exist for generic event view There is no generic event named by the event-name of generic event view Cannot connect one generic-event view to another.
Generic external actions	Target Class does not exist

To view errors for a fault model folder:

➔ Choose View Errors from the popup menu on the fault model folder detail or on the fault model folder itself.

Here are the errors for a fault model folder whose compilation status is incomplete:





Folder Name: Event logic errors

Compiled At: 3/30/2006 10:37:36

Description
Error: Target class CDGM-SYMCURE does not exist for generic event Upstream barrier of an event d

Viewing Warnings

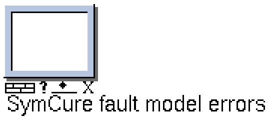
Here are the warnings that can occur during compilation, organized by the object on which the warning can occur, most of which you may ignore:

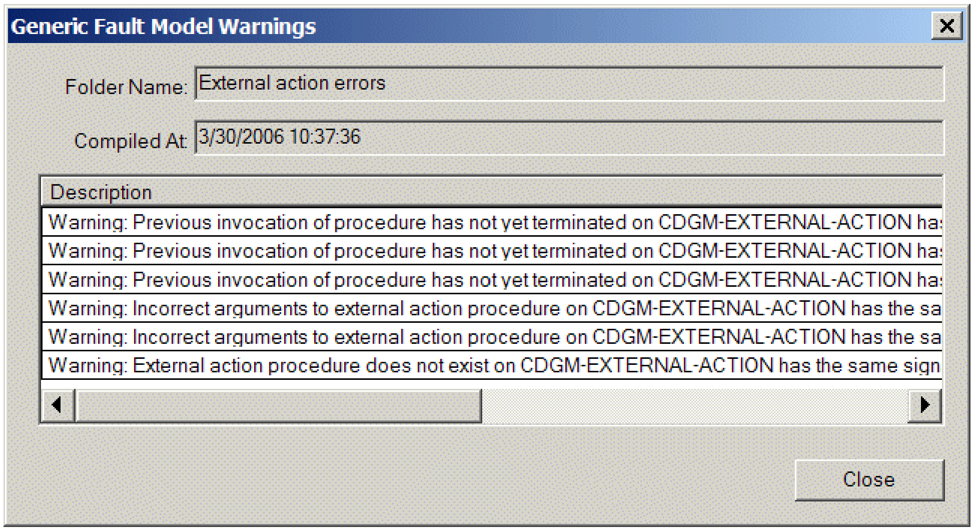
Object	Warnings
Generic event	Event changed procedure does not exist. Event unchanged procedure does not exist. Two mutually exclusive events share a common cause. This indicates that the model is not consistent. Two generic root causes have the same signature (i.e., each one will cause the exact same set of effects thus making it impossible to distinguish one from the other unless there are tests designed to do this.) The downstream barrier for any event upstream of an N/M AND event must be empty The upstream barrier for any event downstream of an OR N/M event must be empty The event type for an event may not be consistent with the topology of the model (e.g., the left-most event in a generic fault model is unspecified instead of root-cause)
Generic external actions	Action not assigned to generic event No G2 procedure for generic action

To view warnings for a fault model folder:

➔ Choose View Warnings from the popup menu on the fault model folder detail or on the fault model folder itself.

Here are the warnings for a fault model folder whose compilation status is complete. Notice that a common warning that can occur on generic events is: "Warning: Event type UNSPECIFIED for event-name on target-class may not be consistent with the topology of the generic model". Thus, any time you do not configure the event type, you will get such a warning.





Folder Name: External action errors

Compiled At: 3/30/2006 10:37:36

Description
Warning: Previous invocation of procedure has not yet terminated on CDGM-EXTERNAL-ACTION ha:
Warning: Previous invocation of procedure has not yet terminated on CDGM-EXTERNAL-ACTION ha:
Warning: Previous invocation of procedure has not yet terminated on CDGM-EXTERNAL-ACTION ha:
Warning: Incorrect arguments to external action procedure on CDGM-EXTERNAL-ACTION has the sa
Warning: Incorrect arguments to external action procedure on CDGM-EXTERNAL-ACTION has the sa
Warning: External action procedure does not exist on CDGM-EXTERNAL-ACTION has the same sign

Exporting and Importing Generic Fault Models

SymCure allows you to export generic fault models to XML files and import them back into SymCure. This feature allows you to transfer generic fault models from one application to another without requiring the source module KBs. You can export and import generic fault models interactively or programmatically.

By default, SymCure exports a generic fault model to an XML file in the `archives` subdirectory of your installation directory each time that the fault model is successfully compiled. You can disable automatic archival or change the target directory for archiving generic fault models through the `config.txt` file.

For information on exporting and importing generic fault models programmatically, see Exporting and Importing Fault Models.

For information about configuring startup parameters for exporting and importing, see Archiving.

To export a generic fault model folder:

➔ Choose Export on a generic fault model folder.

To import a generic fault model folder:

➔ Choose Project > Logic > Diagnose > Import > Import Generic Fault Model or click the equivalent button in the Fault Modeling toolbar ( ).

When the file is successfully parsed, a corresponding generic fault model folder is created.

Running SymCure Applications

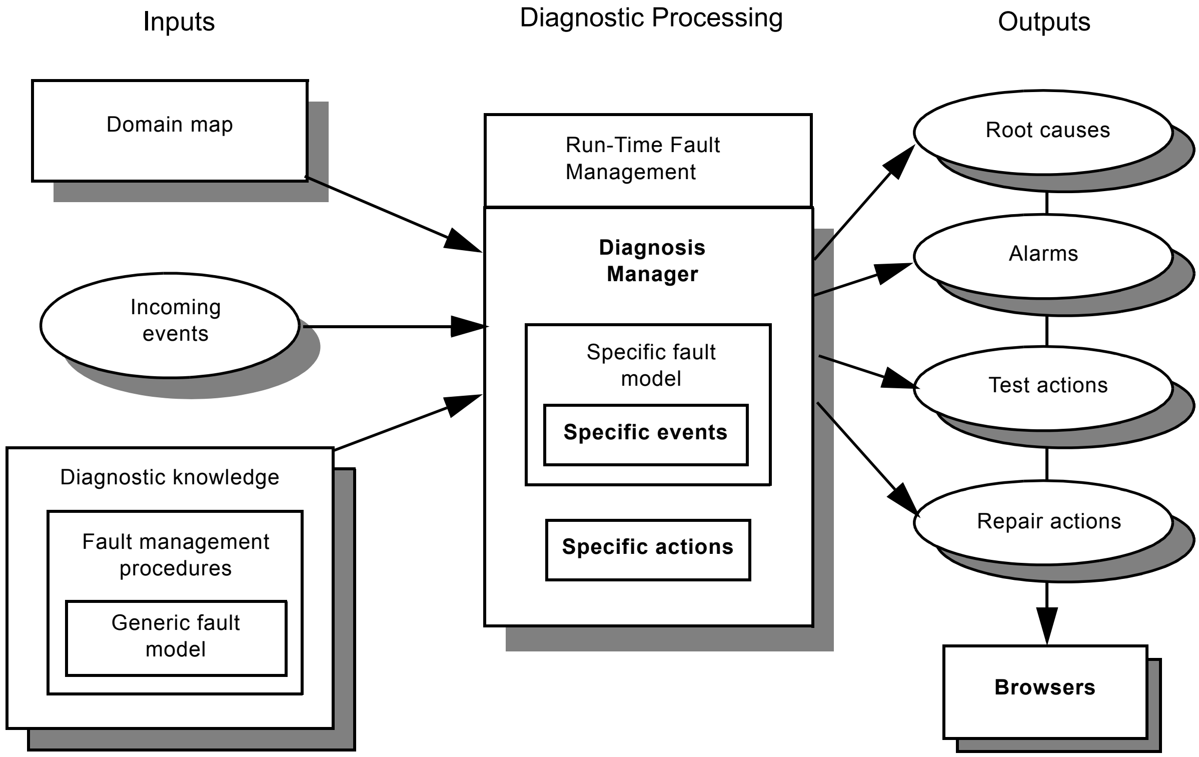
Describes how SymCure performs run-time fault management by creating specific fault models with specific events and specific actions.

Introduction

SymCure's run-time fault management is based on creating a specific fault model, which consists of specific events and specific actions that occur on specific domain objects. SymCure is responsible for run-time diagnostic processing by correlating events, propagating event value, and scheduling external actions.

You can view the specific fault model as a causal directed graph and interact with specific events and specific actions. You can also view and interact with specific events and specific actions through a set of diagnostic consoles and message browsers, which display alarms, root causes, test actions, and repair actions.

This diagram shows the architecture of a SymCure application, with these run-time elements added to the diagram labeled in bold:



SymCure’s Diagnostic Reasoning

SymCure’s diagnostic reasoning comprises specific events and actions, specific fault models, and diagnosis managers and algorithms that use these components to correlate events, identify the root causes of symptoms, predict impacts, run tests, and perform repair actions.

Specific Events and Actions

A **specific event** is a statement about a specific target object, which is uniquely identified by the combination of its name and target object.

A **specific action** is an external action that is uniquely identified by a combination of its name and its associated target object.

Specific events are created from generic events. However, unlike generic events that are defined on domain object classes, specific events apply to specific domain objects. SymCure creates specific events during diagnostic reasoning, unlike generic events, which are built by application developers.

A specific event has a value and a status. An event can take on the following values:

- “true” — The event is known to have occurred.
- “false” — The event is known to not have occurred.
- “unknown” — It is not known whether the event has occurred.
- “suspect” — It is suspected that the event may be true.

The status of an event indicates the justification for the event’s value. The status can be:

- “specified” — The value of the event is observed.
- “upstream inferred” — The value of the event is inferred from one of its effects.
- “downstream inferred” — The value of the event is inferred from one of its causes.
- “mutually exclusive” — The value of the event is inferred by mutual exclusion.

In the specific fault model, SymCure uses color and a letter abbreviation to identify the **event state**, which is a combination of the event value and the event status, as follows:

Color	Abbreviation	Value	Status
Red	T	"true"	"specified"
Salmon	T	"true"	"upstream inferred" or "downstream inferred"
Tan	T	"true"	"mutually exclusive"
Green	F	"false”	"specified"
Yellow-green	F	"false”	"upstream inferred" or "downstream inferred"
Green	F	"false”	"mutually exclusive"
Yellow	S	"suspect"	Any status
Blue	U	"unknown"	Any status

Specific Fault Models

SymCure’s fault management algorithms respond to incoming symptoms by hypothesizing and identifying root causes, predicting their impacts, running tests and repair actions, and notifying operators. At the heart of fault management are a set of **specific fault models**, which SymCure constructs from the generic fault models and specific domain objects.

SymCure diagnoses root causes from known symptoms by tracing upstream along the causal pathways from the symptoms to the faults. SymCure predicts the impact of root causes by propagating downstream from causes to effects.

SymCure combines the generic fault models with the domain representation to build focused specific fault models to investigate observed symptoms. Using the specific fault models, SymCure recognizes that a group of events are correlated to each other, identifies suspect faults that could have caused the symptoms, and selects and executes tests and repair actions to resolve the problems.

A specific fault model describes causal interactions among events within and across the specific domain objects. It also captures the current state of the diagnostic process, and can be used to generate explanations for symptoms, diagnostic conclusions, and tests and repair actions. For the sake of efficiency, in response to an incoming event, SymCure builds the minimal set of specific events, upstream of the event to diagnose possible causes and downstream of the event to predict impacts. Like generic fault models, specific fault models are represented as causal directed graphs where nodes represent specific events, and edges represent causal relations among the events.

Diagnosis Managers

A **diagnosis manager** is an object that SymCure creates to manage a specific fault model. The diagnosis manager keeps track of the root causes, alarms, tests, and repair actions associated with the specific fault model and provides access to the model.

SymCure automatically creates a diagnosis manager to handle each causally independent diagnostic problem. Diagnostic problems are causally independent of each other when they share no specific events. In other words, if you have multiple disjoint, that is, unconnected, specific fault models, SymCure will create multiple diagnosis managers, one per specific fault model.

Diagnosis managers appear in the Project menu and Navigator. You also have full access to the information they store through the SymCure API. You can display a dialog of diagnosis manager properties from a specific fault model display or from the Navigator.

For more information about the diagnosis manager, see Diagnosis Managers.

Simulating Specific Events

You can simulate specific events for domain objects in the domain map, using a menu choice on the domain object. You use this technique to test the causal logic of your generic fault models.

To simulate specific events:

1. Ensure that the Enable Fault Model toggle is enabled for the domain object whose events you want to simulate.
2. Choose Send Fault Model Event on a domain object in the domain map.

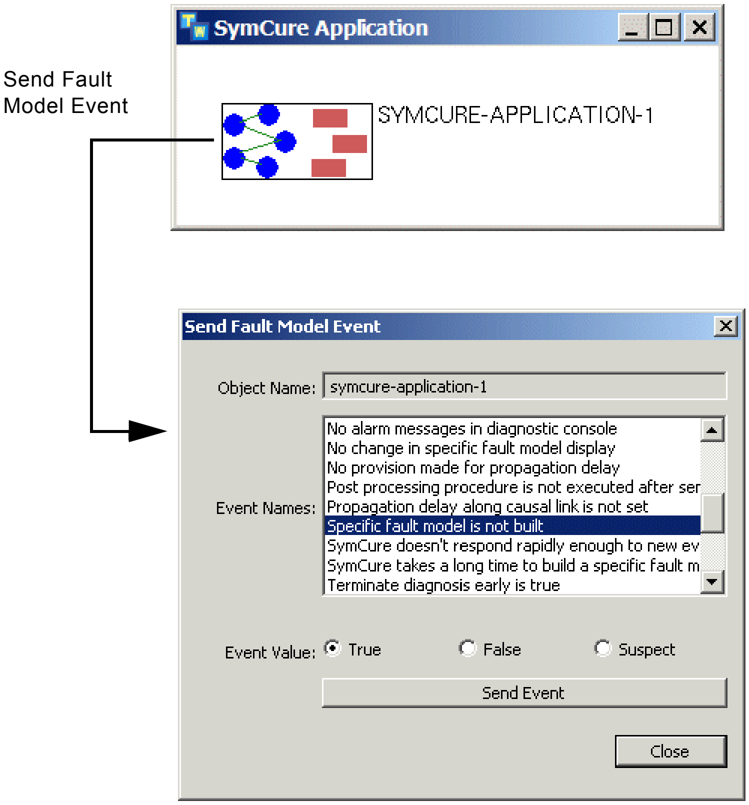
   SymCure displays a dialog that includes all the generic events defined for the domain object class.
3. Select the event you want to send, then choose an event value to be true, false, or suspect.

Note Because embedded objects do not exist in a process map, you cannot interactively send fault model events for embedded objects like you can for domain objects.

For more information about enabling the fault model for domain objects, see Chapter 14, “Running SymCure Fault Models” in the *Optegrity User’s Guide*.

Example: Simulating Events

Suppose you send a value of true for the "Specific fault model is not built" event on the domain object named symcure-application-1. Here is the send event dialog and its associated domain object:



Object Name: symcure-application-1

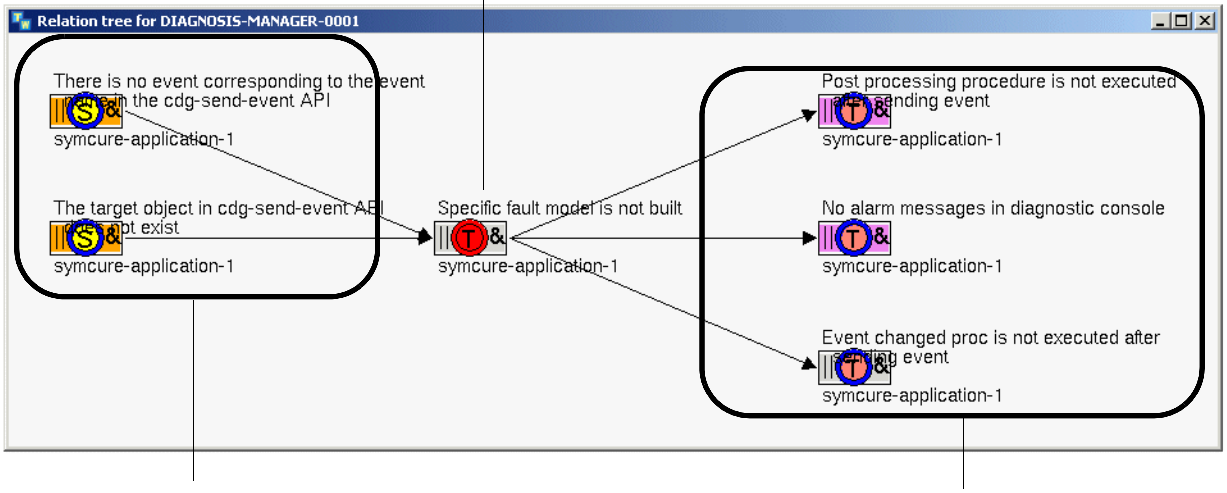
Event Names:
No alarm messages in diagnostic console
No change in specific fault model display
No provision made for propagation delay
Post processing procedure is not executed after ser
Propagation delay along causal link is not set
Specific fault model is not built
SymCure doesn't respond rapidly enough to new ev
SymCure takes a long time to build a specific fault m
Terminate diagnosis early is true

Event Value: ● True ○ False ○ Suspect

Specific Fault Model

Here is the specific fault model that SymCure creates when you send the "Specific fault model is not built" event. The event is in the middle of the diagram. The downstream events are inferred to be true, and the upstream events are suspect. The color of each specific event indicates it state—event value and event status. Notice that the specific events include text in their icons to indicate the value of the specific event, which can be T for true, F for false, S for suspect, and U for unknown. To facilitate viewing, this diagram shows the specific fault model in a different configuration.

"Specific fault model is not built" is an observed symptom; therefore, its color is red.


Relation tree for DIAGNOSIS-MANAGER-0001
There is no event corresponding to the event name in the cdg-send-event API
S
&
symcure-application-1
The target object in cdg-send-event API does not exist
S
&
symcure-application-1
Specific fault model is not built
T
&
symcure-application-1
Post processing procedure is not executed after sending event
T
&
symcure-application-1
No alarm messages in diagnostic console
T
&
symcure-application-1
Event changed proc is not executed after sending event
T
&
symcure-application-1

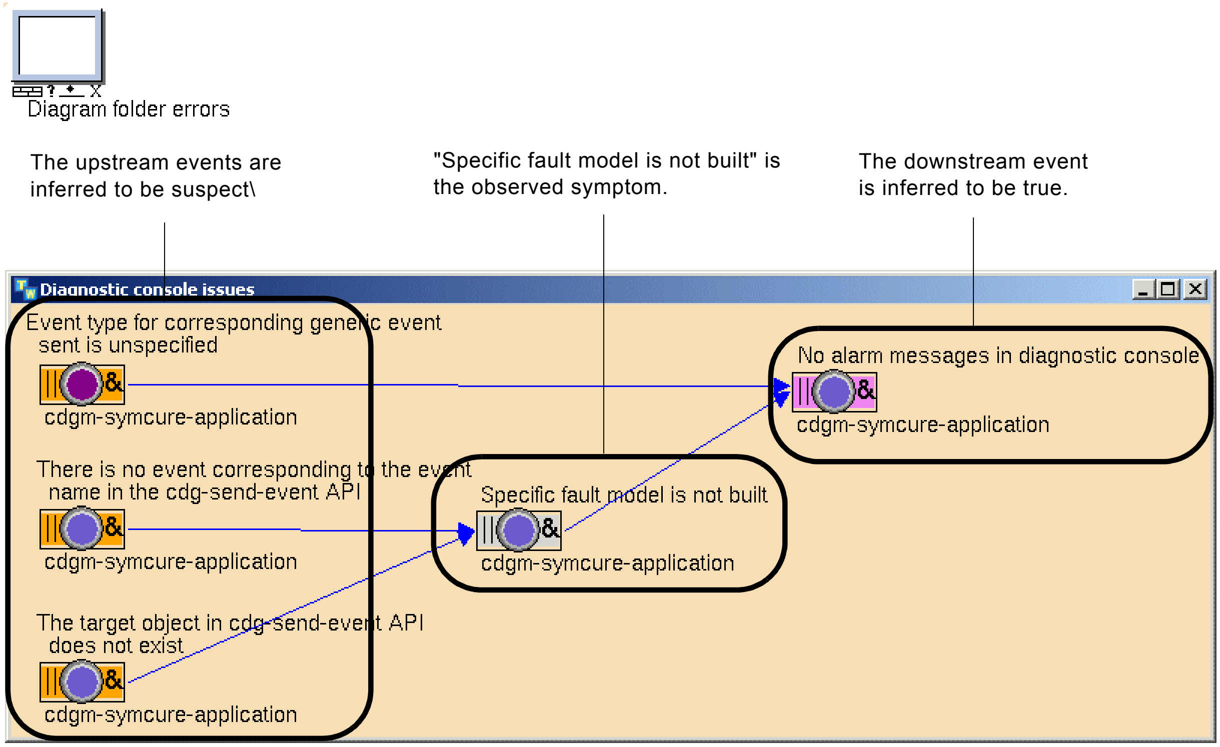
All events that are upstream of the observed symptom are suspect, based on upstream inference; therefore their color is yellow.

All events that are downstream of the observed symptom are true, based on downstream inference; therefore, their color is salmon.

For information on displaying the specific fault model, see Interacting with Specific Fault Models.

Corresponding Generic Fault Model

Compare the specific fault model with the “Diagnostic console issues” generic fault model, which defines generic events for the cdgm-symcure-application class. This diagram associates the generic events with the specific events in the previous diagram. The “Specific fault model is not built” generic event is in the middle of the diagram.

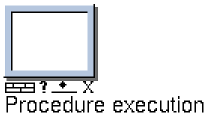

The upstream events are inferred to be suspect\

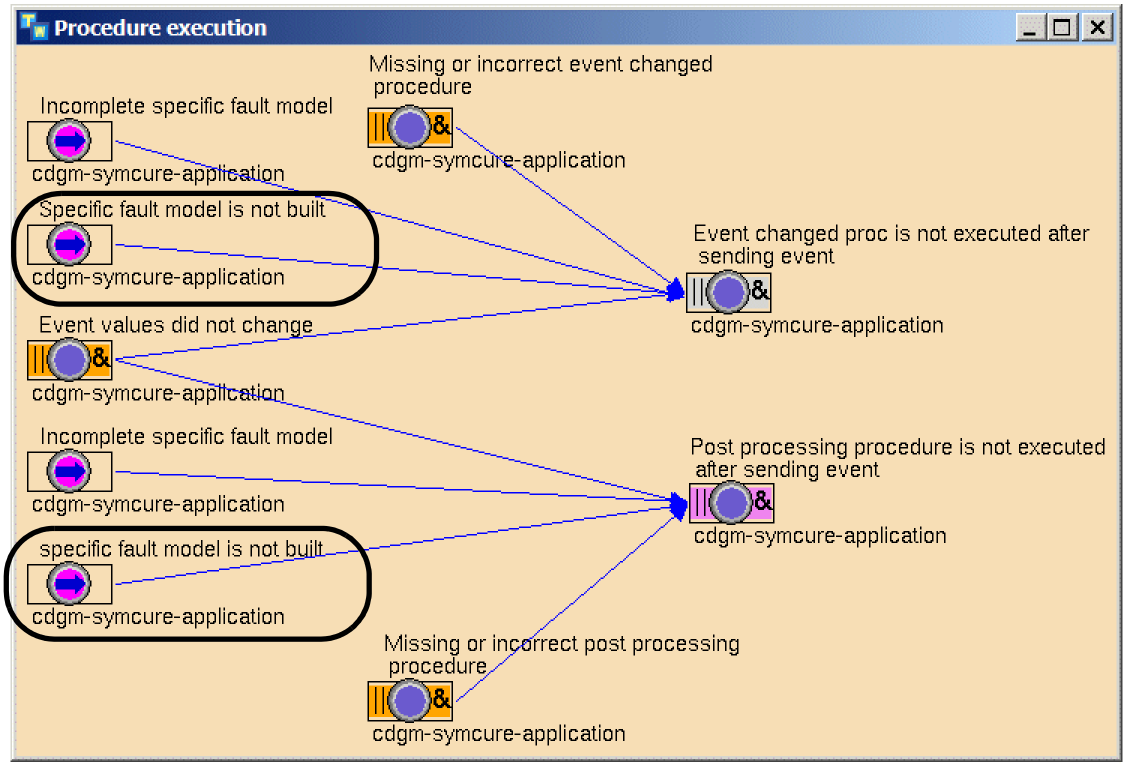
"Specific fault model is not built" is the observed symptom.

The downstream event is inferred to be true.

Diagram folder errors
Diagnostic console issues
Event type for corresponding generic event sent is unspecified
cdgm-symcure-application
There is no event corresponding to the event name in the cdg-send-event API
cdgm-symcure-application
The target object in cdg-send-event API does not exist
cdgm-symcure-application
Specific fault model is not built
cdgm-symcure-application
No alarm messages in diagnostic console
cdgm-symcure-application

Here is the "Procedure execution" generic fault model, which defines generic event views for the "Specific fault model is not built" generic event, which defines additional downstream effects of the "Specific fault model is not built" generic event.




Procedure execution
Missing or incorrect event changed procedure
cdgm-symcure-application
Incomplete specific fault model
cdgm-symcure-application
Specific fault model is not built
cdgm-symcure-application
Event changed proc is not executed after sending event
cdgm-symcure-application
Event values did not change
cdgm-symcure-application
Incomplete specific fault model
cdgm-symcure-application
Post processing procedure is not executed after sending event
cdgm-symcure-application
specific fault model is not built
cdgm-symcure-application
Missing or incorrect post processing procedure
cdgm-symcure-application

Event Propagation Algorithm

SymCure uses heuristic best first search to propagate event values across specific events. At a very high level, starting from an incoming event, the logic for propagating event values in a specific fault model from an incoming event is as follows:

```
for any event e when its value changes do
    propagate the value of the event upstream to all causes;
   propagate the value of the event downstream to effects;
end for
```

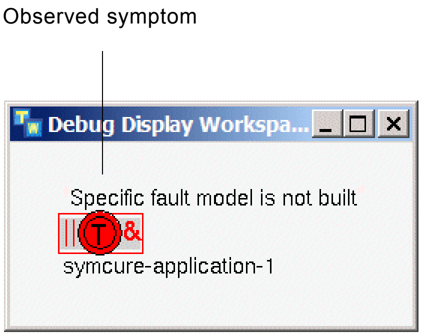
where:

> *causes* are all events that are upstream of event e.
>
> *effects* include all events that are downstream of *causes* plus event e itself.

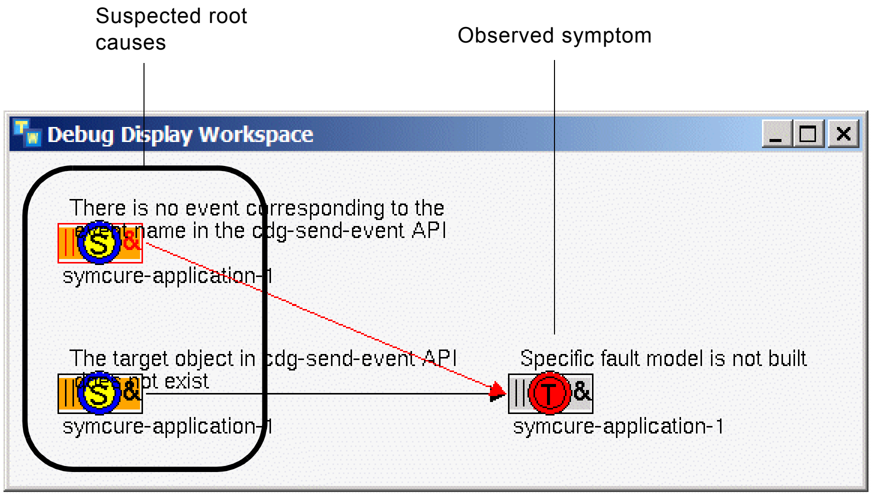
The time complexity of the propagation algorithm is linear in the number of events and the number of edges in the specific fault model. The maximum number of events in a specific fault model is bound by the product of the number of managed domain objects and the size of the largest generic fault model. In practice, because SymCure constructs only the events that are correlated to incoming symptoms, the actual size of a specific fault model is usually a small subset of the maximum possible size.

The following diagrams show the sequence of event propagation, using the SymCure application diagnostics example. It uses the SymCure debugger to show the event propagation algorithm.

1 “Specific fault model is not built” on symcure-application-1 is observed to be true:



2 The diagnosis manager builds the specific events upstream of the observed symptom:

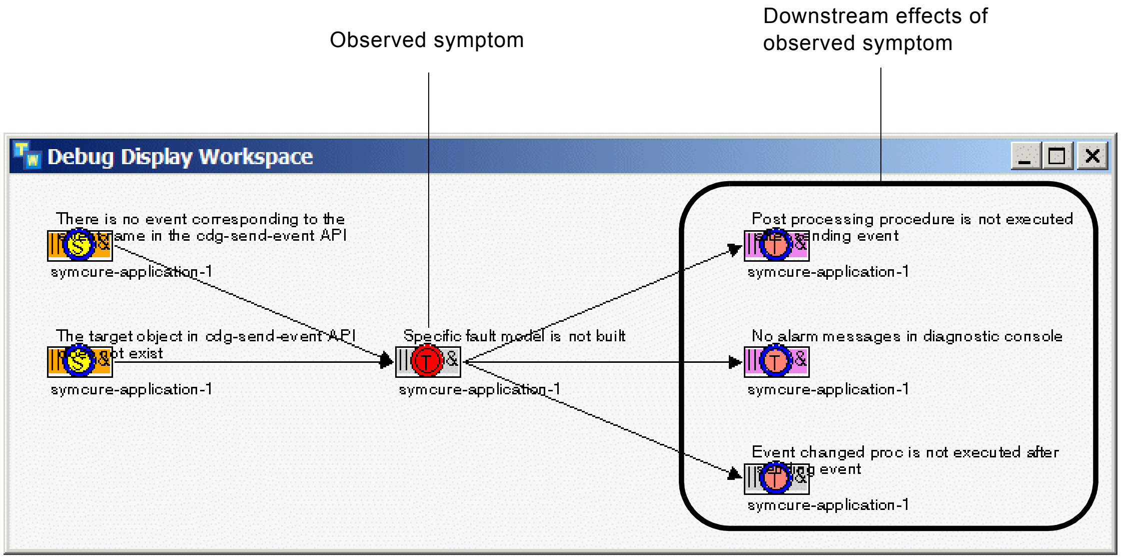


Suspected root causes

Observed symptom

Debug Display Workspace
There is no event corresponding to the event name in the cdg-send-event API
S &
symcure-application-1
The target object in cdg-send-event API does not exist
S &
symcure-application-1
Specific fault model is not built
T &
symcure-application-1

3 The diagnosis manager builds the specific downstream effects of the observed symptom:

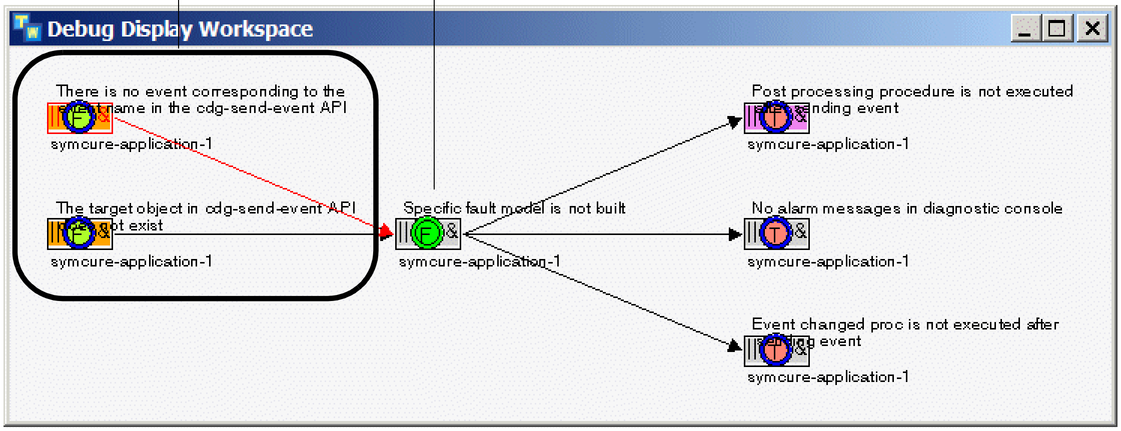


Observed symptom

Downstream effects of observed symptom

Debug Display Workspace
There is no event corresponding to the event name in the cdg-send-event API
S &
symcure-application-1
The target object in cdg-send-event API does not exist
S &
symcure-application-1
Specific fault model is not built
T &
symcure-application-1
Post processing procedure is not executed after pending event
T &
symcure-application-1
No alarm messages in diagnostic console
T &
symcure-application-1
Event changed proc is not executed after pending event
T &
symcure-application-1

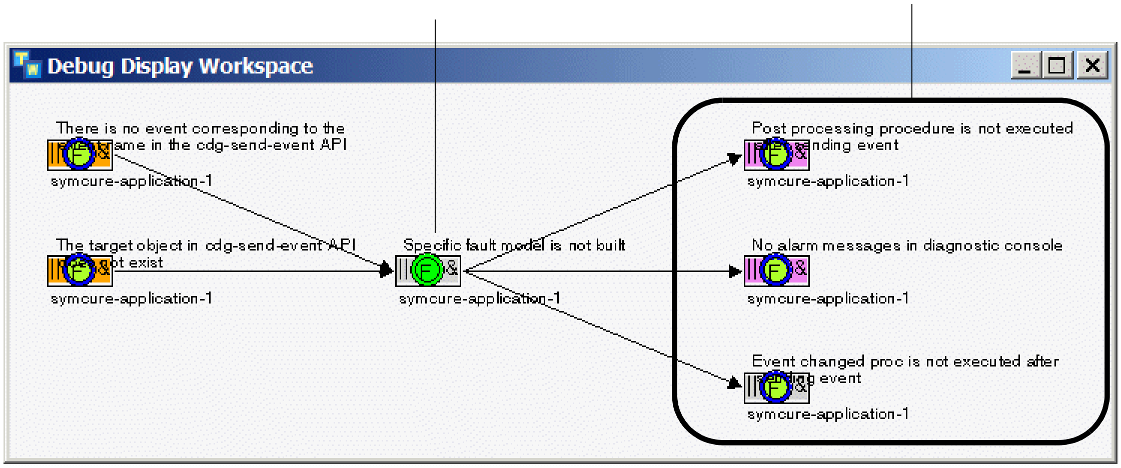
4 "Specific fault model is not built" on symcure-application-1 is observed to be false. The diagnosis manager propagates event value upstream. Both upstream root causes are inferred to be false.


Root causes are inferred to be false.

Observed symptom

Debug Display Workspace
There is no event corresponding to the event name in the cdg-send-event API
F
symcure-application-1
The target object in cdg-send-event API does not exist
F
symcure-application-1
Specific fault model is not built
F
symcure-application-1
Post processing procedure is not executed after sending event
T
symcure-application-1
No alarm messages in diagnostic console
T
symcure-application-1
Event changed proc is not executed after sending event
T
symcure-application-1

5 The diagnosis manager propagates event value downstream. All downstream effects are inferred to be false.


Observed symptom

Downstream effects are inferred to be false.

Debug Display Workspace
There is no event corresponding to the event name in the cdg-send-event API
F
symcure-application-1
The target object in cdg-send-event API does not exist
F
symcure-application-1
Specific fault model is not built
F
symcure-application-1
Post processing procedure is not executed after sending event
F
symcure-application-1
No alarm messages in diagnostic console
F
symcure-application-1
Event changed proc is not executed after sending event
F
symcure-application-1

Interacting with Specific Events and Actions through Diagnostic Console Browsers

You can view and interact with specific events and actions in four built-in diagnostic console browsers:

- Alarms Browser — Shows specific events of type alarm.
- Root Causes Browser — Shows specific events of type root-cause.
- Test Actions Browser — Shows specific test actions.
- Repair Actions Browser — Shows specific repair actions.

The built-in SymCure browsers provide a tabular view of specific events and actions. The browsers show various information about a specific event or specific action, such as the event or action name, the target object, a text message, the event value, the action status and type, and the time at which the event or action was last updated.

The Alarms and Root Causes browsers use the same color scheme as the specific fault model to indicate event value and status. For details, see Specific Events and Actions.

You can configure the default browser that SymCure uses for alarms, root causes, test actions, and repair actions. For example, you might want to display both alarms and root causes in the same browser, and both test actions and repair actions in the same browser. To do this, you configure SymCure initialization parameters. For details, see Default Browsers in Configuring SymCure Applications.

The following sections use the SymCure application diagnostics example to show how to interact with alarms, root causes, test actions, and repair actions. The events that appear in the browsers are the result of sending the “Compilation status incomplete” event on the domain object named gfm-1. For information on how to simulate this event, see Simulating Specific Events.

Note To generate root cause events for this example, you must set the cdg-allow-unspecified-event-to-be-root-cause startup parameter to true. See Specific Fault Model Creation.

Note Unlike the default Message Browser, the SymCure browsers do not allow you to acknowledge or delete messages. Instead, SymCure handles event creation and deletion as part of the diagnostic process.

For further information on running SymCure within Optegrity, see the *Optegrity User’s Guide*.

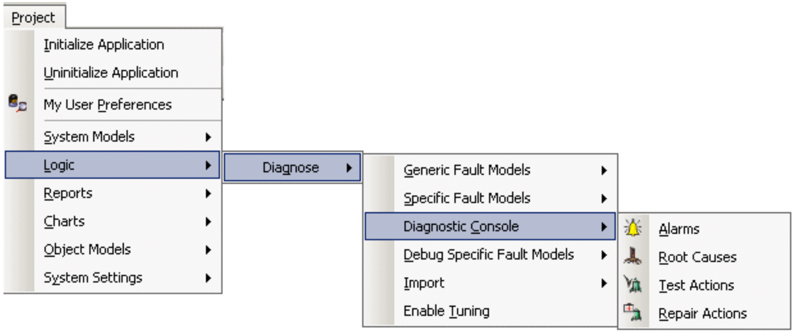
Displaying the Browsers

You access these browsers through the Project menu or the Fault Modeling toolbar.

To display the browsers:

➔ Choose Project > Logic > Diagnose > Diagnostic Console and choose the browser you want to display or click the equivalent button in the Fault Modeling toolbar ( ).

The Diagnostic Console menu includes the four SymCure browsers: Alarms, Root Causes, Test Actions, and Repair Actions.

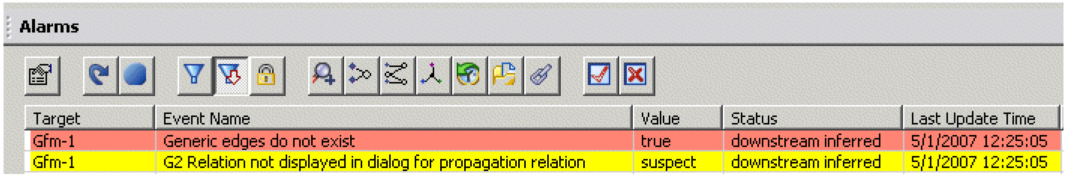


Alarms Browser

The Alarms Browser has these columns:

- Target — The domain object that is the target of the event.
- Event Name — The name of the specific event.
- Value — The value of the event, which is "true", "false", "suspect", or "unknown".
- Status — The status of the event, which is "specified", "downstream inferred", "upstream inferred" or "mutually exclusive".
- Last Update Time — The time at which the event value or status was last updated.

Here is the Alarms Browser that results when sending the “Compilation status is incomplete” event on gfm-1:

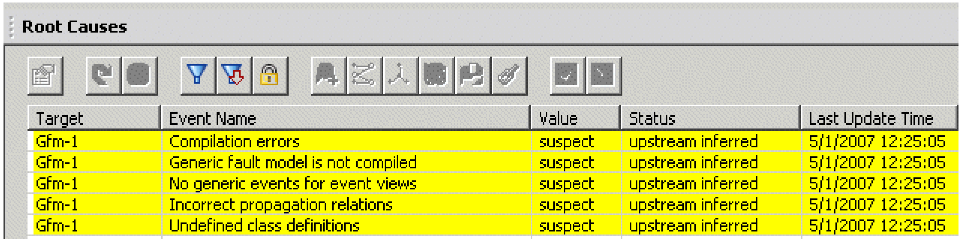


Target	Event Name	Value	Status	Last Update Time
Gfm-1	Generic edges do not exist	true	downstream inferred	5/1/2007 12:25:05
Gfm-1	G2 Relation not displayed in dialog for propagation relation	suspect	downstream inferred	5/1/2007 12:25:05

Root Causes Browser

The Root Causes Browser has the same columns as the Alarms Browser.

Here is the Root Causes Browser that results when sending the "Compilation status is incomplete" event on gfm-1:



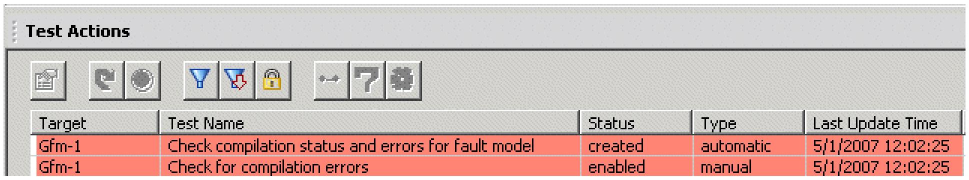
Target	Event Name	Value	Status	Last Update Time
Gfm-1	Compilation errors	suspect	upstream inferred	5/1/2007 12:25:05
Gfm-1	Generic fault model is not compiled	suspect	upstream inferred	5/1/2007 12:25:05
Gfm-1	No generic events for event views	suspect	upstream inferred	5/1/2007 12:25:05
Gfm-1	Incorrect propagation relations	suspect	upstream inferred	5/1/2007 12:25:05
Gfm-1	Undefined class definitions	suspect	upstream inferred	5/1/2007 12:25:05

Test Actions Browser

The Test Actions Browser has these columns:

- Target — The domain object that is the target of the test action.
- Test Name — The name of the specific test.
- Status — The status of the test. The values are: "create", "enabled", "running", and "inactive".
- Type — The type of test. The values are: "manual" and "automatic".
- Last Update Time — The time at which the event value was last updated.

Here is the Test Actions Browser that results when sending the "Compilation status is incomplete" event on gfm-1:

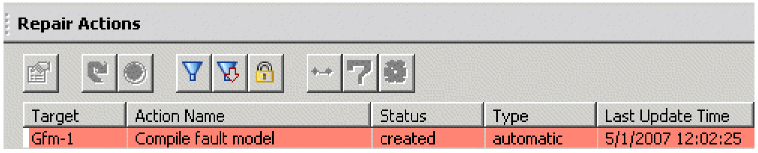


Target	Test Name	Status	Type	Last Update Time
Gfm-1	Check compilation status and errors for fault model	created	automatic	5/1/2007 12:02:25
Gfm-1	Check for compilation errors	enabled	manual	5/1/2007 12:02:25

Repair Actions Browser

The Repair Actions Browser has the same columns as the Test Actions Browser.

Here is the Repair Actions Browser that results when sending the "Compilation status is incomplete" event on gfm-1:



Target	Action Name	Status	Type	Last Update Time
Gfm-1	Compile fault model	created	automatic	5/1/2007 12:02:25

Toolbar Buttons

You interact with events and actions in the browsers by selecting the event or action to enable the toolbar buttons, then clicking a button.

This table describes toolbar buttons in each of the browsers. The toolbar buttons are available in all browsers unless otherwise noted.

Button	Name	Description
	Properties	Shows the properties of the event.
	Target	Shows the domain object on which the event has occurred.
●	Show Generic Event	(Alarms and Root Causes Browser) Shows the generic event in the generic fault model folder.
○	Show Generic Action	(Test Actions and Repair Actions Browser) Shows the generic action in the generic fault model folder.
	Configure Filter	Configures filter criteria for filtering events.
	Filters	Applies the selected event filter.
	Lock View	Locks the browser so no more events can appear in the browser.
	Detailed Explanation	(Alarms and Root Causes Browsers) Shows the detailed explanation of the selected alarm or root cause event.
	Root Causes	(Alarms Browser) Shows the root causes for the selected alarm event.
	Causal Model	(Alarms and Root Causes Browsers) Shows the causal model for an alarm or root cause event.

○	Show Generic Action	(Test Actions and Repair Actions browser) Shows the generic action in the generic fault model folder.
▽	Configure Filter	Configures filter criteria for filtering events.
▽	Filters	Applies the selected event filter.
🔒	Lock View	Locks the browser so no more events can appear in the browser.
🔍	Detailed Explanation	(Alarms and Root Causes Browsers) Shows the detailed explanation of the selected alarm or root cause event.
⇄	Root Causes	(Alarms Browser) Shows the root causes for the selected alarm event.
⇄	Causal Model	(Alarms and Root Causes Browsers) Shows the causal model for an alarm or root cause event.

Button	Name	Description
☒	False	(Alarms and Root Causes Browsers) Sets the value of the event to false.
←	Events	(Test and Repair Actions Browsers) Shows the underlying event associated with the external test or repair action.
?	Explanation	(Test and Repair Actions Browsers) Describes the events that led up to creating and activating the action.
⚙	Run	(Test and Repair Actions Browsers) Runs the selected manual test or repair action.

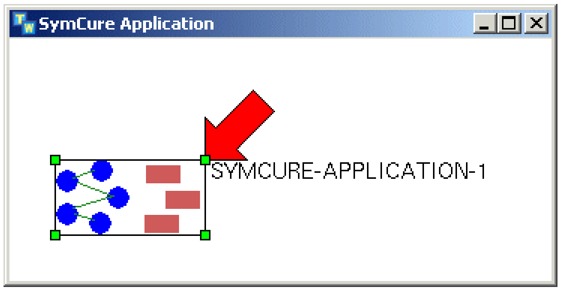
Showing the Event Target

You can go to the event target of an alarm or root cause. The event target is the domain object for which the specific event is generated.

To show the event target:

➔ Select an event and click the Target toolbar button: ↻

The event target for the "Incomplete specific fault model" alarm on symcure-application-1 is the symcure-application-1 domain object:

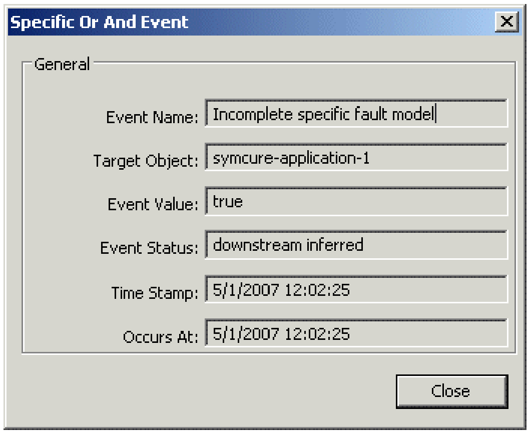


Showing Event Properties

To show event properties:

➔ Select an event and click the Properties toolbar button:

Here are the properties for the "Incomplete specific fault model" alarm on symcure-application-1:



Specific Or And Event

General

Event Name: Incomplete specific fault model

Target Object: symcure-application-1

Event Value: true

Event Status: downstream inferred

Time Stamp: 5/1/2007 12:02:25

Occurs At: 5/1/2007 12:02:25

Sorting Events and Actions

By default, events and actions are sorted in the order in which they are received, with the most recent messages at the top. You can sort events and actions, based on any column.

To sort events based on a column:

➔ Click the column header to sort messages by the information in the selected column.

The selected column includes an arrow in the column header to indicate that messages are sorted, based on that column. Click the arrow to sort the events in reverse order.

Locking the Browser

By default, events and actions appear in the browsers as soon as they are created. You can lock a browser to prevent further alarms from arriving. When you unlock the browser, all the events and actions that would have appeared, appear all at once.

To lock/unlock the browser:

➔ Click the Lock View toggle button:

When the browser is locked, the lock icon appears pressed.

To unlock the browser:

➔ Click the Lock View toggle button when it is pressed:

Filtering Events and Actions

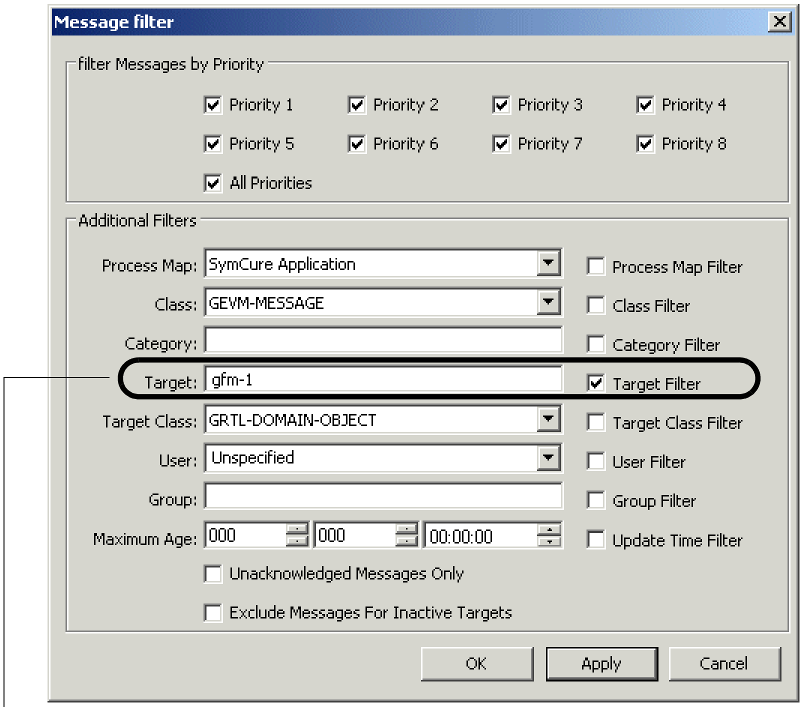
You can filter events and actions, based on the target object. To filter events and actions, you configure the filter criteria, then you apply it to the browser. The filter criteria is inclusive, that is, it shows the events and actions that meet the specified criteria.

Note The only relevant filter criteria for use with SymCure events is the target object; the other filter criteria are relevant for general operator messages only.

To configure the filter:

➔ Click the Configure Filters toolbar button:

This filter dialog shows only events whose target is gfm-1:



Message filter

filter Messages by Priority

☑ Priority 1 ☑ Priority 2 ☑ Priority 3 ☑ Priority 4

☑ Priority 5 ☑ Priority 6 ☑ Priority 7 ☑ Priority 8

☑ All Priorities

Additional Filters

Process Map: SymCure Application □ Process Map Filter

Class: GEVM-MESSAGE □ Class Filter

Category: □ Category Filter

Target: gfm-1 ☑ Target Filter

Target Class: GRTL-DOMAIN-OBJECT □ Target Class Filter

User: Unspecified □ User Filter

Group: □ Group Filter

Maximum Age: 000 000 00:00:00 □ Update Time Filter

□ Unacknowledged Messages Only

□ Exclude Messages For Inactive Targets

Only events whose target is gfm-1 appear in the browser.

To apply the filter:

➔ Click the Filters toggle button: ▽

The Filters button appears pressed when the filter is in effect.

To remove the filter:

➔ Click the Filters toggle button when it is pressed: ▽

Interacting with Alarms and Root Causes

You can show various information about the specific fault model associated with alarms and root causes.

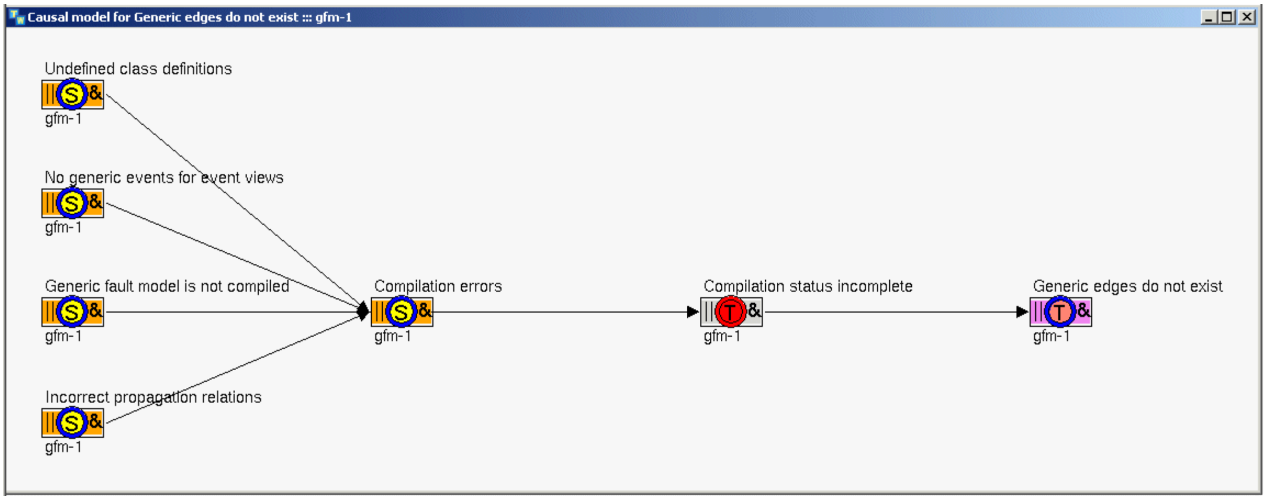
Showing Root Causes for Alarms

All alarms have one or more suspected root causes, based on upstream event propagation.

To show root causes for an alarm:

➔ Select an alarm and click the Root Causes toolbar button: ⊱

Here is the specific fault model that shows the root causes for the "Generic edges do not exist" predicted alarm on gfm-1:


Causal model for Generic edges do not exist ::: gfm-1
Undefined class definitions
gfm-1
No generic events for event views
gfm-1
Generic fault model is not compiled
gfm-1
Incorrect propagation relations
gfm-1
Compilation errors
gfm-1
Compilation status incomplete
gfm-1
Generic edges do not exist
gfm-1
S
T
&

Showing Causal Models

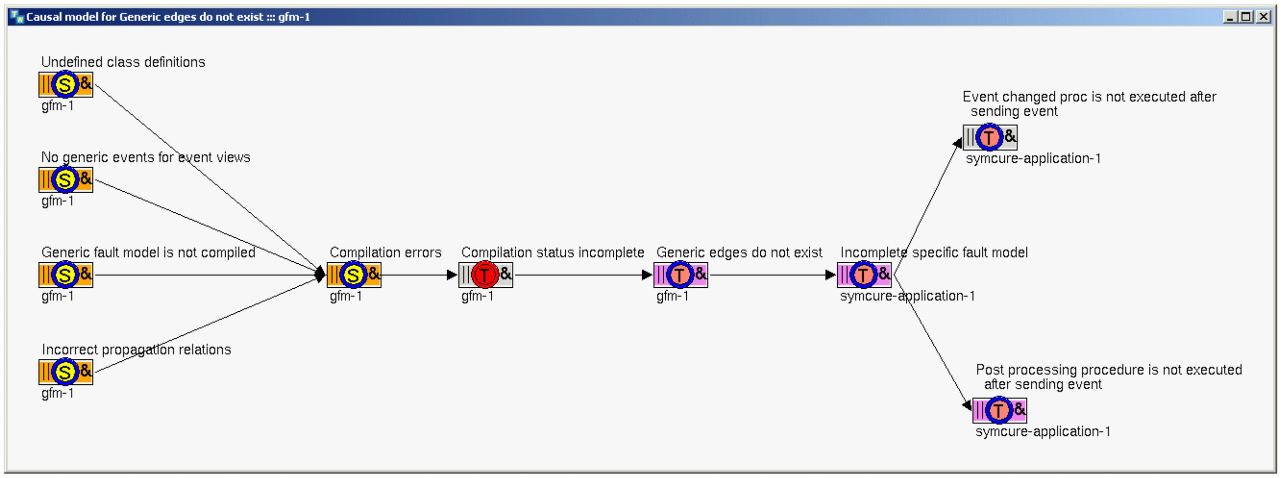
You can show the causal model for an alarm or root cause, which is the specific fault model focused around a specific event. The causal model includes observed symptoms, predicted events, and suspected root causes, based on upstream and downstream propagation. The causal model includes alarm events, root cause events, and events that are neither alarms nor root causes.

When the number of specific events is less than or equal to 100, the causal model show all specific events in the model. When the number of events is greater than 100, the causal model shows an encapsulated view, which includes only immediately upstream and downstream events, and root causes.

To show the causal model:

➔ Select an event and click the Causal Model toolbar button:

Here is the causal model for the "Generic edges do not exist" predicted alarm on gfm-1:


Causal model for Generic edges do not exist ::: gfm-1
Undefined class definitions
gfm-1
No generic events for event views
gfm-1
Generic fault model is not compiled
gfm-1
Incorrect propagation relations
gfm-1
Compilation errors
gfm-1
Compilation status incomplete
gfm-1
Generic edges do not exist
gfm-1
Incomplete specific fault model
symcure-application-1
Event changed proc is not executed after sending event
symcure-application-1
Post processing procedure is not executed after sending event
symcure-application-1

Showing the Event Summary

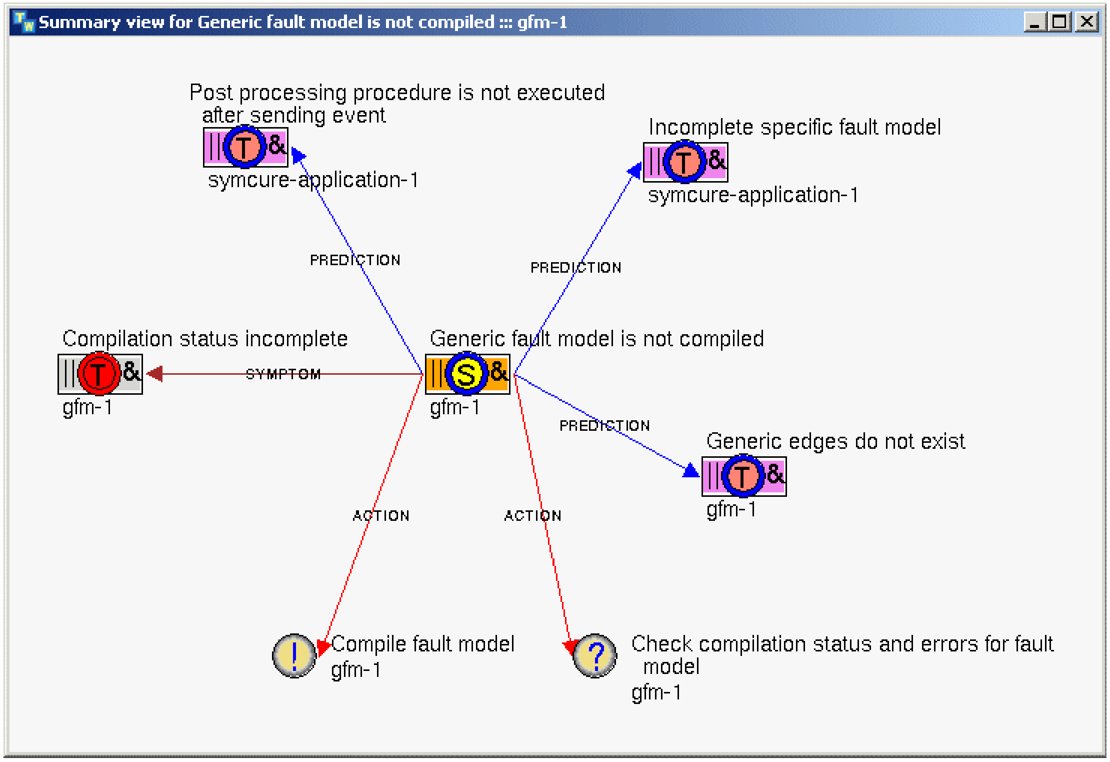
The summary view for a specific event summarizes the key relationships between the event, its causes, effects and actions, without displaying the complete details of the specific fault model.

You can show a summary of the predicted alarms, observed symptoms, root causes, and actions for an alarm. For a root cause, you can show the predictions, symptoms, and actions. The relationships between the alarms and actions are labeled in the summary view.

To show the event summary:

➔ Select an event and click the Summary toolbar button:

Here is the event summary for the root cause event “Generic fault model is not compiled” on gfm-1, which shows its predicted alarms, symptoms, and associated actions:


Summary view for Generic fault model is not compiled ::: gfm-1
Post processing procedure is not executed after sending event
T
symcure-application-1
Incomplete specific fault model
T
symcure-application-1
PREDICTION
PREDICTION
Compilation status incomplete
T
gfm-1
SYMPTOM
Generic fault model is not compiled
S
gfm-1
PREDICTION
Generic edges do not exist
T
gfm-1
ACTION
ACTION
!
Compile fault model
gfm-1
?
Check compilation status and errors for fault model
gfm-1

Showing the Event Chronology

The event chronology contains a history of all alarm and root cause events, and actions related to the diagnostic problem. The event chronology supplements the specific fault model by explicitly outlining the chronological sequence of events that lead up to the current state of the specific fault model.

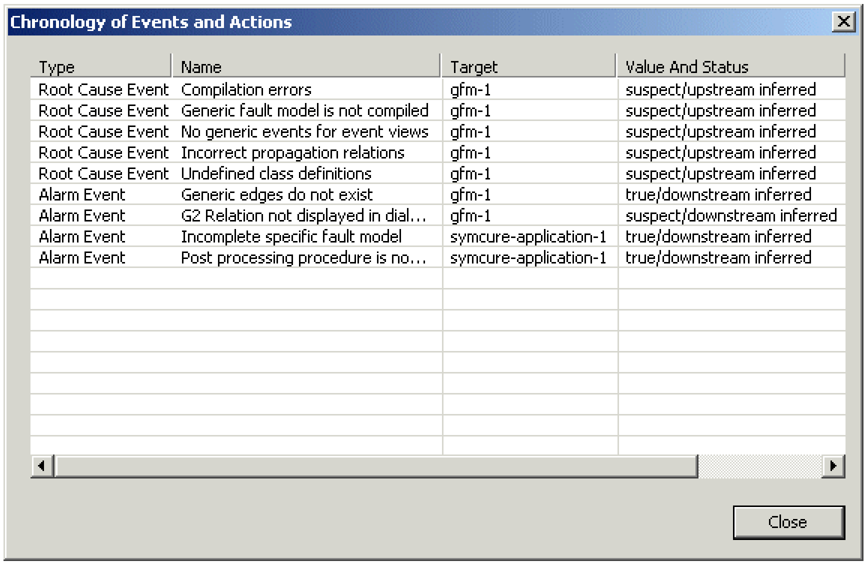
Each alarm and root cause has an associated chronology of events that led up to the alarm or root cause being created. Only alarm and root cause events, and actions appear in the event chronology.

SymCure represents timestamps of specific events in subsecond real time. As a result, the event chronology table displays specific events in chronological order according to subsecond time. Note that the timestamp is displayed as a combination of date and time in the event chronology table, all message browsers, and event display objects in all specific fault model displays.

To show the event chronology:

➔ Select an event and click the Chronology toolbar button:

Here is the event chronology for “Generic fault model is not compiled” root cause on gfm-1, with the selected event at the top of the list and the events leading up to that event following.



Type	Name	Target	Value And Status
Root Cause Event	Compilation errors	gfm-1	suspect/upstream inferred
Root Cause Event	Generic fault model is not compiled	gfm-1	suspect/upstream inferred
Root Cause Event	No generic events for event views	gfm-1	suspect/upstream inferred
Root Cause Event	Incorrect propagation relations	gfm-1	suspect/upstream inferred
Root Cause Event	Undefined class definitions	gfm-1	suspect/upstream inferred
Alarm Event	Generic edges do not exist	gfm-1	true/downstream inferred
Alarm Event	G2 Relation not displayed in dial...	gfm-1	suspect/downstream inferred
Alarm Event	Incomplete specific fault model	symcure-application-1	true/downstream inferred
Alarm Event	Post processing procedure is no...	symcure-application-1	true/downstream inferred

Saving the Event Chronology

You can save the event chronology to a .csv file for analysis outside of SymCure.

To save the event chronology:

➔ Select an event and click the Save toolbar button:

Setting the Event Value

You can interactively set the value of a specific event to true or false. A model developer might do this to test the behavior of the generic fault model. An operator might do this at run time when the event occurs to inject a new value.

When you set the event value to true or false, the event status is "specified".

To set the event value to true:

➔ Select an event and click the True toolbar button:

To set the event value to false:

➔ Select an event and click the False toolbar button:

Interacting with External Actions

An external action is a procedure that SymCure activates when the value of the underlying event changes, according to the specified enabling transition for the particular type of action. The external action can execute automatically or manually, depending on how it is configured.

You can interact with two types of external actions through the built-in browsers: test actions and repair actions. Test actions test for the occurrence of a specific event and returns the value of the event via an API call to SymCure. Repair actions perform some type of repair to the domain model.

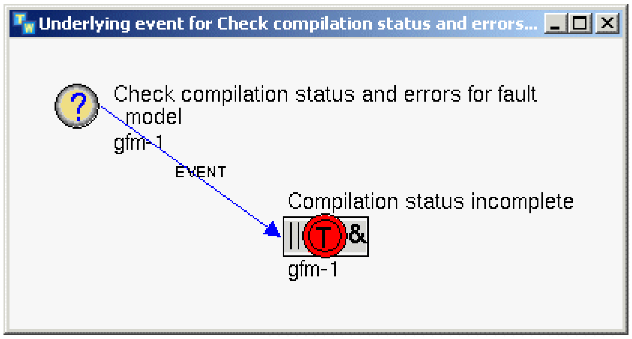
For more information, see Creating Generic External Actions.

Showing Underlying Events

To show the underlying event:

➔ Select a test or repair action and click the Events toolbar button: ⟵

Here is the underlying event for the "Check compilation status and errors for fault model" specific test action for gfm-1:


Underlying event for Check compilation status and errors...
?
Check compilation status and errors for fault model
gfm-1
EVENT
Compilation status incomplete
|| T &
gfm-1

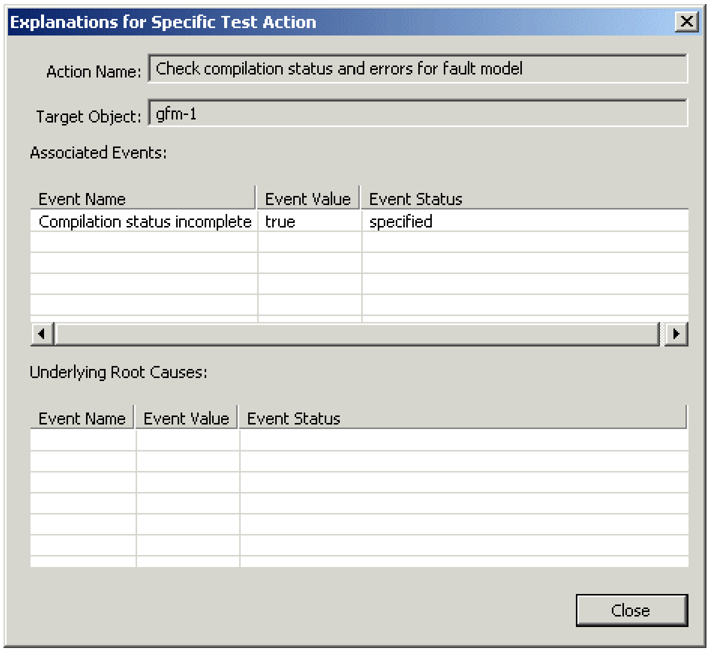
Explaining Actions

You can query each test or repair action for an explanation of the event that lead up to the action being created. The explanation comprises of the list of root causes tested or repaired by the external action.

To explain a test or repair action:

➔ Select a test or repair action and click the Explanation toolbar button: ?

Here is the explanation of the “Check compilation status and errors for fault model” test action for gfm-1:



Explanations for Specific Test Action

Action Name: Check compilation status and errors for fault model

Target Object: gfm-1

Associated Events:

Event Name	Event Value	Event Status
Compilation status incomplete	true	specified

Underlying Root Causes:

Event Name	Event Value	Event Status

Executing Manual Tests and Repair Actions

A test or repair action can be automatic, in which case it executes whenever it is enabled, based on the enabling transition defined for the action. A manual test or repair action is enabled when the value of the underlying event changes, according to its enabling transition; however, you must execute the action manually.

For more information on enabling transitions, see Creating Generic External Actions.

To execute a manual test or repair action:

➔ Select a manual test or repair action and click the Run toolbar button: ⚙

The status of the test changes to “running”. When the test completes, the status changes to “inactive.”

Interacting with Specific Fault Models

You can interact with specific events and actions through a graphical display of the specific fault model. You can view and interact with a specific fault model by:

- Creating a dynamic view of a specific fault model for a specific event through one of the built-in browsers.
- Creating a dynamic view of a specific fault model managed by a diagnosis manager.

SymCure allows multiple clients to view and update specific fault models simultaneously.

For information on showing specific fault models through the built-in browsers, see Interacting with Alarms and Root Causes.

For information on debugging specific fault models, see Debugging SymCure Fault Models.

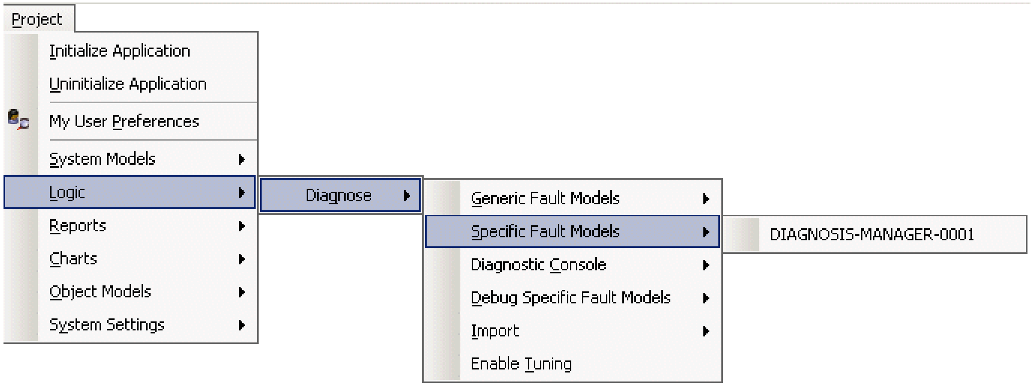
Showing Specific Fault Models

You can display a specific fault model from the Project menu or the Navigator.

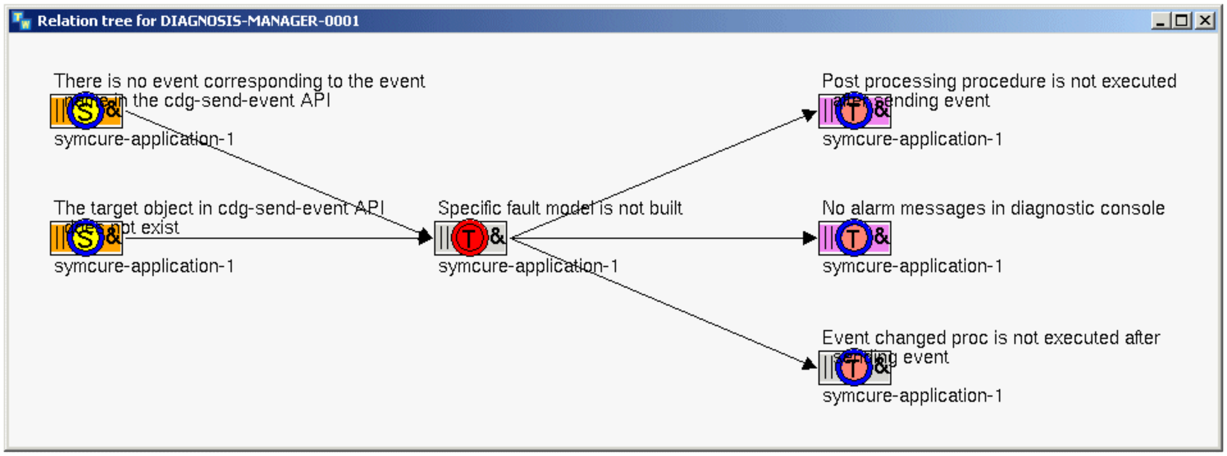
To show a specific fault model:

➔ Choose Project > Logic > Diagnose > Specific Fault Models menu.

The specific fault model associated with each diagnosis manager appears in the list, for example:



Here is a specific fault model:


Relation tree for DIAGNOSIS-MANAGER-0001
There is no event corresponding to the event value in the cdg-send-event API
symcure-application-1
The target object in cdg-send-event API does not exist
symcure-application-1
Specific fault model is not built
symcure-application-1
Post processing procedure is not executed after sending event
symcure-application-1
No alarm messages in diagnostic console
symcure-application-1
Event changed proc is not executed after sending event
symcure-application-1

Showing Specific Event Properties

Specific events define these properties:

- Event Name — The name of the specific event.
- Target Object — The name of domain object that is the target of the specific event.
- Event Value — The value of the specific event, which can be “true”, “false”, “Suspect”, or “unknown”.
- Event Status — The status of the specific event, which can be “specified”, “upstream inferred”, “downstream inferred”, or “mutually exclusive”.
- Timestamp — The last timestamp that the specific event received.
- Occurs At — The time at which the event occurred.
- Last Specified Value — The last value that was specified. This property is only relevant for specific events whose values are specified.
- Last Specified Timestamp — The time at which the event was last specified. This property is only relevant for specific events whose values are specified.

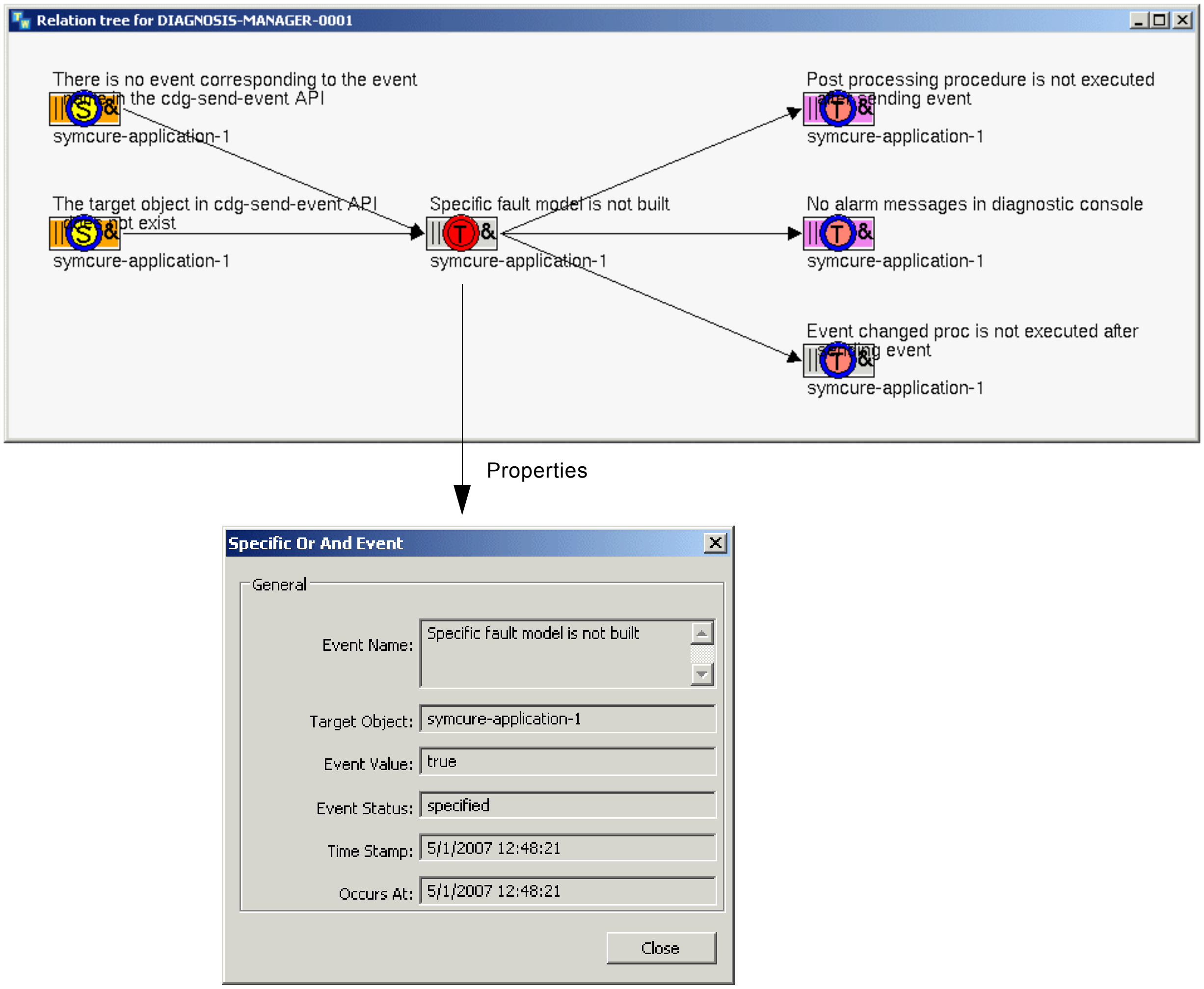
You can view the properties of a specific event through a specific fault model.

The icon of a specific event contains an outer circle, whose color indicates the last specified value, and an inner circle, whose color indicates the present value, which is either inferred or specified. The color-coding scheme is the same as the one used for event values: red=true, green=false, yellow=suspect, blue=unknown.

To show the specific event properties:

➔ Choose Properties on a specific event in a specific fault model.

Here is a specific fault model that includes the “Specific fault model is not built” specific event and its associated properties dialog:


Relation tree for DIAGNOSIS-MANAGER-0001
There is no event corresponding to the event defined in the cdg-send-event API
S &
symcure-application-1
The target object in cdg-send-event API does not exist
S &
symcure-application-1
Specific fault model is not built
T &
symcure-application-1
Post processing procedure is not executed after sending event
T &
symcure-application-1
No alarm messages in diagnostic console
T &
symcure-application-1
Event changed proc is not executed after sending event
T &
symcure-application-1
Properties
Specific Or And Event
General
Close

Event Name: Specific fault model is not built

Target Object: symcure-application-1

Event Value: true

Event Status: specified

Time Stamp: 5/1/2007 12:48:21

Occurs At: 5/1/2007 12:48:21

Associating User-Defined Attributes with Specific Events

SymCure provides support for quantitative information and reasoning about events in the form of probabilities and heuristic estimates. End user applications may find it useful to reason over the fraction of causes and effects that support the value of an event. Depending on the particular application, the fraction of true causes or effects supporting a true event may reflect the degree of degradation of the event.

For example, first suppose a generic service outage event is defined to occur if over 50% of the servers providing the service have failed. You typically model this scenario by using an N/M-AND event. Now, suppose that 80% of the servers providing the specific service have failed in the domain map. This knowledge, which is available in a specific fault model, may be used to indicate the extent of the service outage. Second, suppose that an AND-AND event is considered to be suspect and that *m* out of its *n* causes are true, which means the remaining *m* - *n* causes are suspect. Further, imagine there is no additional information forthcoming that can decisively determine whether the AND-AND event is true or false. The fraction *m*/*n* indicates the level of confidence in our belief that the AND-AND event is indeed true.

The fraction of causes or effects may be useful for any class of event, not just the fractional ones, and can be used for computing the degree of degradation or the level of confidence, as in the examples above.

Showing Detailed Explanations of Specific Events

You can show a detailed explanation about a specific event, which includes information about:

- Upstream events.
- Downstream events.

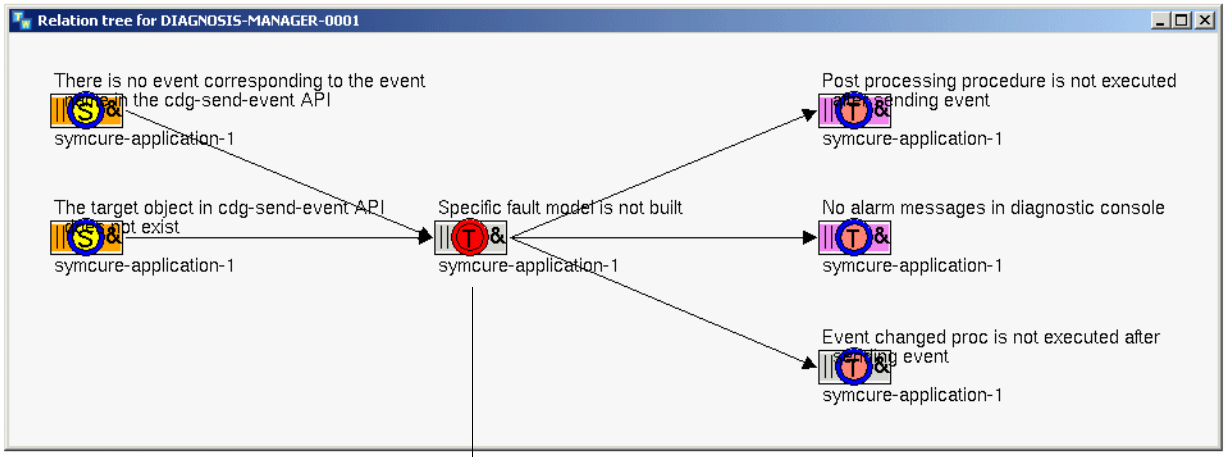
To show a detailed explanation about a specific event:

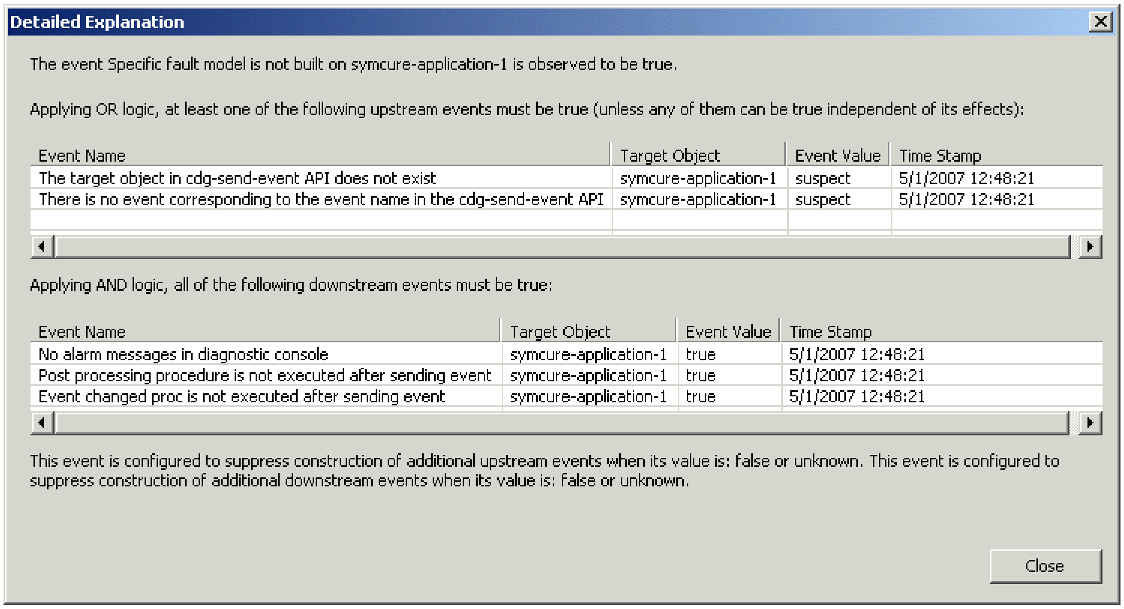
➔ Choose Detailed Explanation on the specific event.

or

➔ Click the Detailed Explanation button on an alarm or root cause event in the Alarms Browser or Root Causes Browser.

For example, here is the detailed explanation for the specific event “Specific fault model is not built on symcure-application-1:”


Relation tree for DIAGNOSIS-MANAGER-0001
There is no event corresponding to the event name in the cdg-send-event API
symcure-application-1
The target object in cdg-send-event API does not exist
symcure-application-1
Specific fault model is not built
symcure-application-1
Post processing procedure is not executed after sending event
symcure-application-1
No alarm messages in diagnostic console
symcure-application-1
Event changed proc is not executed after sending event
symcure-application-1
Detailed Explanation


Detailed Explanation

The event Specific fault model is not built on symcure-application-1 is observed to be true.

Applying OR logic, at least one of the following upstream events must be true (unless any of them can be true independent of its effects):

Event Name	Target Object	Event Value	Time Stamp
The target object in cdg-send-event API does not exist	symcure-application-1	suspect	5/1/2007 12:48:21
There is no event corresponding to the event name in the cdg-send-event API	symcure-application-1	suspect	5/1/2007 12:48:21

Applying AND logic, all of the following downstream events must be true:

Event Name	Target Object	Event Value	Time Stamp
No alarm messages in diagnostic console	symcure-application-1	true	5/1/2007 12:48:21
Post processing procedure is not executed after sending event	symcure-application-1	true	5/1/2007 12:48:21
Event changed proc is not executed after sending event	symcure-application-1	true	5/1/2007 12:48:21

This event is configured to suppress construction of additional upstream events when its value is: false or unknown. This event is configured to suppress construction of additional downstream events when its value is: false or unknown.

Setting a Specific Event Value

You can manually set the value of a specific event to true or false through the specific fault model.

This feature has the same effect as setting the specific event value through one of the built-in browsers. For details, see Setting the Event Value.

To set a specific event value:

➔ Choose True or False on a specific event.

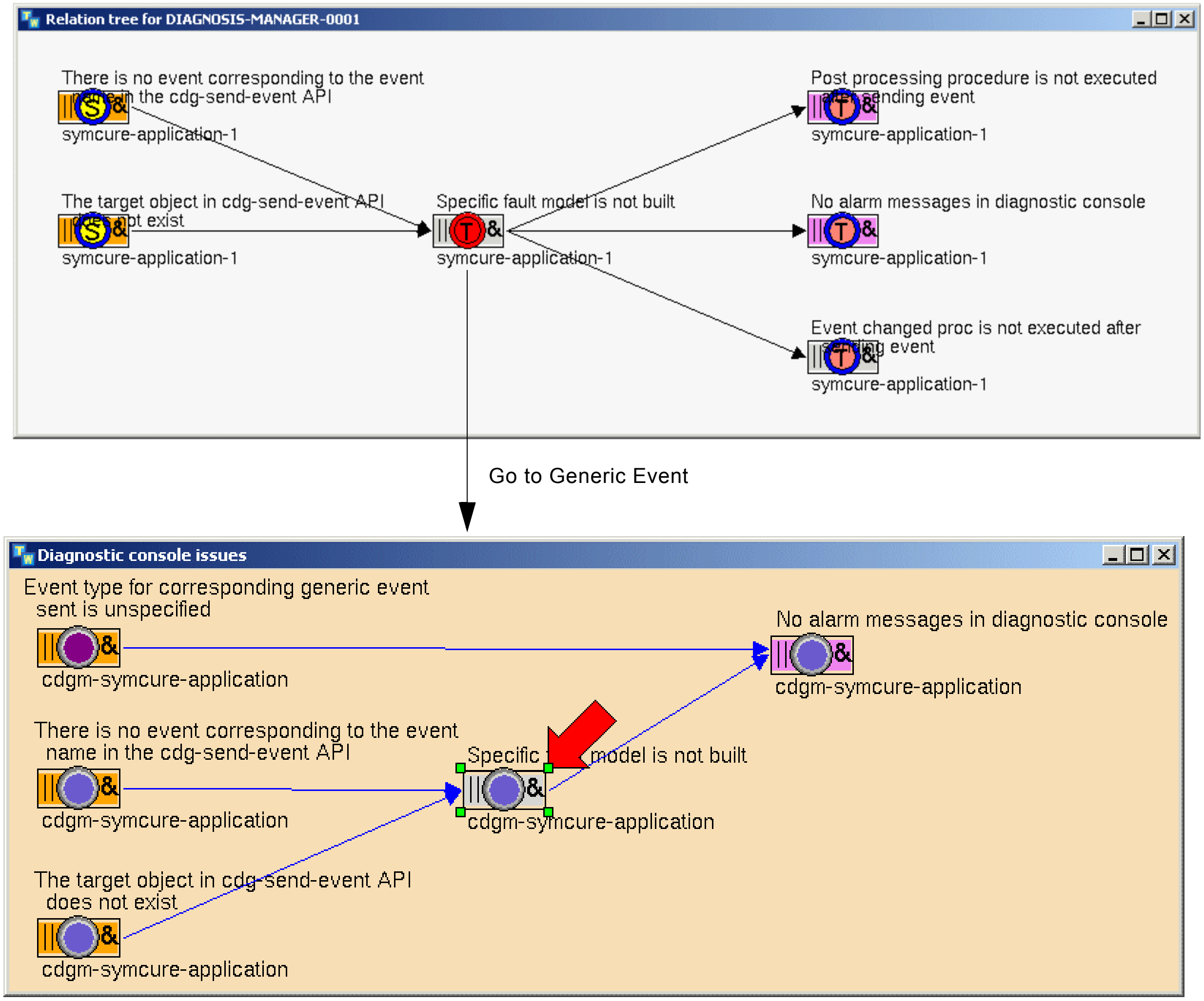
Showing the Generic Event for a Specific Event

You can show the generic event from a specific event.

To show the generic event for a specific event:

➔ Choose Go to Generic Event on a specific event.

Here is the generic event associated with the specific event named “Compilation status incomplete”:


Relation tree for DIAGNOSIS-MANAGER-0001
There is no event corresponding to the event name in the cdg-send-event API
symcure-application-1
The target object in cdg-send-event API does not exist
symcure-application-1
Specific fault model is not built
symcure-application-1
Post processing procedure is not executed after sending event
symcure-application-1
No alarm messages in diagnostic console
symcure-application-1
Event changed proc is not executed after sending event
symcure-application-1
Go to Generic Event
Diagnostic console issues
Event type for corresponding generic event sent is unspecified
cdgm-symcure-application
No alarm messages in diagnostic console
cdgm-symcure-application
There is no event corresponding to the event name in the cdg-send-event API
cdgm-symcure-application
Specific fault model is not built
cdgm-symcure-application
The target object in cdg-send-event API does not exist
cdgm-symcure-application

Showing the Specific Event Target

You can show the domain object that is the target of the specific event through the specific fault model.

This feature has the same effect as showing the specific event target through one of the built-in browsers. For details, see Showing the Event Target.

To show the specific event target:

➔ Choose Show Target on a specific event.

Showing the Event Summary

You can view a summary of all the objects that are related to a specific event through the specific fault model.

This feature has the same effect as showing the event summary through one of the built-in browsers. For details, see Showing the Event Summary.

To show the event summary:

➔ Choose Show Summary on a specific event.

Showing the Causal Model

You can show the causal model for a specific event through a specific fault model, which is a focused view of the specific fault model around a specific event.

This feature has the same effect as showing the specific fault model through one of the built-in browsers. For details, see Showing Causal Models.

To show the focused causal model for a specific event:

➔ Choose Show Causal Model on a specific event.

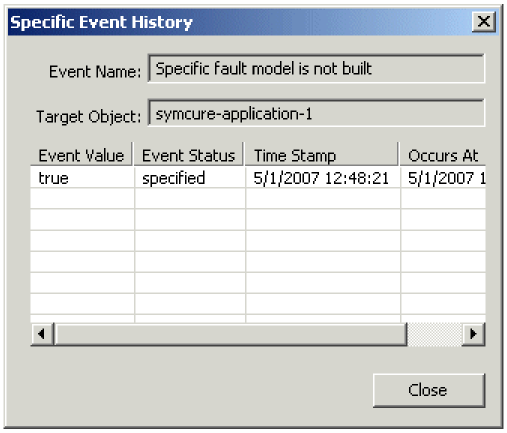
Showing the Specific Event History

You can show the history of a specific event through the specific fault model. The history includes the event value, event status, and timestamp. The event history does not include information on the current event.

To show the specific event history:

➔ Choose Show History on a specific event.

Here is the specific event history for the “Compilation status incomplete” event whose value changed from false to true:



Event Name: Specific fault model is not built

Target Object: symcure-application-1

Event Value	Event Status	Time Stamp	Occurs At
true	specified	5/1/2007 12:48:21	5/1/2007 1

Showing Specific Action Properties

Specific actions define these properties:

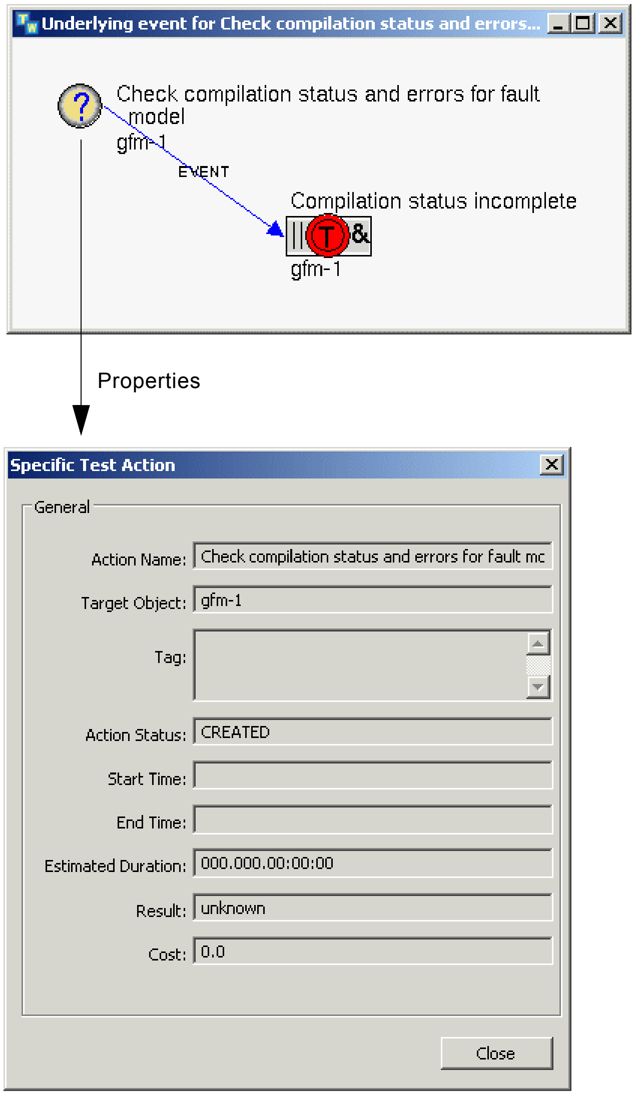
- Action Name — The name of the specific action.
- Target Object — The name of domain object that is the target of the specific event.
- Tag — A unique ID for the specific action.
- Action Status — The status of the specific action.
- Start Time — The time at which the specific action started executing.
- End Time — The time at which the specific action finished executing.
- Estimated Duration — The estimated amount of time it took to execute the action.
- Result — The result of running the test.
- Cost — The cost of running the test.

You can view the properties of a specific action through a specific fault model or through the various message browsers, including the Test and Repair Actions Browsers.

To show specific action properties:

➔ Select a specific test or repair action and choose Properties.

Here is the “Compilation status incomplete” underlying event for the “Check compilation status and errors for fault model” specific test action and the properties dialog for the specific action:


Underlying event for Check compilation status and errors...
Check compilation status and errors for fault model
gfm-1
EVENT
Compilation status incomplete
T
&
gfm-1
Properties
Specific Test Action
General
Close

Action Name: Check compilation status and errors for fault mc

Target Object: gfm-1

Tag:

Action Status: CREATED

Start Time:

End Time:

Estimated Duration: 000.000.00:00:00

Result: unknown

Cost: 0.0

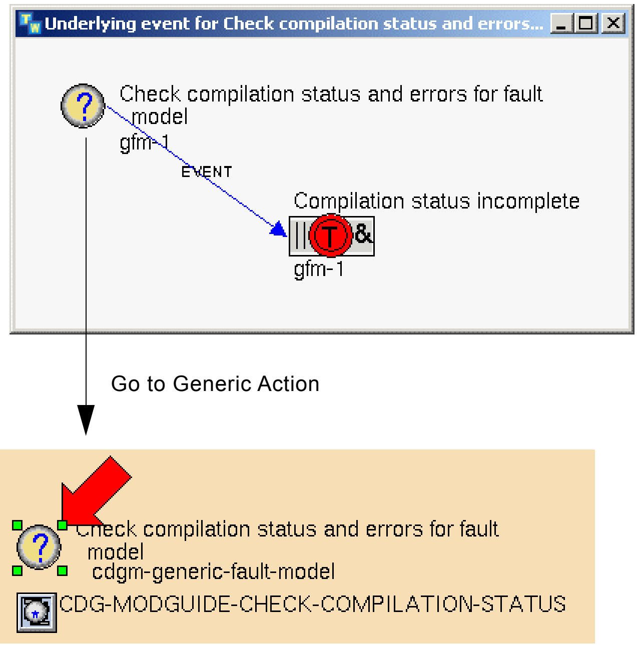
Showing the Generic Action for a Specific Action

You can show the generic action from a specific action.

To show the generic action for a specific action:

➔ Choose Go to Generic Action on a specific action.

Here is the generic action associated with the specific action named "Check compilation status and errors for fault model":


Underlying event for Check compilation status and errors...
Check compilation status and errors for fault model
gfm-1
EVENT
Compilation status incomplete
T
&
gfm-1
Go to Generic Action
Check compilation status and errors for fault model
cdgm-generic-fault-model
CDG-MODGUIDE-CHECK-COMPILATION-STATUS

Running a Specific Action

You can run a specific action from a specific fault model.

This feature has the same effect as running the specific action through the Test or Repair Actions Browser. For details, see Executing Manual Tests and Repair Actions.

To run a specific action:

➔ Choose Run Action on a specific action.

This menu choice is only available when the status is not running. When the status of a specific action is running, the Run Action menu choice is disabled.

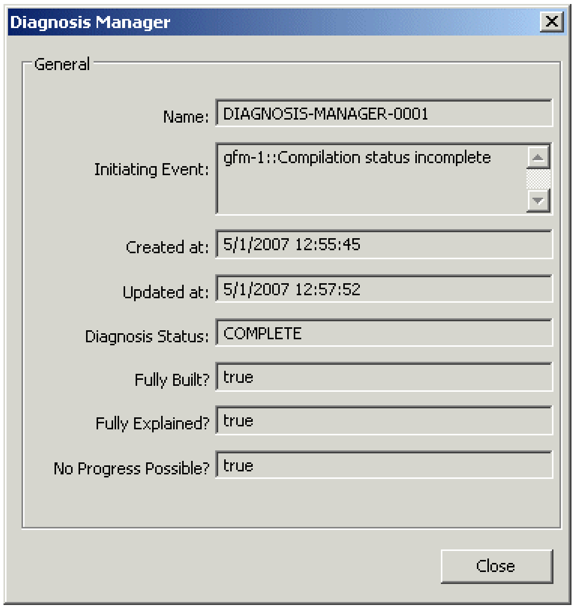
Showing the Properties of the Diagnosis Manager

Diagnosis managers define various properties, which you can view. For an explanation of these properties, see Diagnosis Managers.

To show the properties of a diagnosis manager:

➔ Choose Properties on the background of a specific fault model.

Here is the properties dialog of the diagnosis manager associated with a specific fault model:



Diagnosis Manager

General

Name: DIAGNOSIS-MANAGER-0001

Initiating Event: gfm-1::Compilation status incomplete

Created at: 5/1/2007 12:55:45

Updated at: 5/1/2007 12:57:52

Diagnosis Status: COMPLETE

Fully Built? true

Fully Explained? true

No Progress Possible? true

For a description of these properties, see Getting Diagnosis Information.

Showing Diagnostic Console Browsers for Individual Diagnosis Managers

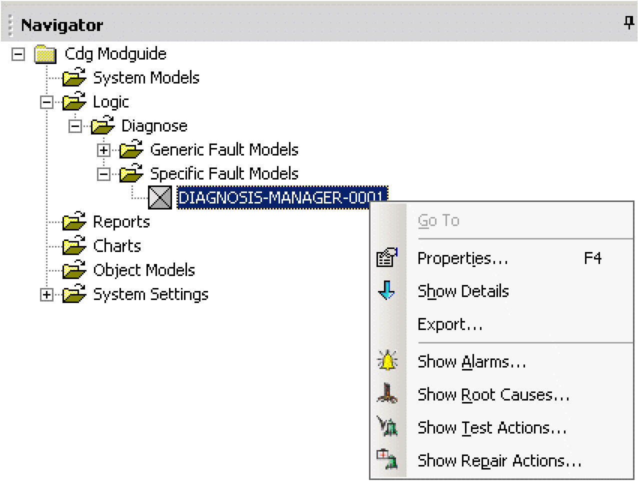
Each specific fault model represents an isolated problem. The diagnostic console browsers and the operator message browser combine messages from all root causes, alarms, tests, and repair actions.

You can show alarms, root causes, test actions, and repair actions for individual diagnosis managers, from a diagnosis manager in the Navigator or from a specific fault model.

Note that unlike the diagnostic console browsers, which update dynamically, these dialogs are static. Furthermore, unlike the diagnostic console browsers, these dialogs do not allow you to interact with the events. When the underlying diagnosis manager is deleted, all associated open dialogs are automatically deleted.

To show events for individual diagnosis managers:

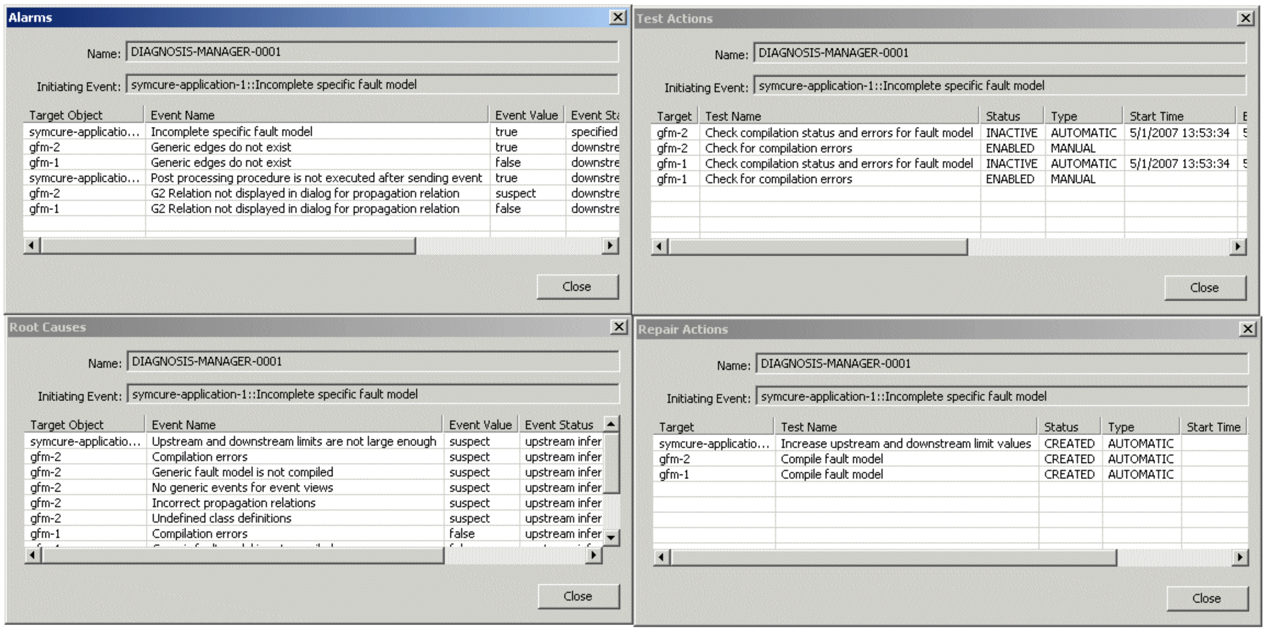
➔ In the Navigator, expand the Logic > Diagnose > Specific Fault Models tree node, mouse right on an individual diagnosis manager, and choose Show Alarms, Show Root Causes, Show Test Actions, or Show Repair Actions:



or

➔ Mouse right on a specific fault model workspace and choose Show Alarms, Show Root Causes, Show Test Actions, or Show Repair Actions.

For example, here is the Alarms, Root Causes, Test Actions, and Repair Actions browsers for an individual diagnosis manager:



Alarms

Name: DIAGNOSIS-MANAGER-0001

Initiating Event: symcure-application-1::Incomplete specific fault model

Target Object	Event Name	Event Value	Event Sta
symcure-applicatio...	Incomplete specific fault model	true	specified
gfm-2	Generic edges do not exist	true	downstre
gfm-1	Generic edges do not exist	false	downstre
symcure-applicatio...	Post processing procedure is not executed after sending event	true	downstre
gfm-2	G2 Relation not displayed in dialog for propagation relation	suspect	downstre
gfm-1	G2 Relation not displayed in dialog for propagation relation	false	downstre

Test Actions

Name: DIAGNOSIS-MANAGER-0001

Initiating Event: symcure-application-1::Incomplete specific fault model

Target	Test Name	Status	Type	Start Time	E
gfm-2	Check compilation status and errors for fault model	INACTIVE	AUTOMATIC	5/1/2007 13:53:34	5
gfm-2	Check for compilation errors	ENABLED	MANUAL		
gfm-1	Check compilation status and errors for fault model	INACTIVE	AUTOMATIC	5/1/2007 13:53:34	5
gfm-1	Check for compilation errors	ENABLED	MANUAL		

Root Causes

Name: DIAGNOSIS-MANAGER-0001

Initiating Event: symcure-application-1::Incomplete specific fault model

Target Object	Event Name	Event Value	Event Status
symcure-applicatio...	Upstream and downstream limits are not large enough	suspect	upstream infer
gfm-2	Compilation errors	suspect	upstream infer
gfm-2	Generic fault model is not compiled	suspect	upstream infer
gfm-2	No generic events for event views	suspect	upstream infer
gfm-2	Incorrect propagation relations	suspect	upstream infer
gfm-2	Undefined class definitions	suspect	upstream infer
gfm-1	Compilation errors	false	upstream infer

Repair Actions

Name: DIAGNOSIS-MANAGER-0001

Initiating Event: symcure-application-1::Incomplete specific fault model

Target	Test Name	Status	Type	Start Time
symcure-applicatio...	Increase upstream and downstream limit values	CREATED	AUTOMATIC	
gfm-2	Compile fault model	CREATED	AUTOMATIC	
gfm-1	Compile fault model	CREATED	AUTOMATIC	

Refreshing Specific Fault Models

When an underlying event for a diagnosis manager changes, any specific fault model views that are currently visible are automatically refreshed. SymCure maintains any positional changes that you have made to the specific fault model when the model is refreshed.

You can also redraw any view of a specific fault model by choosing Project > Logic > Diagnose > Specific Fault Models menu, which causes it to revert to its original size and layout.

Deleting All Diagnoses

During development, you might want to delete all diagnoses.

To delete all diagnoses:

1 Click the Delete All Diagnoses button in the Fault Modeling toolbar:

2 Click OK in the confirmation dialog.

Learning Generic Models from Specific Events

You can “tune” an N/M-N/M event in a specific fault model by changing its Input Fraction, Output Fraction, and Independent Of Effects attributes, without requiring a change to the underlying generic model. This feature allows you to tinker with a specific model, then to apply the changes to the underlying generic fault model when you are satisfied with the settings.

To learn generic models from specific events:

1 Create a generic fault model that uses N/M-N/M events.

For details, see Configuring Generic N/M-N/M Events.

2 Choose Project > Logic > Diagnose > Enable Tuning or click the equivalent button in the Fault Modeling toolbar ( ).

Tuning is now enabled for specific N/M-N/M events.

3 Display a specific fault model or any view that includes a specific N/M-N/M event.

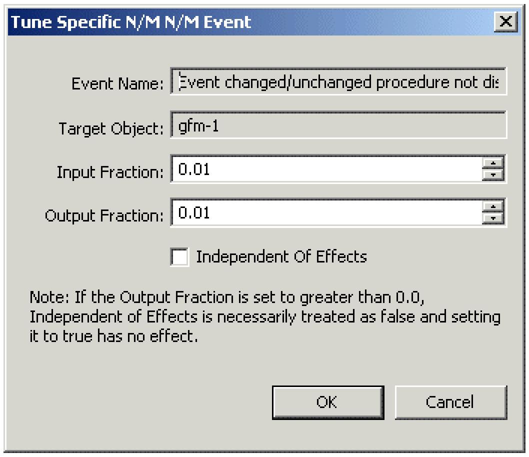
For details, see Interacting with Specific Fault Models.

4 Choose Tune Event on a specific event in the fault model.

A dialog appears for configuring the specific event.

5 Configure the properties of the event, as needed.

For example, this dialog sets the input and output fractions to 0.01:



Event Name: Event changed/unchanged procedure not di

Target Object: gfm-1

Input Fraction: 0.01

Output Fraction: 0.01

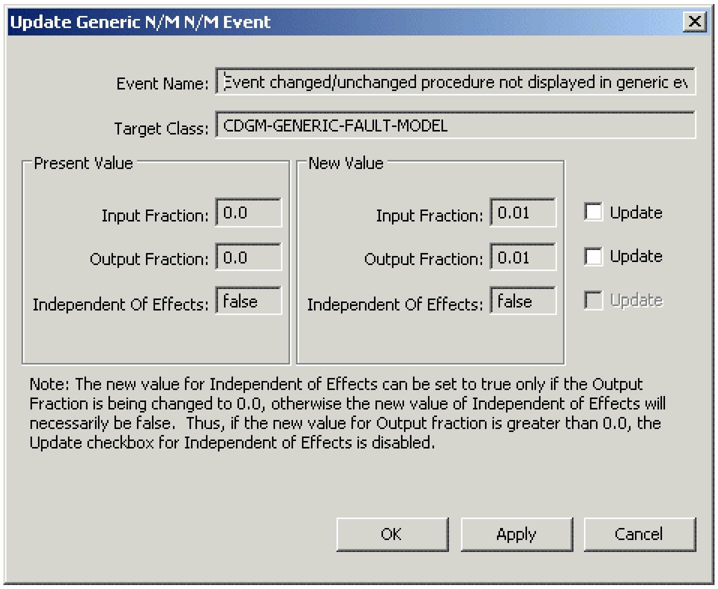
□ Independent Of Effects

Note: If the Output Fraction is set to greater than 0.0, Independent of Effects is necessarily treated as false and setting it to true has no effect.

The dialog notes that the Independent of Effects option can be changed to true only if the new output fraction is equal to 0.0.

6 To apply the values from the specific event to the generic event, choose Update Generic Event on the specific event.

SymCure displays a dialog that shows the present values and new values with options for updating each changed value, for example:



Event Name: Event changed/unchanged procedure not displayed in generic ev

Target Class: CDGM-GENERIC-FAULT-MODEL

Present Value		New Value		
Input Fraction:	0.0	Input Fraction:	0.01	□ Update
Output Fraction:	0.0	Output Fraction:	0.01	□ Update
Independent Of Effects:	false	Independent Of Effects:	false	□ Update

Note: The new value for Independent of Effects can be set to true only if the Output Fraction is being changed to 0.0, otherwise the new value of Independent of Effects will necessarily be false. Thus, if the new value for Output fraction is greater than 0.0, the Update checkbox for Independent of Effects is disabled.

Note Independent of Effects is meaningful only when Output Fraction is 0.0. Its Update check box is disabled when the old value and new value are the same and when the old value = false and the new Output Fraction > 0.0.

7 Click the Update option for each new value that you want to use and apply.

The generic event is now configured to use the values specified in the specific event.

8 When you are finished tuning the generic fault model, disable tuning.

Detecting Chattering Events

If event detection logic for two causally related events in a diagnostic fault model is inconsistent, this may result in infinite *chattering*. For example, an incoming symptom event may cause the fault model to infer that an underlying root cause event is true or suspect. At the same time, an automatic test associated with the root cause event may conclude that the root cause is false; consequently, the original symptom may also be inferred to be false. If the symptom has an associated test that is run automatically each time it changes value, the test may return true because event detection logic for the symptom and the root cause and their causal relationship are inconsistent. This situation can result in an indefinite causal propagation cycle.

SymCure detects that an event is chattering when the event gets the same value repeatedly over a short period of time, where the repetition count and duration are configurable parameters. In response to this situation, a suitable error message is generated and the diagnostic engine is taken offline. No further diagnostic progress is possible until SymCure is taken online again. Thus, a user can gracefully recover from this situation with some understanding of the nature of the problem.

You can configure parameters related to chattering events in the `config.txt` file. For details, see Chattering Events.

Note In general, SymCure is always online, ready to receive events and perform fault diagnosis. When SymCure detects chattering events, it automatically goes offline, in which case you must bring it back online manually.

SymCure also automatically goes offline while generic fault model folders are compiled, and automatically goes back online when the compilation is complete.

To bring SymCure back online manually:

➔ Click the Take Online button in the Fault Modeling toolbar ( ).

This button is only available when SymCure is offline.

Whenever SymCure detects a chattering event, it notifies any registered listeners that a cdg-chattering-event has occurred. The source of a cdg-chattering-event is the built-in object cdg-reported-events; thus any object interested in listening to this event must register with cdg-reported-events. For example, to register the object MyListener:

```
call grtl-add-event-listener
(cdg-reported-events, MyListener, the symbol cdg-chattering-event,
sequence())
```

Exporting and Importing Specific Fault Models

SymCure allows you to export specific fault models to XML files and import them back into SymCure. Exporting allows recovery whenever the server goes down, because archived specific fault models can be imported back SymCure.

By default, when a diagnostic problem is suitably resolved, its information is purged at periodic intervals to reclaim memory. Archiving the information associated with diagnosis managers and their associated specific fault models allows you to capture the history of diagnostic processing.

Exporting and importing specific fault models also allows you to perform what-if simulations in separate SymCure processes. To perform offline what-if simulations, you need access to the current state of a specific fault model in a separate SymCure process. You can achieve this by exporting one or more diagnosis managers from one process to another through the specific fault model export import capabilities. Once a specific fault model has been imported into a separate process, you can simulate any event for that model without impacting your deployed application.

When importing specific fault models:

- **Warning**: Do not import a specific fault model while the model exists in the SymCure process. This will create duplicate specific events, actions, and diagnosis managers resulting in a highly unstable application.
- If the action status of a specific action in the stored XML file is RUNNING, then as soon as it is imported, its activation procedure is executed.
- When you import a specific fault model, SymCure creates messages for the newly created specific events and actions. These messages are displayed in the general messages browser and the diagnostic console. Note that:
  - The repetition count of the messages is set to 0.
  - For the external action messages, the last update time stamps represent the times that messages are created after importation.

You can export and import specific fault models interactively or programmatically.

By default, SymCure exports the contents of the specific fault model detail to an XML file stored in the `archives` subdirectory of your installation directory, which you can configure in the `config.txt` file.

For information on exporting and importing specific fault models programmatically, see Exporting and Importing Fault Models.

For information about configuring startup parameters for exporting and importing, see Archiving.

To export a specific fault model:

➔ Choose Export on the background of a specific fault model.

To import a specific fault model folder:

➔ Choose Project > Logic > Diagnose > Import > Import Specific Fault Model or click the equivalent button in the Fault Modeling toolbar (📂)

When the file is successfully parsed, a corresponding specific fault model is created.

Debugging SymCure Fault Models

Describes run-time debugging tools for SymCure applications.

Introduction

You can step through the propagation of events in specific fault models. This feature helps you to understand the behavior of SymCure's event propagation algorithm and the propagation logic of different events, and to debug any fault models that you create for your application.

When debugging is enabled, SymCure creates an internal log of every incoming event, every edge that is traversed in the specific fault model, and every event that is processed in response to each incoming event. SymCure provides a user interface to navigate the debug log graphically.

Debugging does not require access to underlying specific events. The log contains all information required for debugging. Thus, you can debug fault models even after the underlying specific events have been deleted. Note that you must have access to the generic fault models and the target domain objects for proper operation of the debugging utility.

Enabling Debugging

By default, debugging is disabled. Thus, you must explicitly enable it in one of three ways:

- Interactively through the Project menu or Fault Modeling toolbar.
- At startup through the `config.txt` file.
- Programmatically by using an API procedure.

The log may become moderately expensive in terms of memory, so we recommend that you set cdg-enable-debugging to false during the deployment of any application. Use debugging in a deployed application only if you flush the log at regular intervals by disabling debugging with this API. You can empty the debugging log at any time by disabling debugging.

Note If you initialize process maps from the Project menu, debugging is disabled and the log is emptied.

To enable debugging interactively:

➔ Choose Project > Logic > Diagnose > Debug Specific Fault Models > Enable Debugging or click the equivalent button in the Fault Modeling toolbar ( ).

You can only enable and disable debugging through the menus in Modeler and Developer mode. The menu choice is grayed out in Operator mode. Once debugging is enabled, you can debug models in any of the following modes: Developer, Modeler, and Operator.

To disable debugging:

➔ When debugging is enabled, choose Enable Debugging or click the toolbar button, and confirm that you want to disable debugging.

To enable debugging at startup:

➔ Set this parameter to true in the configuration file:

```
cdg-enable-debugging=true
```

For details on this parameter, see Debugging.

To enable debugging programmatically:

➔ Use this API procedure:

```
cdg-enable-fault-model-debugging
    (enable: truth-value)
```

For details on this API procedure, see Debugging.

Debugging Modes

SymCure provides two debugging modes:

- Sequential
- Parallel

Debugging in either mode processes events in the exact same order. The difference between these two modes arises when an event has one or more causes and effects. In sequential mode, the debugger visits events and their causes and effects through a series of steps, one event at a time. In parallel mode, the debugger first visits an event, next it propagates values to all neighboring events in parallel, then visits each of the neighboring events.

Accessing the Debugger

You can access the debugger from the menu bar or toolbar by choosing either sequential or parallel mode. These menu choices are only available when debugging is enabled.

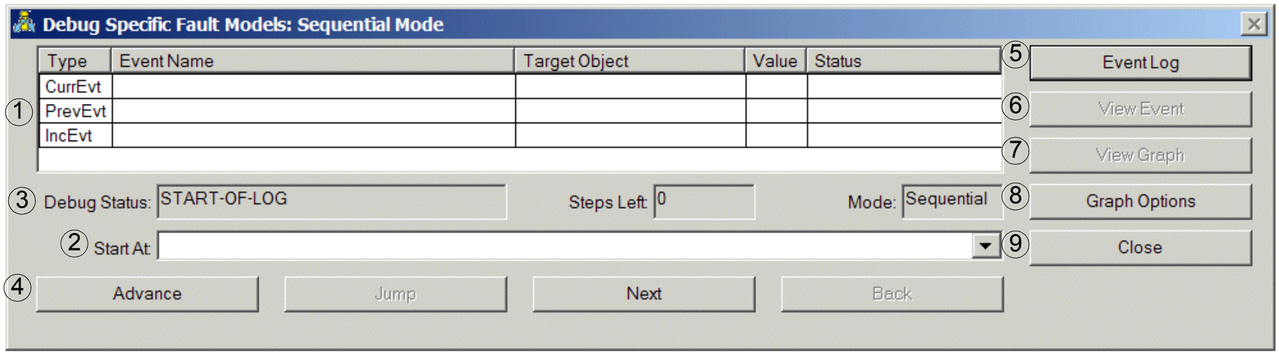
To access the debugger:

➔ Choose Project > Logic > Diagnose > Debug Specific Fault Models > Sequential Mode or Parallel Mode or click the equivalent buttons in the Fault Modeling toolbar ( )

Selecting either Sequential Mode or Parallel Mode initiates a new debugging session and displays the debugging control panel, which manages the debugging process. The debugging mode for a session is indicated by Mode in the control panel. You can access the debug display workspace by clicking the View button.

Each debugging session uses a local copy of the log of the specific events. Thus, different clients can initiate multiple simultaneous debugging sessions. However, note that currently, only one debugging session is permitted per client. The debug log is cleared whenever you toggle between enabling and disabling debugging.

The following figure shows the debugging control panel. The next section describes each labeled sections of the control panel.



Type	Event Name	Target Object	Value	Status
CurrEvt				
PrevEvt				
IncEvt				

Debug Status: START-OF-LOG

Steps Left: 0

Mode: Sequential

Start At:

The Control Panel

The control panel is divided into the following sections.

1 Event Navigation Table

2 Start At

3 Status Indicators

4 Debug Model Buttons

5 Event Log Button

6 View Event Button

7 View Graph Button

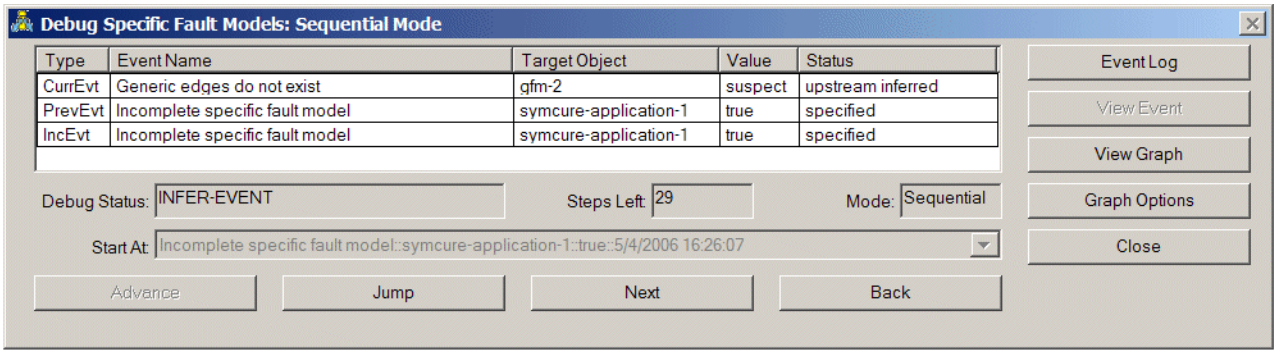
8 Graph Options Button

9 Close Button

Event Navigation Table

Event propagation begins in response to an incoming event. During debugging, the Event Navigation table displays the latest incoming event in the third row (Incoming Event). Event propagation traverses the edges of specific fault models to compute values for each visited event. The latest event to have its value computed is displayed in the first row (Current Event). The cause, effect, or mutually exclusive event that precedes the current event is displayed in the second row (Previous Event). The previous event is relevant to debugging because it is often a significant influence on the current event.

This figure shows the control panel and the corresponding debug display workspace with several events:



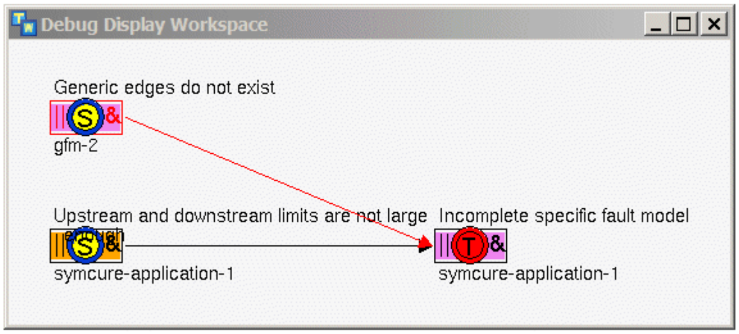
Type	Event Name	Target Object	Value	Status
CurrEvt	Generic edges do not exist	gfm-2	suspect	upstream inferred
PrevEvt	Incomplete specific fault model	symcure-application-1	true	specified
IncEvt	Incomplete specific fault model	symcure-application-1	true	specified

Debug Status: INFER-EVENT

Steps Left: 29

Mode: Sequential

Start At: Incomplete specific fault model::symcure-application-1::true::5/4/2006 16:26:07



Debug Display Workspace
Generic edges do not exist
S
&
gfm-2
Upstream and downstream limits are not large enough
S
&
symcure-application-1
Incomplete specific fault model
T
&
symcure-application-1

Start At

The Start At field provides a dropdown list of all logged incoming events sorted in temporal order. You can "jump start" the debugging process to a chosen incoming event by selecting it from this list. The Advance button jumps to the beginning of the selected incoming event, and debugging starts from the selected event. If no event is selected, debugging starts from the first incoming event in the internal debug log.

Status Indicators

The control panel has two status indicators:

- Debug Status indicates the status of the debugging process as described by the following symbols:
  - START-OF-LOG indicates that the debugging process is at its beginning.
  - INCOMING-EVENT (IncEvt) indicates that the current event displayed in the event navigation table is an incoming event.
  - INFER-EVENT (InfEvt) indicates that the current event in the event navigation table was processed by the diagnostic algorithm in an attempt to infer a value by traversing an edge in the specific fault model or through mutual exclusion.

- END-OF-PROPAGATION is used at the end of Jump to indicate that the end of processing an incoming event.
- EXPAND-EVENT is used only in Parallel mode and indicates the propagation of values to the neighbors of the current event.
- DELETED-EVENT (DelEvt) indicates that the event has been deleted.
- INCREMENTAL-PROCESSING (IncrPro) indicates that diagnostic propagation is driven not by an incoming event but by a new wave of incremental processing. For more information on incremental processing, see Specific Fault Model Display.
- END-OF-LOG indicates that the end of the log has been reached and the debugging session is complete.
- EMPTY-LOG indicates that the debug log is empty.

- Steps Left shows a running count of the remaining items in the debug log. Pressing Next decrements the value and pressing Back increments it. You can use this status indicator to recognize when the end of the debug log is being reached

  When the count reaches 0, there are no more events in the log. In Sequential mode, you can still click the Back button; pressing Next after the count reaches 0 takes you to the end of the event log at which point you can no longer go back. In Parallel mode, you can click Next once more after the count reaches 0 and still go back; pressing Next a second time takes you to the end of the log and you can no longer go back.
- Mode specifies the debugging mode (Sequential or Parallel) for the debugging process.

Debug Model Buttons

The Debug dialog has these buttons for debugging the model:

- Advance advances the debugging process to any incoming event that is selected from the Incoming Events list. Select the desired event in this list and click Advance. This action displays specific events that result from processing all incoming events starting from the first logged event up to the selected incoming event. You can advance only once at the start of the debugging process.
- Jump moves the debugging process to the end of the latest incoming event. This button is enabled whenever there is a new incoming event (Debug Status = INCOMING-EVENT) and is disabled at the end of the incoming event (Debug Status = END-OF-INCOMING-EVENT).

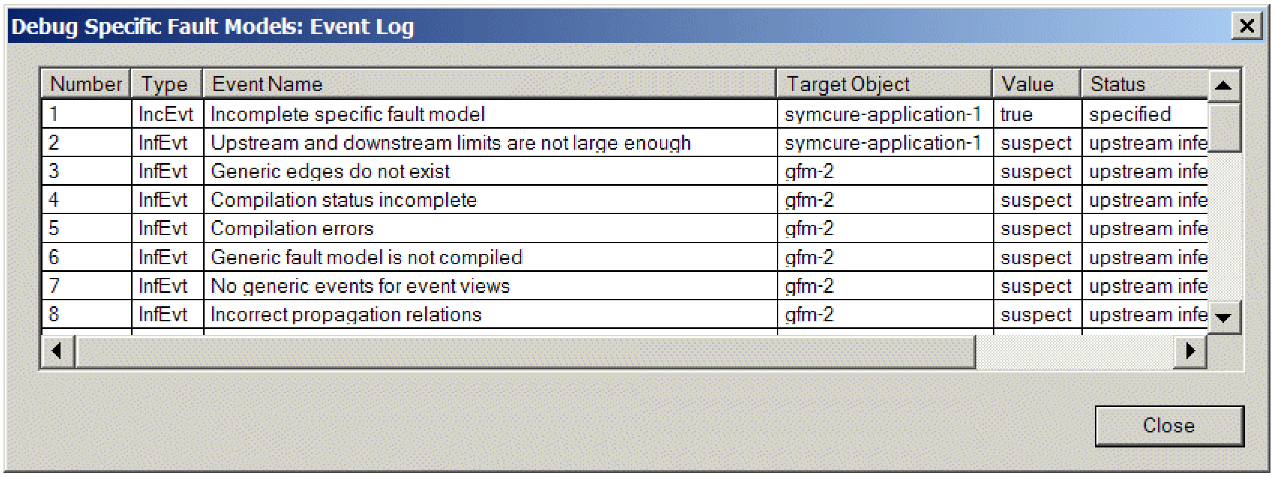
- Next moves the debugging process forward as follows, depending on the mode:
  - In Sequential mode, clicking Next shows the next event encountered by the event propagation algorithm. In the debug display workspace, the incoming event or the edge traversed in this propagation and the current event are highlighted in red.
  - In Parallel mode, clicking Next alternates between navigating to the next event and expanding the current event by showing its causes and effects in parallel. When the next propagation is shown, the edge traversed by the propagation algorithm and the current event are highlighted in red. When the expansion of the current event is shown, only the event is highlighted.

  In either mode, when the end of the log is reached (Debug Status = END-OF-LOG), the Next button is disabled.

- Back moves the debugging process backward by undoing the effect of the last Next step. Back does not allow you to undo the effects of a Jump or Advance.

Event Log Button

The Event Log is a static table showing the contents of the internal debugging log used for the debugging session, for example:



Debug Specific Fault Models: Event Log

Number	Type	Event Name	Target Object	Value	Status
1	IncEvt	Incomplete specific fault model	symcure-application-1	true	specified
2	InfEvt	Upstream and downstream limits are not large enough	symcure-application-1	suspect	upstream infe
3	InfEvt	Generic edges do not exist	gfm-2	suspect	upstream infe
4	InfEvt	Compilation status incomplete	gfm-2	suspect	upstream infe
5	InfEvt	Compilation errors	gfm-2	suspect	upstream infe
6	InfEvt	Generic fault model is not compiled	gfm-2	suspect	upstream infe
7	InfEvt	No generic events for event views	gfm-2	suspect	upstream infe
8	InfEvt	Incorrect propagation relations	gfm-2	suspect	upstream infe

The Type column indicates the corresponding event type shown as the current event in the Event Navigation Table. The possible status values are:

- IncEvt = Incoming Event
- InfEvt = Infer Event
- DelEvt = Deleted Event

View Event Button

You can locate the Current Event, Previous Event, and Incoming Event, as shown in the Event Navigation table, in the debug display workspace by selecting an event in the table and clicking the View Event button. A red arrow points to the event in the debug display workspace.

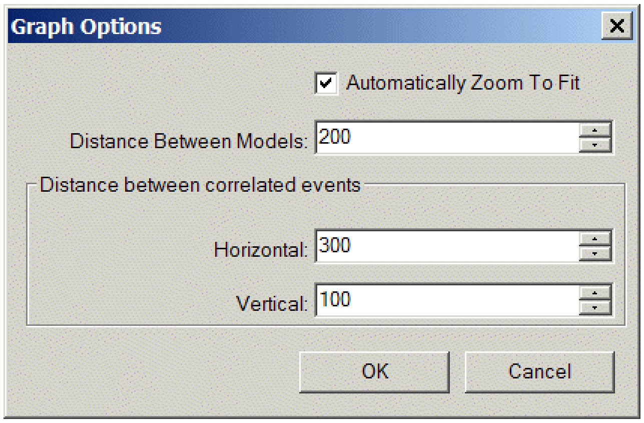
View Graph Button

Click the View Graph button to show the debug display workspace for the specific fault models constructed during the debugging process. Click the Hide button to hide the view.

This button is disabled when the display workspace is empty; it is enabled as soon as the display workspace is populated with specific event display objects.

Graph Options Button

Click the Graph Options button to configure options for displaying the specific fault model in the graph. You can configure the distance between disjoint specific models, and the vertical and horizontal distance between correlated specific events, in pixels. By default, the specific fault model workspace is automatically zoomed to fit the fault model. Here are the graph options you can configure:



Graph Options

☑ Automatically Zoom To Fit

Distance Between Models: 200

Distance between correlated events

Horizontal: 300

Vertical: 100

Close Button

Click the Close button to end the debugging session, close the control panel, and delete the debug display workspace. Note that clicking Close is the only safe way to delete the graphical display workspace. You must confirm that you want to exit the debugging session by selecting Yes in the confirmation dialog, or select No to continue with the debugging session.

The Debug Display Workspace

The following menu choices are available for the objects in the debug display workspace:

- Properties
- Go To Generic Event
- Show Target
- Show History

Debugging with Sequential and Parallel Mode

SymCure uses heuristic best first search to propagate event values for a specific event. At a very high level, starting from an incoming event, the algorithm for propagating event values is as follows:

```
for any event e when its value changes do
    propagate the value of the event upstream to all causes;
    propagate the value of the event downstream to effects;
end for
```

where:

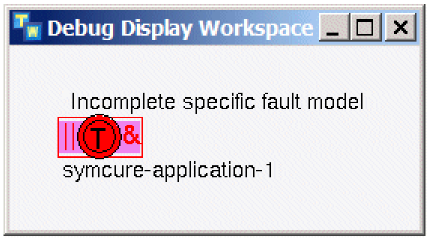
- *causes* are all events that are upstream of event e.
- *effects* include all events that are downstream of causes plus event e itself.

The examples below illustrate the propagation of events in sequential and parallel modes. Each step is generated by clicking the Next button in the Debug Model section of the control panel.

Sequential Mode

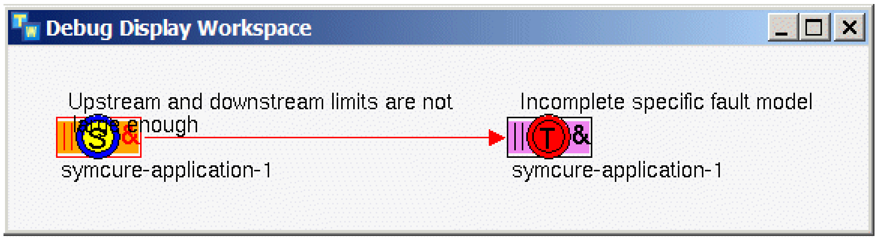
In sequential mode, SymCure constructs and propagates events through a series of steps, one event at a time, as follows. The debugger uses red outline and paths to indicate the current event, which are circled in the following diagrams.

1 Displays the incoming event “Incomplete specific fault model” on symcure-application-1 is specified as true. “Incomplete specific fault model” is also the Current Event.



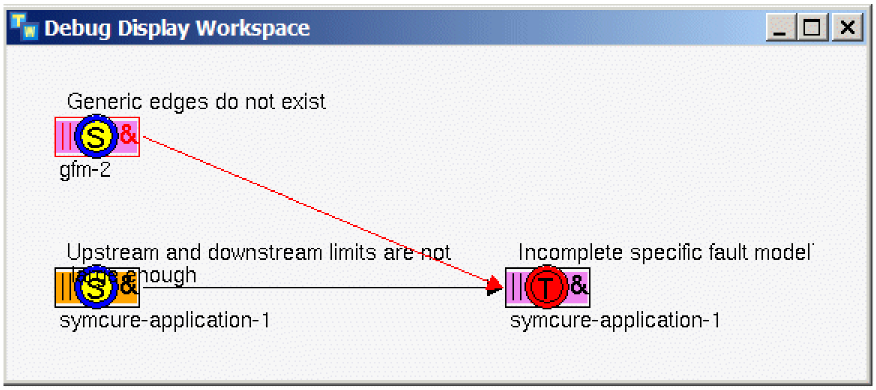
The color red means that the specified event is true.

2 Shows that “Upstream and downstream limits are not large enough” on symcure-application-1 is a suspected cause of “Incomplete specific fault model”. “Upstream and downstream limits are not large enough” is now the Current Event, while “Incomplete specific fault model” is both the Incoming Event and the Previous Event.

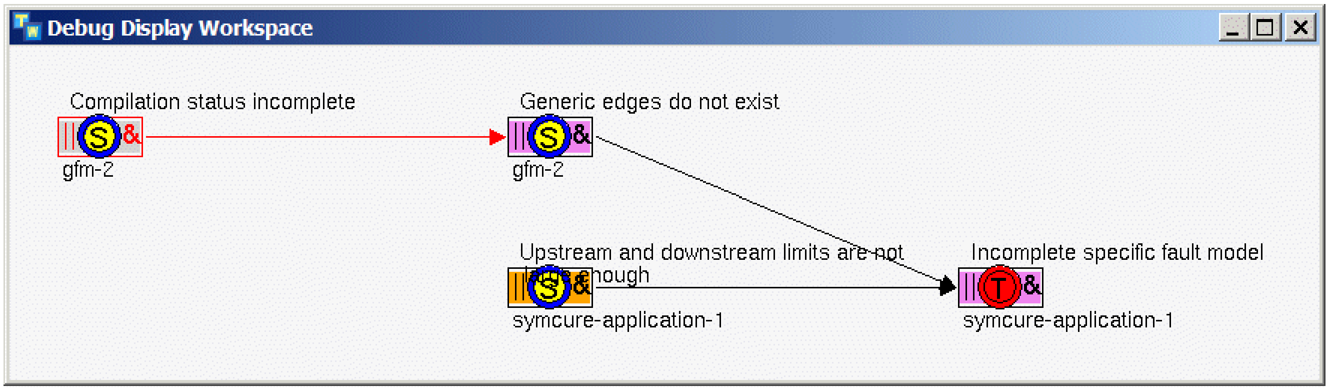

Debug Display Workspace
Upstream and downstream limits are not large enough
S
&
symcure-application-1
Incomplete specific fault model
T
&
symcure-application-1

The color yellow means the specified event is suspect.

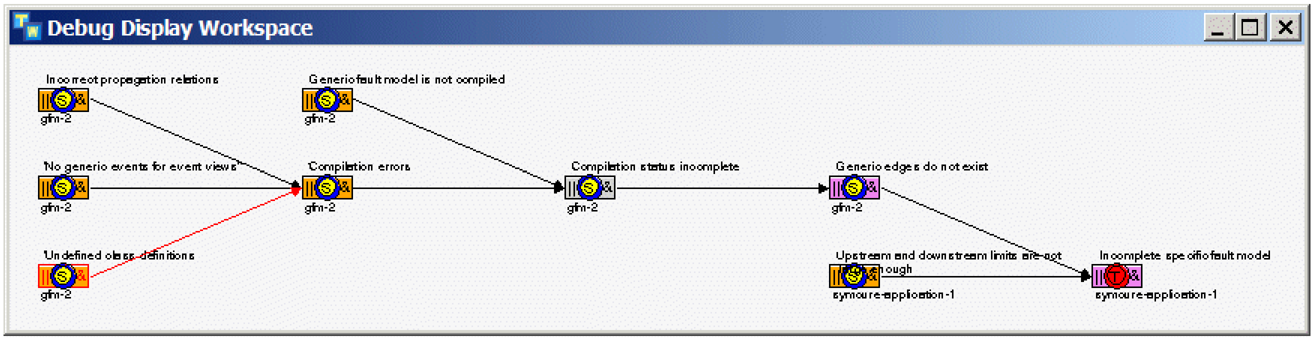
3 Constructs another suspected cause “Generic edges do not exist” on gfm-2 of “Incomplete specific fault model”. “Generic edges do not exist” is now the Current Event, while “Incomplete specific fault model” is both the Incoming Event and the Previous Event.


Debug Display Workspace
Generic edges do not exist
gfm-2
Upstream and downstream limits are not large enough
symcure-application-1
Incomplete specific fault model
symcure-application-1

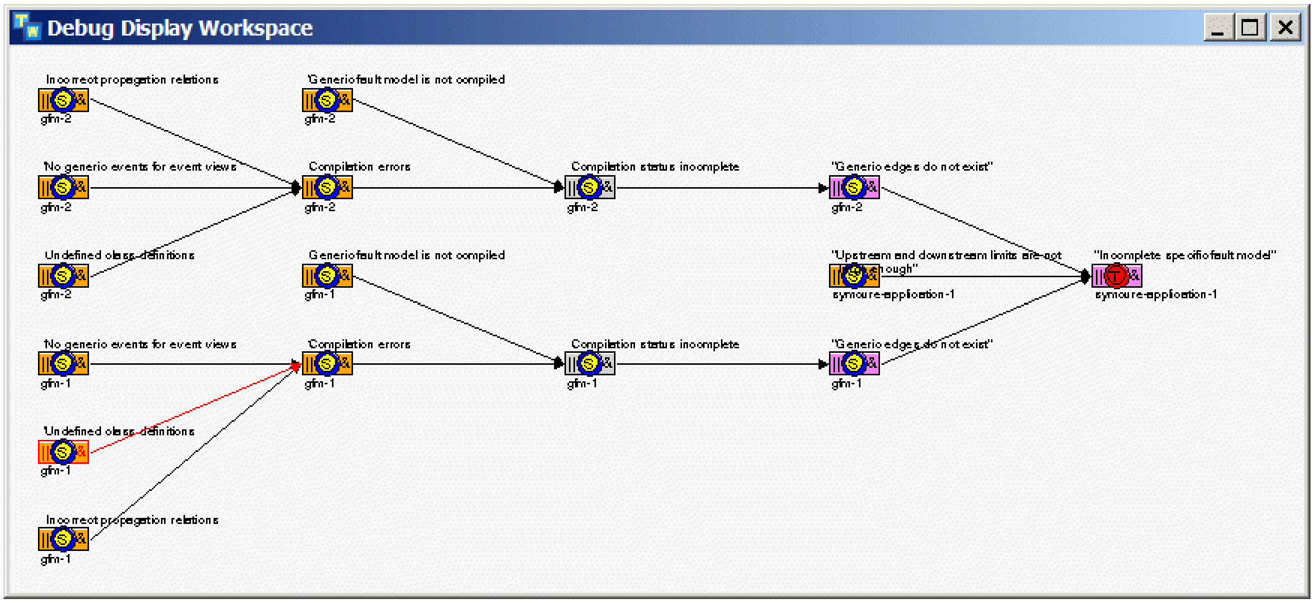
4 Begins to construct the upstream causes of “Generic edges do not exist” on gfm-2.


Debug Display Workspace
Compilation status incomplete
gfm-2
Generic edges do not exist
gfm-2
Upstream and downstream limits are not large enough
symcure-application-1
Incomplete specific fault model
symcure-application-1

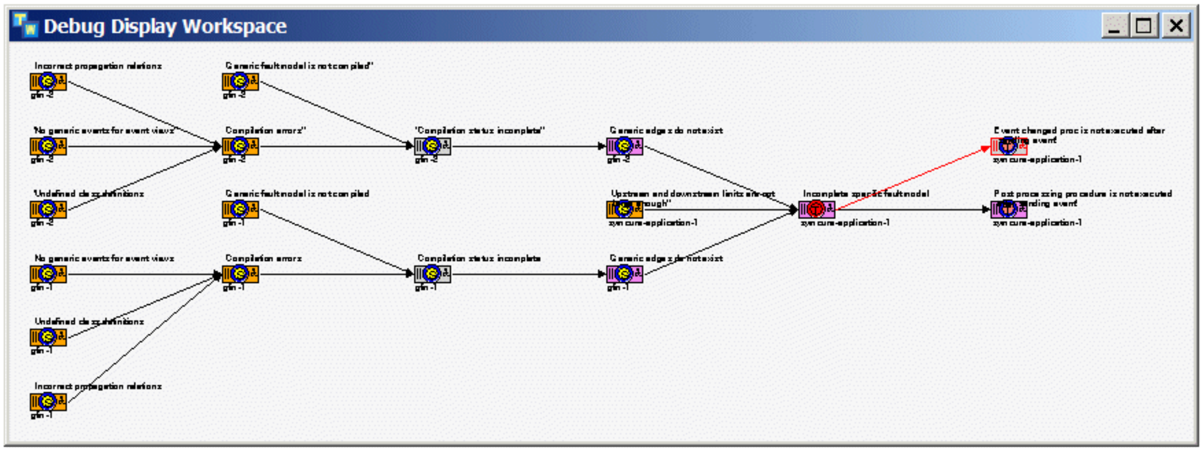
5 Constructs all the upstream causes of “Generic edges do not exist” on gfm-2. Note that in sequential mode, SymCure builds all upstream causes of each upstream cause until it gets to the root causes before it goes on to the next upstream cause. This figure shows the results after clicking Next five times.


Debug Display Workspace
Incorrect propagation relations
gfm-2
Generic fault model is not compiled
gfm-2
No generic events for event views"
gfm-2
Compilation errors
gfm-2
Compilation status incomplete
gfm-2
Generic edges do not exist
gfm-2
Undefined class definitions
gfm-2
Upstream and downstream limits are not large enough
symcure-application-1
Incomplete specific fault model
symcure-application-1

6 Constructs another suspected cause “Upstream and downstream limits are not large enough” on symcure-application-1 of “Incomplete specific fault model” and all its suspected causes up to the root causes. This figure shows the results after clicking Next seven times.


Debug Display Workspace
Incorrect propagation relations
gfm-2
"Generic fault model is not compiled
gfm-2
No generic events for event views
gfm-2
Compilation errors
gfm-2
Compilation status incomplete
gfm-2
"Generic edges do not exist"
gfm-2
Undefined class definitions
gfm-2
Generic fault model is not compiled
gfm-1
"Upstream and downstream limits are not large enough"
symcure-application-1
"Incomplete specific fault model"
symcure-application-1
No generic events for event views
gfm-1
Compilation errors
gfm-1
Compilation status incomplete
gfm-1
"Generic edges do not exist"
gfm-1
Undefined class definitions
gfm-1
Incorrect propagation relations
gfm-1

7 After constructing all upstream causes up to the root causes, constructs the two downstream effects of “Incomplete specific fault model”. This figure shows the results after clicking Next twice.


Debug Display Workspace
Incorrect propagation relations
gfn-2
Generic fault model is not compiled"
gfn-2
No generic events for event views"
gfn-2
Compilation errors"
gfn-2
"Compilation status incomplete"
gfn-2
Generic edges do not exist
gfn-2
Event changed proc is not executed after
ending event
syn cure-application-1
Undefined class definitions
gfn-2
Generic fault model is not compiled
gfn-1
Upstream and downstream limits are not
large enough"
syn cure-application-1
Incomplete specific fault model
syn cure-application-1
Post processing procedure is not executed
ending event
syn cure-application-1
No generic events for event views
gfn-1
Compilation errors
gfn-1
Compilation status incomplete
gfn-1
Generic edges do not exist
gfn-1
Undefined class definitions
gfn-1
Incorrect propagation relations
gfn-1

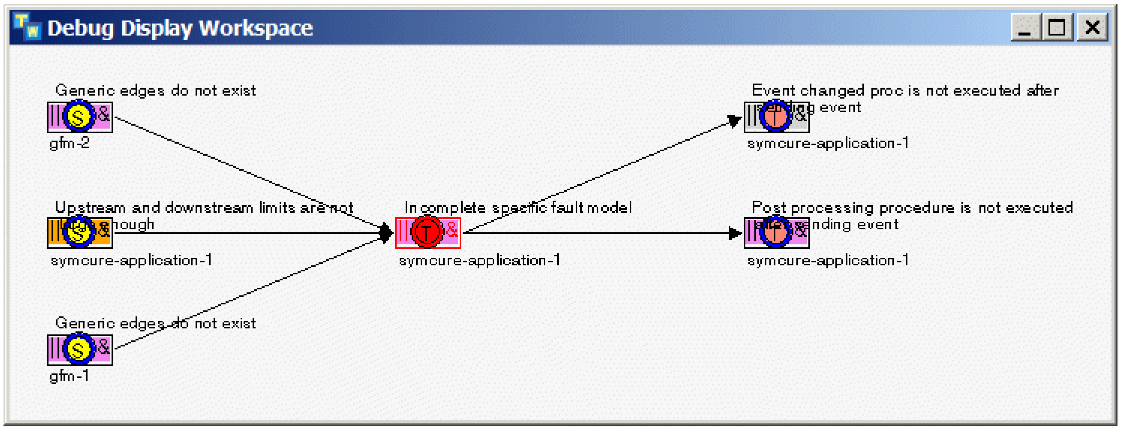
While sequential mode makes navigation through the specific fault model a simple step-by-step procedure, this example demonstrates its only disadvantage. In step 2, it is not immediately clear why “Upstream and downstream limits are not large enough” is suspect and not true, until step 3, which shows that there are alternative explanations (“Generic edges do not exist”) for “Incomplete specific fault model”. This is why each of the causes of “Incomplete specific fault model” is suspect.

Parallel Mode

In parallel mode, the debugger constructs and propagates to all the known causes and effects of an event in parallel, as this example shows:

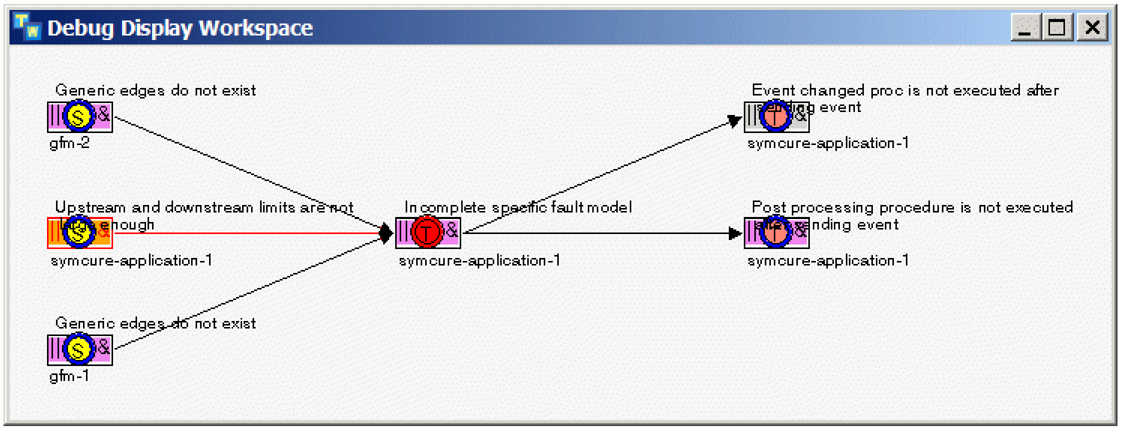
1 Displays the incoming event “Incomplete specific fault model” on symcure-application-1 is specified as true. “Incomplete specific fault model” is also the Current Event.

2 Constructs and propagates all causes and effects of Current Event in parallel. Thus, it is immediately obvious that “Generic edges do not exist”, “Upstream and downstream limits are not large enough”, and “Generic edges do not exist” are all causes of “Incomplete specific fault model”, and you can now understand why each is suspect in that context.


Debug Display Workspace
Generic edges do not exist
gfm-2
Upstream and downstream limits are not large enough
symcure-application-1
Generic edges do not exist
gfm-1
Incomplete specific fault model
symcure-application-1
Event changed proc is not executed after pending event
symcure-application-1
Post processing procedure is not executed after pending event
symcure-application-1

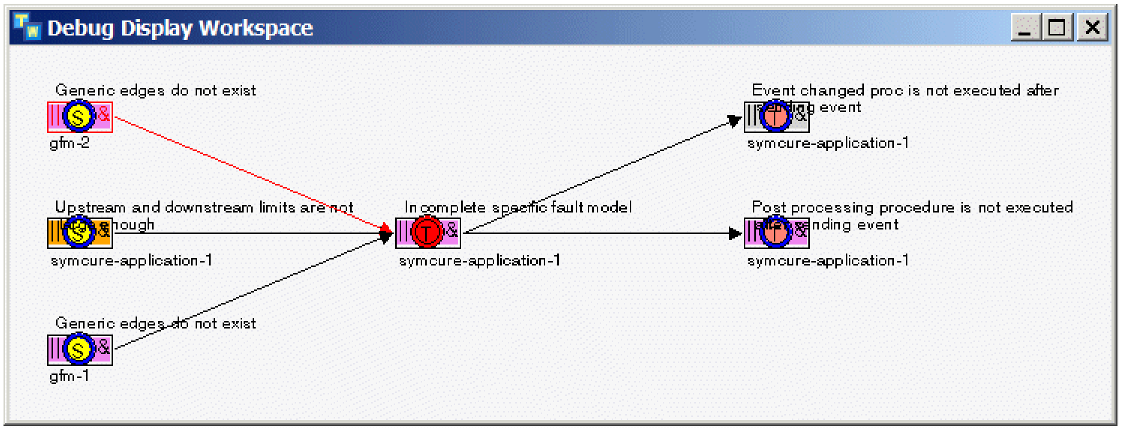
Note that the current event is still “Incomplete specific fault model”. This is crucial to understanding parallel mode. As explained earlier, in parallel mode, the debugger alternates between navigating to a neighboring event and showing the neighbors of the current event, in parallel.

3 Traverses the edge between “Upstream and downstream limits are not large enough” and “Incomplete specific fault model” and makes “Upstream and downstream limits are not large enough” the Current Event, while “Incomplete specific fault model” is the Previous Event and the Incoming Event.


Debug Display Workspace
Generic edges do not exist
gfm-2
Upstream and downstream limits are not large enough
symcure-application-1
Generic edges do not exist
gfm-1
Incomplete specific fault model
symcure-application-1
Event changed proc is not executed after pending event
symcure-application-1
Post processing procedure is not executed after pending event
symcure-application-1

"Upstream and downstream limits are not large enough" has no causes or additional effects. Thus, there is no expansion of this event. Consequently, step 4 traverses the edge between "Generic edges do not exist" on gfm-2 and "Incomplete specific fault model" designating "Generic edges do not exist" the Current Event, while "Incomplete specific fault model" remains the Previous Event and the Incoming Event.

4 "Generic edges do not exist" on gfm-2 has no causes or additional effects. Thus, there is no expansion of this event. Consequently, step 5 traverses the edge between "Generic edges do not exist" on gfm-1 and "Incomplete specific fault model" designating "Generic edges do not exist" on gfm-1 the Current Event, while "Incomplete specific fault model" remains the Previous Event and the Incoming Event.


Debug Display Workspace
Generic edges do not exist
gfm-2
Upstream and downstream limits are not large enough
symcure-application-1
Generic edges do not exist
gfm-1
Incomplete specific fault model
symcure-application-1
Event changed proc is not executed after pending event
symcure-application-1
Post processing procedure is not executed after pending event
symcure-application-1

Note that parallel mode requires a number of extra steps to get to "Generic edges do not exit" on gfm-1 as the Current Event. This is because of the extra EXPAND-EVENT step for each event. While parallel mode overcomes the disadvantage of sequential mode by displaying all causes and effects of an event in a single step, it requires additional steps for expanding events to complete the debugging process and it is not as simple to navigate through the model as in sequential mode.

Notes

Keep in mind these points while using the SymCure debugger:

- Do not invoke the debug panel in administrator mode, since it is not possible to control the user menu choices on the event display objects, some of which are not relevant for debug display, for example, Set event to true.
- You will find the following differences between specific fault model displays that you access by choosing Project > Logic > Diagnose > Specific Fault Models and the debug displays:
  - Each specific fault model display shows a set of correlated events managed by a single diagnosis manager. The debugger does not concern itself with different diagnoses managers. Event propagation can be understood independently of diagnosis managers. The debug display workspace shows all events even when they are not correlated, that is, even when they are associated with different diagnosis managers.
  - When an event receives a value by mutual exclusion, the debug display workspace draws an undirected edge between the event and its mutually exclusive counterpart. Such edges do not represent causality; they are shown for debugging purposes only and are not displayed in the specific fault model displays.
  - The debug display workspace is intended to trace event propagation, which occurs as a result of causal interactions represented by edges in the graph. In rare situations, SymCure creates edges between events when there is no propagation across these edges. The debug display omits such edges, but the specific fault model display shows them.
  - The debug display layout is different because the debug display builds the model one event at a time. Vertical spacing between events differs if the events are not correlated at the time they are first displayed. This is to ensure that events for different diagnosis managers do not try to occupy the same “real estate” on the debug display workspace.
  - You can manually move events on the debug display workspace. The boundaries of the workspace automatically shrink to fit the events. The debug display layout algorithm uses the boundaries of the workspace to position events that are not known to be correlated to existing events on the workspace.

Root Cause Episode Management

Describes run-time root cause episode management tools for SymCure applications.

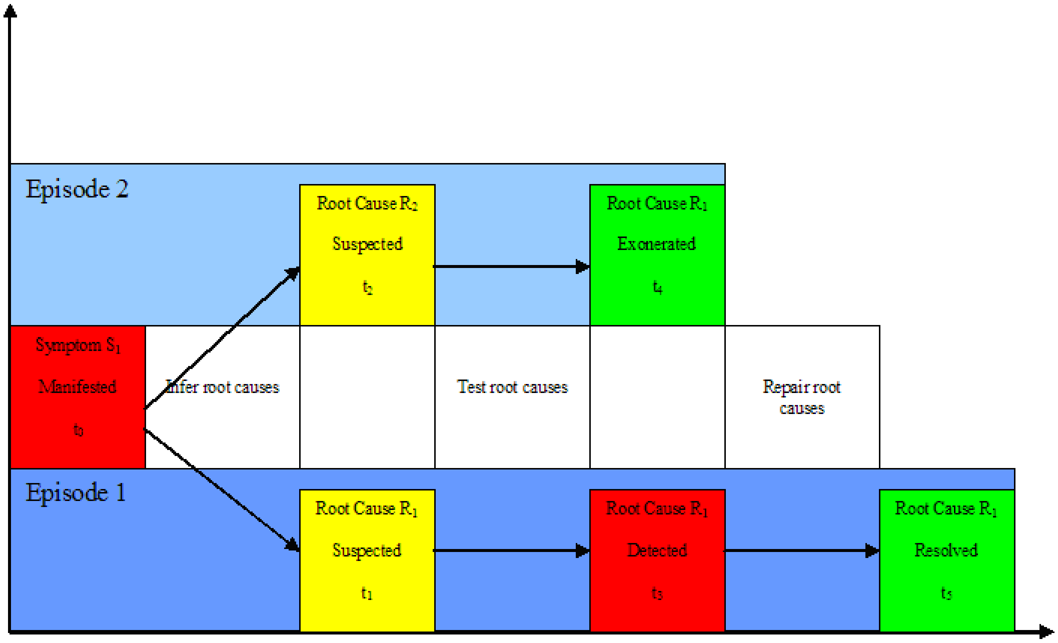
Introduction

SymCure identifies root causes for problems on managed domain objects by correlating observed symptoms and test results. It then initiates repair actions to fix the root causes.

An intelligent fault management system consciously considers the existence of a fault following the manifestation of one or more of its symptoms. Fault management allocates various resources towards the isolation and eradication of the faults suspected to be the root causes of the manifested symptoms: diagnostic inference identifies suspect faults; testing may be used to exonerate or implicate suspected faults; repair activity is directed towards resolving implicated failures. A root cause event thus traces a path of values in the fault management cycle: initially it is implicitly assumed to be false, i.e., it does not exist. Following the manifestation of its symptoms it is treated as suspect. Generally, after testing or the arrival of additional information, it is either considered false (exonerated) or true (implicated). An implicated root cause event is set to false when it is repaired.

A root cause episode captures the trajectory of values traced by a root cause event from its manifestation to its eradication. Each symptom leads the diagnostic engine to identify one or more possible root cause faults. On further testing, a fault may be exonerated, which completes an episode, or it may be detected to be true, requiring suitable repair actions. On completion of the repair actions, the fault is resolved, thereby completing the fault's episode. Thus, a root cause episode comprises a collection of event states that describe the evolution of the fault from a symptomatic manifestation to its exoneration or repair leading to the eradication of the fault.

The following figure depicts the time line for two examples Episode 1 and Episode 2 associated with root cause events R_1 and R_2. The squares in red, yellow, and green represent events (red implies the event is "true", yellow implies "suspect", and green is "false"). The arrows between squares represent fault management activities (e.g., infer, test, and repair root causes).


Episode 2
Root Cause R_2 Suspected t_2
Root Cause R_1 Exonerated t_4
Symptom S_1 Manifested t_0
Infer root causes
Test root causes
Repair root causes
Episode 1
Root Cause R_1 Suspected t_1
Root Cause R_1 Detected t_3
Root Cause R_1 Resolved t_5

1 Symptom S_1 is manifested at time t_0.

2 At t_1, root cause R_1 is suspected. This sets up Episode 1, whose starting point is assumed to be t_0. Tests may be run to implicate or exonerate the suspected root cause.

3 At t_2, root cause R_2 is suspected. This sets up Episode 2, whose starting point is assumed to be t_0. Tests may be run to implicate or exonerate the suspected root cause.

4 At t_3, the tests for R_1 are concluded and R_1 is detected to be true. Any repair action associated with R_1 is run at this time.

5 At t_4, the tests for R_2 are concluded and R_2 is exonerated, and Episode 2 spanning from t_0 to t_4 comes to an end. Note that fault represented by R_2 must actually have been false for the entire duration t_0 to t_4.

6 At t_5, root cause R_1 is resolved, and Episode 1 spanning from t_0 to t_5 comes to an end. Note that fault represented by R_1 must actually have been true for the entire duration t_0 to t_5.

Motivation

Root cause episodes need to be explicitly represented for the following reasons:

- Diagnostic introspection.
- Effective repair and maintenance.
- Episodes are a critical requirement for service management.

Diagnostic Introspection

An intelligent fault management system should be equipped with some capability to monitor and report on its own effectiveness. A root cause episode allows SymCure to answer the following questions about its own performance:

- How quickly did SymCure identify the suspected root causes for a problem?
- How soon after the manifestation of the problem did SymCure detect its root causes?
- How quickly did SymCure help the managed system to recover from the problem?

Furthermore, diagnostic introspection opens up the potential to report other relevant details about fault management, such as:

- What is the cost of testing the problem?
- What is the cost of recovery?
- How long was the domain object offline? How much did that cost?

Effective Repair and Maintenance

Keeping a history of root cause episodes over a period of time facilitates intelligent proactive repair and maintenance policies that may enhance the lifetime of domain objects, facilitate better handling of equipment deterioration, and reduce system downtime, as demonstrated by the following examples.

Repair actions are intended to eradicate a root cause. Usually, they are dependent on the history of the root cause. For example, if your computer freezes because of a bad sector on its hard drive, a simple repair action may be to reboot the computer. However, if problem is occurring with increasing frequency, the appropriate repair action might be to replace the deteriorating hard drive.

The frequency of the occurrence of a root cause event may also point to the need for accelerated maintenance actions. For example, you might schedule disk defragmentation on your computer on a frequent basis once you start experiencing repeated instances of hard drive problems.

Episodes are a Critical Requirement for Service Management

Service management is a promising domain for fault management applications that brings together the need for storing individual root cause episodes as well as their history. Service contracts typically mandate specific targets for the delivery of a capability. Consequently, fault management in this domain often requires considering the aggregate of all root cause episodes over a period of time.

For example, a Service Level Agreement (SLA) between a provider and a consumer may stipulate that the aggregate downtime within a specified time period should not exceed a certain time duration. While no individual root cause episode that results in a service outage may exceed this duration, the aggregation of all such episodes over the specified time period may trigger an SLA violation. Service outages may be more or less serious depending upon their durations, time of occurrence, and past history, and may require escalated priority depending on these factors. Knowledge of the frequency and the details of each episode may be utilized to identify escalating problems, thus supporting intelligent fault management that dynamically targets its resources at the most critical problems.

SymCure Root Cause Episode Management

SymCure incorporates out-of-the-box root cause episode management to archive, retrieve, chart, and delete root cause episodes associated with a domain object.

You must enable root cause episode management for individual domain objects, because it may not be desirable for all domain objects. Once root cause episode management is enabled, you can view root cause episodes for that object.

SymCure provides a set of APIs that provide access to information contained in episode archives, as well as for charting root cause episodes. For details, see Root Cause Episode Management and Charting.

Definitions

SymCure's root cause episode management capabilities uses these concepts:

- *Root Cause Episode* — A collection of event states that describe the evolution of the fault from a symptomatic manifestation to its exoneration or repair leading to the eradication of the fault.
- *Root Cause Episode Archive* — A collection of episodes that describe the history of a fault.
- *Root Cause Episode Archive Manager* — The manager of all root cause episode archives associated with a domain object.

Enabling Root Cause Episode Management

You must manually enable root cause episode management for individual domain objects in a process map.

To enable root cause episode management:

➔ Choose Enable Root Cause Episode Management on a domain object.

Displaying the Root Cause Episode Manager

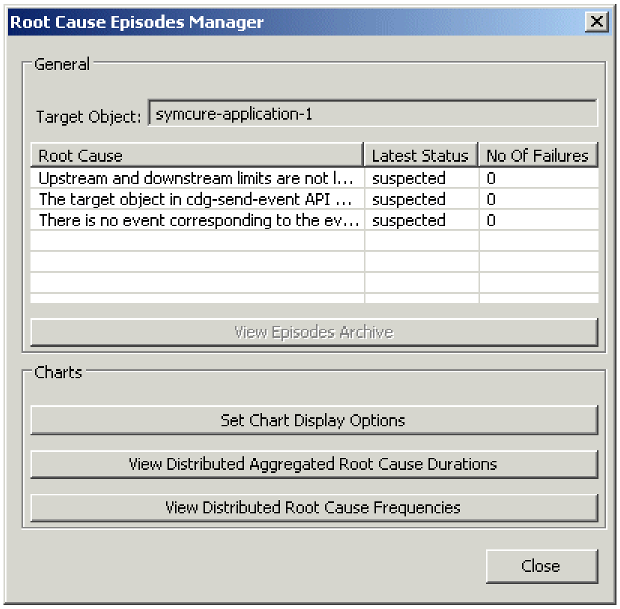
A root cause episode manager is associated with a domain object. It manages all root cause episode archives for the domain object.

By default, the archive manager keeps episodes for 1 day after it is completed. You can configure the amount of time in the `config.txt` file. For details, see Root Cause Episode Management.

To display the root cause episode manager:

➔ Choose Show Root Cause Episodes on a domain object for which root cause episode management is enabled.

Here is the root cause episode manager for the symcure-application-1 domain object when the “Specific fault model is not built” event is true:



Root Cause Episodes Manager
General
Charts
View Episodes Archive
Set Chart Display Options
View Distributed Aggregated Root Cause Durations
View Distributed Root Cause Frequencies
Close

Target Object: symcure-application-1

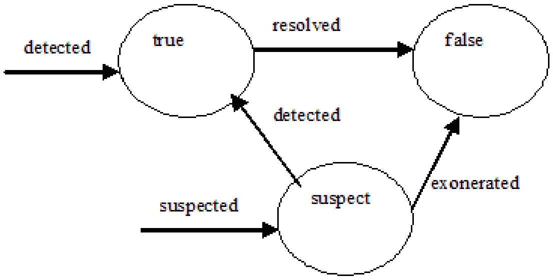
Root Cause	Latest Status	No Of Failures
Upstream and downstream limits are not l...	suspected	0
The target object in cdg-send-event API ...	suspected	0
There is no event corresponding to the ev...	suspected	0

Displaying Root Cause Episodes

A root cause episode comprises a collection of event states that describe the evolution of the fault from a symptomatic manifestation to its exoneration or repair leading to the eradication of the fault.

A root cause episode archive represents the episodic history of each root cause event. SymCure maintains a separate archive for each root cause. You can query an archive to provide information about stored episodes over specified time periods.

The following figure shows how the event value and episode status are related. The event value "suspect" is a consequence of the transition "suspected", "false" a consequence of "exonerated" (as a result of a test) or "resolved" (after running a repair action), and "true" a consequence of "detected".



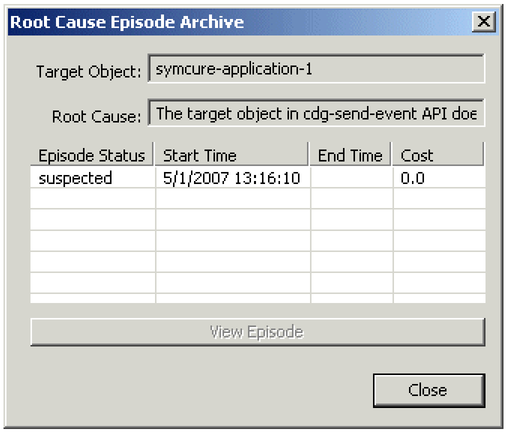
A root cause episode has the following properties:

- Target Object — The name of the domain object that is the target of the root cause event.
- Root Cause — The name of the root cause event.
- Event Value — The last known value of the root cause event, which is "suspect", "false", or "true". The root cause must be "true" or "suspect" to initiate the recording of a root cause episode. When the root cause becomes "false", the episode is considered to have ended. If, subsequently, the root cause becomes "suspect" or "true" again, a new episode is recorded.
- Episode Status — The latest transition of the episode, which is one of these text values: "exonerated", "resolved", "suspected", or "detected".
- Cost — The cost of detecting and, if required, repairing the root cause. By default, this is a summation of the costs of each external action associated directly with the root cause or indirectly through causal links, for example, an action associated with a causally downstream event that is invoked during the episode.
- Manifested At — The time at which the first symptom associated with the fault is manifested. SymCure treats this as the start time for the episode.
- Suspected At — The time at with SymCure infers that the root cause is suspect.
- Time Detected — The time that the root cause is inferred or observed to be true, usually as a consequence of test actions.
- Time Exonerated — The time that a suspected root cause is inferred or observed to be false, usually as a consequence of test actions.
- Time Resolved — The time that a root cause event transitions from true to false, usually as a consequence of a repair action.

To display a root cause episode:

1 In the Root Cause Episode Manager dialog, select a root cause, then click the View Episode Archive button.

Here a root cause episode archive for the "The target object in cdg-send-event API does not exist" root cause for symcure-application-1 when the root cause has been resolved:


Root Cause Episode Archive

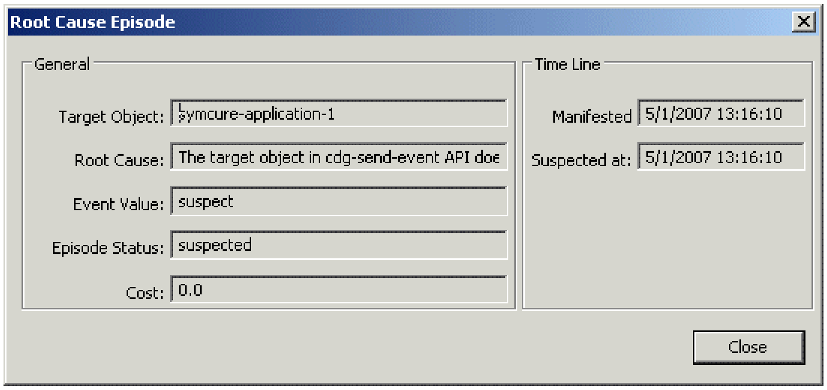
Target Object: symcure-application-1

Root Cause: The target object in cdg-send-event API doe

Episode Status	Start Time	End Time	Cost
suspected	5/1/2007 13:16:10		0.0

2 Select an episode, then click the View Episode button.

Here is the root cause episode, which shows the event value, episode status, and the time at which the root cause was manifested, suspected, detected, and resolved:


Root Cause Episode

General

Target Object: symcure-application-1

Root Cause: The target object in cdg-send-event API doe

Event Value: suspect

Episode Status: suspected

Cost: 0.0

Time Line

Manifested 5/1/2007 13:16:10

Suspected at: 5/1/2007 13:16:10

Charting Root Cause Duration and Frequency Distributions

You can display aggregated durations distributions and frequency distributions for root cause episodes associated with a target object in charts.

The aggregate durations chart plots the distributions of the aggregate (total) durations for each root cause event for a target object over a specified period of time. For example, you can show the total time for which a root cause event is manifested per hour for a period of 24 hours, or per day for period of a month, or per week for a period of year.

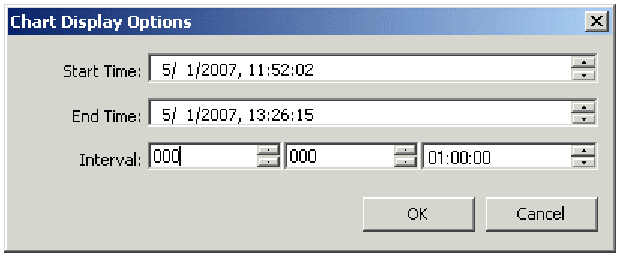
The frequency chart plots the distributions of the number of occurrences of each root cause event for a target object over a specified time period.

You can specify the distribution interval and the overall time period for any chart. By default, each chart plots the durations/number of occurrences every hour for the past 24 hours, going backwards from the current time.

To chart root cause durations and frequency distributions:

1 Display the Root Cause Episode Manager dialog.

2 Click the Set Chart Display Options button and configure the start time and end time for the chart and the length of the interval:



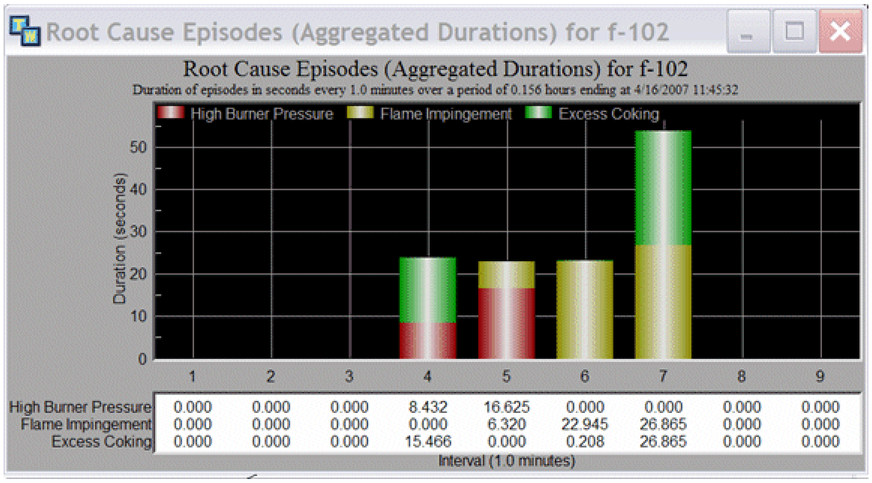
Start Time: 5/ 1/2007, 11:52:02

End Time: 5/ 1/2007, 13:26:15

Interval: 000 000 01:00:00

3 Click the View Distributed Aggregated Root Cause Durations button to chart root cause durations.

For example, here is the Aggregated Durations chart for the F-102 demo:



Root Cause Episodes (Aggregated Durations) for f-102

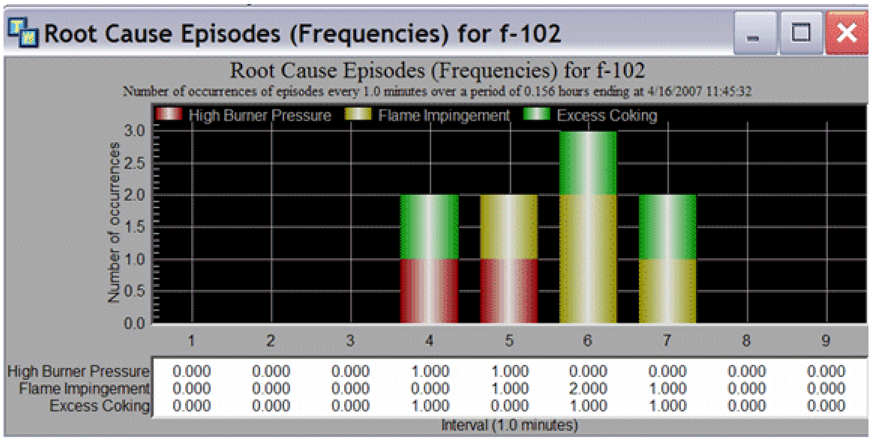
Duration of episodes in seconds every 1.0 minutes over a period of 0.156 hours ending at 4/16/2007 11:45:32

	1	2	3	4	5	6	7	8	9
High Burner Pressure	0.000	0.000	0.000	8.432	16.625	0.000	0.000	0.000	0.000
Flame Impingement	0.000	0.000	0.000	0.000	6.320	22.945	26.865	0.000	0.000
Excess Coking	0.000	0.000	0.000	15.466	0.000	0.208	26.865	0.000	0.000

Interval (1.0 minutes)

4 Click the View Distributed Root Cause Frequencies button to chart root cause frequencies.

For example, here is the Frequencies chart for the F-102 demo:



Root Cause Episodes (Frequencies) for f-102

Number of occurrences of episodes every 1.0 minutes over a period of 0.156 hours ending at 4/16/2007 11:45:32

	1	2	3	4	5	6	7	8	9
High Burner Pressure	0.000	0.000	0.000	1.000	1.000	0.000	0.000	0.000	0.000
Flame Impingement	0.000	0.000	0.000	0.000	1.000	2.000	1.000	0.000	0.000
Excess Coking	0.000	0.000	0.000	1.000	0.000	1.000	1.000	0.000	0.000

Interval (1.0 minutes)

Saving Root Cause Episodes

To save a root cause episode to an XML file:

➔ Choose Save Root Cause Episodes on a domain object for which root cause episode management is enabled.

The XML file is saved to the `archives` directory of the application root directory, by default.

Configuring SymCure Applications

Describes the parameter initializations that you can configure at startup to customize various aspects of SymCure's default browser.

Introduction

SymCure supports a number of user-defined parameter initializations, which you to define initial values for various SymCure features and customize its behaviors. SymCure also provides a number of user-defined procedures that can execute under various conditions.

You configure these parameters in an external file named `config.txt`, which must be located in the same directory as your SymCure application. The configuration file is loaded at startup when the SymCure application is loaded and whenever you restart G2.

Here are sample configuration parameters with several parameters specified:

```
cdg-terminate-diagnosis-early=false
cdg-upstream-limit=500
cdg-downstream-limit=500
cdg-compute-priority-procedure=my-compute-priority-proc
cdg-default-target-priority=5
cdg-audit-incoming-event-procedure=my-audit-incoming-event-proc
```

You can configure parameters to control:

- The size of the specific fault model that SymCure builds during incremental builds of the specific fault model.
- How long SymCure keeps diagnostic information before deleting it.
- Event unchanged procedures of generic events.
- How SymCure handles event priority.
- How SymCure schedules specific actions.
- The display of specific fault models.
- The ability to display dynamically updating specific fault models.
- User-defined methods that SymCure runs for auditing purposes.
- The logging of events for debugging purposes.

Loading Fault Model Configuration Parameters

You can load fault model configuration parameters from SymCure without restarting the application. For example, while viewing a specific fault model, you might want to change the default horizontal and vertical spacing between events in the `config.txt` file. Once you modify the file, click the Load Fault Model Configuration Parameters button on the Fault Modeling toolbar (⚙).

Note This toolbar button only loads the parameters in the CDG and CDGUI groups.

Specific Fault Model Creation

SymCure's diagnostic algorithm builds specific fault models incrementally, which allows it to respond to new information as rapidly as possible. During each incremental build, the algorithm limits the number of upstream and downstream events it creates. You can control these limits by configuring upstream and downstream limit parameters and the amount of time between incremental builds. Note that these limits apply to the number of specific events built at each incremental stage, not to the total number of specific events in the specific event model.

You can also limit the size of the specific fault model by terminating the diagnosis early.

cdg-upstream-limit=1000

An integer that determines the maximum number of specific events that can be built upstream of a specific event during any incremental stage of building the specific fault model. On subsequent diagnostic processing, SymCure might build additional events upstream. This limit ensures that any observed event that arrives subsequently will be processed in a finite amount of time. Additional events can be built following the arrival of new information or by SymCure's periodic incremental processing.

cdg-downstream-limit=1000

An integer that determines the maximum number of specific events that can be built downstream of a specific event during any incremental stage of building the specific fault model. On subsequent diagnostic processing, SymCure might build additional events downstream. This limit ensures that any observed event that arrives subsequently will be processed in a finite amount of time. Additional events can be built following the arrival of new information or by SymCure's periodic incremental processing. As a result, the number of upstream events built during any pass can exceed this limit.

cdg-incremental-diagnosis-monitor-interval=120

An integer that determines the number of seconds SymCure waits before it makes an incremental passes to construct a specific fault model. SymCure builds large specific fault models incrementally within the upstream and downstream limits described above. Once the specific model has been built to its limits, SymCure waits for additional information, in the form of new symptoms and test results, before initiating further construction of the specific fault model. However, when new information is not forthcoming, SymCure initiates periodic, incremental construction of any specific fault model that is not complete. During each incremental build, SymCure observes the limits on the number of events built upstream and downstream.

To use SymCure's debugging feature, you must set this parameter to a negative number. For details, see Debugging SymCure Fault Models.

cdg-terminate-diagnosis-early=false

A truth-value that determines whether diagnosis can complete before the complete specific fault model has been built. When set to true, diagnosis is said to be complete when all observed events are fully explained by the set of known root causes, even if the entire specific fault model has not been built. When set to false, the diagnosis is said to be complete when all observed events can be explained and the entire specific fault model has been built.

cdg-allow-unspecified-event-to-be-root-cause=false

A truth-value that determines whether unspecified specific events with no other causes are considered root causes. By default, this parameter is false, which means unspecified specific events with no other causes are not treated as root causes. This means they do not appear in the Root Causes Browser unless the Event Type of the corresponding generic event is explicitly configured as a root cause. Set this parameter to true to cause all specific events with no other causes to be considered root causes, which means these events automatically appear in the Root Causes Browser.

Diagnosis Timing

You can configure the amount of time the specific fault model and the diagnosis manager exist after all the root causes associated with a set of correlated events become false.

cdg-diagnosis-deletion-interval=300

An integer that determines the number of seconds SymCure waits before deleting a diagnosis manager that has not been updated. SymCure deletes every diagnosis manager that has no true and suspect incoming events, no specific action that is currently running for the diagnosis manager, and that has not been updated within the diagnosis deletion interval. SymCure also deletes all specific events and specific actions that the diagnosis manager manages.

cdg-diagnosis-deletion-monitor-interval=300

An integer that determines the number of seconds that SymCure waits before checking whether diagnosis managers should be deleted. SymCure periodically checks all diagnosis managers to see if any of them should be deleted.

To use SymCure's debugging feature, you must set this parameter to a negative number. For details, see Debugging SymCure Fault Models.

Event Unchanged Procedure

You can configure generic events with two user-defined procedures that can execute at run time under different conditions: the event changed procedure and the event unchanged procedure. You can configure various parameters related to the event unchanged procedure.

cdg-unchanged-events-monitor-name=cdg-unchanged-events-monitor

A symbol that specifies the name of a user-defined procedure that SymCure calls at periodic intervals to determine when to invoke the event unchanged procedure of an event. Using this procedure, you can access the unchanged events and provide customized logic to invoke the unchanged event procedure associated with each unchanged event.

The default unchanged events monitor is cdg-unchanged-events-monitor, which identifies each specific event that has an event unchanged procedure. If the value of the specific event is not filtered by the cdg-unchanged-events-filter, then the default procedure calls the event unchanged procedure for the specific event. For the signature of this procedure, see the description of the event unchanged procedure in Configuring User-Defined Procedures for Generic Events.

If you use the default unchanged events monitor procedure, you can customize it by configuring cdg-unchanged-events-filter and cdg-unchanged-events-monitor-interval, described next.

cdg-unchanged-events-filter=false suspect unknown

A text array that specifies the values for unchanged events that the application does not need to monitor. You use this parameter in conjunction with the default unchanged events monitor procedure, cdg-unchanged-events-monitor, to filter unchanged events. By default, the default unchanged events monitor does not monitor false, suspect, and unknown events.

cdg-unchanged-events-monitor-interval=21600

An integer that specifies the time interval that the default unchanged events monitor procedure, cdg-unchanged-events-monitor, waits for before looking for unchanged events.

Priority

You can configure the priority of generic events to be an integer that represents the level of importance of the event, the likelihood that the event has occurred, or any other numerical measure. SymCure uses the priority to prioritize suspected root causes.

You can configure the default priority of all generic events. You can also configure a procedure to compute the priority of a specific event, based on the generic event priority and the target domain object.

cdg-default-target-priority=1

An integer that defines the default priority value for every target object. SymCure defines a method for grtl-domain-object called cdg-get-priority, which returns the value of this parameter for any grtl-domain-object. You can create your own cdg-get-priority method to associate different default priorities for user-defined domain object classes.

cdg-compute-priority-procedure=unspecified

A symbol that specifies a user-defined procedure that determines how SymCure computes priority for specific events with the same target class.

You can create the procedure from the palettes by choosing View > Toolbox - Fault Models > User-Defined Procedures and Methods and creating a Compute Specific Event Priority Procedure. The procedure's signature is automatically populated from its signature attribute.

The signature of the procedure is:

my-compute-specific-event-priority-procedure
(*Target*: class grtl-domain-object, *GenericEventPriority*: integer,
SpecificEvent: class cdg-specific-event)
-> *priority*: float

where:

Target is the target class for the generic event.

GenericEventPriority is the value of the priority attribute of the generic event.

SpecificEvent is the specific event whose priority you want to compute.

The return value of *priority* is a measure of the priority or probability of the specific event.

The cdg-sort-events-by-priority API procedure uses this parameter to sort events by priority. The default value is unspecified, which means SymCure uses its own default logic for determining the priority of events.

Specific Action Scheduling

cdg-user-defined-scheduling-procedure=unspecified

A symbol that specifies a user-defined procedure that determines how SymCure schedules specific actions, based on the Cost and Reliability of the specific action. For more information, see Customizing the Scheduling of External Actions.

You can create the procedure from the palettes by choosing View > Toolbox - Fault Models > User-Defined Procedures and Methods and creating a User-Defined Scheduling Procedure. The procedure's signature is automatically populated from its signature attribute.

Specific Fault Model Display

You can configure parameters to control the distance between specific events in a specific event model.

cdg-horizontal-distance=300

An integer that defines the horizontal distance, in pixels, between the columns of specific event display objects in the display of a specific fault model.

cdg-vertical-distance=100

An integer that specifies the vertical distance, in pixels, between the rows of specific event display objects in the display of a specific fault model.

Debugging

You can configure SymCure to allow the creation of dynamically updating specific event models for a specified set of domain objects. You use this feature for debugging generic fault models.

For details on how to use this feature, see Debugging SymCure Fault Models.

cdg-display-animated-specific-fault-model=false

Whether to allow initialization of dynamically updating specific fault models. You must set this parameter to true to enable this feature. **Note:** This parameter is deprecated.

cdg-enable-debugging=false

Enables the logging of specific events that are used for debugging. For more information, see Debugging SymCure Fault Models.

Chattering Events

You can configure the default criteria for chattering events. For details, see Detecting Chattering Events.

cdg-enable-check-for-chattering-events=true

Enables/disables chattering detection. By default, it is true.

cdg-lookback-for-chattering=30

Controls how far back in seconds SymCure examines the historical values of each event to determine if it is chattering. By default, this is 30 seconds. For the sake of run-time efficiency, the maximum permissible value for lookback is 300 seconds; any value larger than this is ignored.

cdg-max-chattering-repetitions=10

Determines the number of repetitions of an event's value that results in chattering behavior. By default, if the event gets the same value 10 times within the lookback period, the event is chattering. The minimum permissible repetitions value is 5; any lower value is ignored. This safeguard prevents SymCure from going offline prematurely.

Root Cause Episode Management

cdg-episode-deletion-monitor-interval=86400

The amount of time a root cause episode should remain in the application after it is completed. Episodes completed prior to the persistence interval are automatically deleted by a “garbage collection” background process.

cdg-lookback-for-charting-root-cause-episodes-distributions=86400

The difference between end time and start time for charting root cause episodes. The default value is 24 hours.

cdg-interval-for-charting-root-cause-episodes-distributions=3600

The interval for charting root cause episodes. The default is 1 hour.

cdg-root-cause-episodes-archiving-directory=$application-root-directory/archives

The default directory where the XML representations of root cause episodes are stored.

User-Defined Methods

SymCure provides "hooks" at various points at run time for executing user-defined methods, beginning with the reception of incoming events and the creation of diagnosis managers through diagnosis completion. You configure these user-defined methods, using the parameters described in this section.

You can use these methods to:

- Send messages to operators.
- Log all event correlation and diagnosis information to provide an audit trail of a diagnosis, a diagnosis manager is deleted.
- Send this information to an external database.
- Construct a "trouble ticket" or other sets of information, and export it to an external system after diagnosis completes.
- Send information on predicted events to an external system for proactive service and maintenance.

SymCure passes the appropriate correlation manager or event as arguments of these user-defined methods. You can use SymCure API to access the information passed into the user-defined methods. For a description of the SymCure API, see Application Programmer's Interface.

Some methods execute in a separate thread, while others execute in the same thread.

You can create the procedures from the palettes by choosing View > Toolbox - Fault Models > User-Defined Procedures and Methods and creating one of the audit procedures on the palette. The procedure's signature is automatically populated from its signature attribute.

cdg-audit-incoming-event-procedure=unspecified

A symbol that specifies a user-defined procedure that audits incoming events. The signature of the audit method is:

```
my-audit-incoming-event-proc
    (SpecificEvent: class cdg-specific-event)
```

The procedure is invoked in a separate thread on the acceptance of an incoming *SpecificEvent*. To process incoming events within the same thread, use the cdg-send-event-with-post-processing API procedure, described in Sending Events.

cdg-audit-root-cause-procedure=unspecified

A symbol that specifies a user-defined procedure that audits root cause events. The signature of the audit method is:

```
my-audit-root-cause-proc
   (SpecificEvent: class cdg-specific-event)
```

The procedure is invoked in a separate thread whenever *SpecificEvent* is identified to be a root cause for one or more symptoms.

cdg-audit-alarm-procedure=unspecified

A symbol that specifies a user-defined procedure that audits alarm events. The signature of the audit method is:

```
my-audit-alarm-proc
   (SpecificEvent: class cdg-specific-event)
```

The procedure is invoked in a separate thread whenever there is a change in the event-value of *SpecificEvent* with event-type equal to alarm.

cdg-audit-diagnosis-before-deletion-procedure=unspecified

A symbol that specifies a user-defined procedure that audits diagnosis managers before they are deleted. The signature of the audit method is:

```
my-audit-diagnosis-before-deletion-proc
   (DiagnosisManager: class cdg-diagnosis-manager)
```

This procedure is invoked in the same thread just before *DiagnosisManager* is deleted.

cdg-audit-diagnosis-status-procedure=unspecified

A symbol that specifies a user-defined procedure that audits the status of a diagnosis manager. The signature of the audit method is:

```
my-audit-diagnosis-status-proc
   (DiagnosisManager: class cdg-diagnosis-manager,
   OldStatus: symbol, NewStatus: symbol)
```

The procedure is invoked in a separate thread whenever SymCure completes its processing of an incoming event. If the diagnosis-status of *DiagnosisManager* is unchanged, *OldStatus* and *NewStatus* are identical; otherwise, they are different.

cdg-audit-diagnosis-after-merger-procedure=unspecified

A symbol that specifies a user-defined procedure that audits diagnosis managers following the merger of two diagnosis managers. The signature of the audit method is:

my-diagnosis-after-merger-audit-procedure
(*DiagnosisManager*: class cdg-diagnosis-manager)

DiagnosisManager is the surviving diagnosis manager.

Default Browsers

You can configure the default browsers that SymCure uses for displaying alarms, root causes, test actions, and repair actions. For example, you might want to combine events and external actions in a single browser.

resources-subdirectory=resources/symcure

The directory in which SymCure GFR text resources are stored.

cdg-alarm-message-queue=alarms

The default queue to use for alarm events, which is the Alarms Browser, by default.

cdg-root-causes-message-queue=root causes

The default queue to use for roo cause events, which is the Root Causes Browser, by default.

cdg-test-actions-message-queue=test actions

The default queue to use for test actions, which is the Test Actions Browser, by default.

cdg-repair-actions-message-queue=repair actions

The default queue to use for repair actions, which is the Repair Actions Browser, by default.

Archiving

You can configure the default directory for exporting generic and specific fault model folders, and whether to archive folders automatically upon compilation.

cdg-archive-generic-fault-models-on-compilation=true

Whether to export generic fault model folders automatically upon compilation.

cdg-generic-fault-model-archiving-directory=$application-root-directory/archives

The default directory for exporting generic fault model folders to XML.

cdg-specific-fault-model-archiving-directory=$application-root-directory/archives

The default directory for exporting specific fault model folders to XML.

Application Programmer's Interface

Describes the SymCure application programmer's interface (API).

Introduction

All interactions with SymCure take place through a small set of procedures, methods, classes, and attributes. These items are referred to as the Application Programmer's Interface (API).

This chapter describes the API procedures and methods for SymCure, which you get by writing your own G2 procedures that call these API procedures and methods. It also describes the SymCure classes that you can subclass to create custom implementations of generic and specific events and actions, and their associated displays.

Sending Events

An event is a cdg-specific-event that is identified by its event-name and target-object, where:

- event-name is a text string that identifies the event.
- target-object is a text string that identifies an instance of grtl-domain-object.

To send an event, you specify its event-value, which can be "true", "false", "suspect", or "unknown". Sending an event:

- Propagates event-value and event-status to all upstream and downstream events.
- Diagnoses root causes by determining event-value and event-status of all potential root causes for the event.
- Predicts event-value and event-status for all downstream events.

You can also simply diagnose an event, predict an event, or update an event.

The API provides different versions of the procedures for sending, diagnosing, or predicting an event, which include the following arguments:

- Target object, event name, and value only.
- Target object, event name, value, and sender.
- Target object, event name, value, sender and window.

The sender is an object that is responsible for sending the event. The sender is not required for diagnostic reasoning, but it can be used for reporting purposes.

The window is a g2-window on which to send the event. If the window is not specified, the procedure uses the gfr-default-window.

The API also provides different versions of each procedure for sending, diagnosing, and predicting an event with a post-processing procedure. The post-processing procedure executes in the same thread as the event propagation and has the following signature:

```
my-post-processing-proc
    (TargetObject: class grtl-domain-object,
    SpecificEvent: class cdg-specific-event,
    DiagnosisManager: class cdg-diagnosis-manager,
    Window: class g2-window)
```

Sending Events

cdg-send-event

(*TargetObject*: class grtl-domain-object, *EventName*: text, *EventValue*: text, *Sender*: class grtl-domain-object, *Window*: class g2-window)

Sends a value *EventValue* for the event defined by *EventName* and *TargetObject*, diagnoses its root causes, and predicts its impacts. *Sender* and *Window* are optional arguments, as described in the introduction to this section.

cdg-send-event

(*TargetObject*: class grtl-domain-object, *EventName*: text, *EventValue*: text, *UserDefinedData*: sequence, *Sender*: class object, *Client*: class object)

Same as the previous except allows you to send user-defined data with the event.

cdg-send-event-with-post-processing

(*TargetObject*: class grtl-domain-object, *EventName*: text, *EventValue*: text, *Sender*: class grtl-domain-object, *Window*: class g2-window, *PostProcessingProcedure*: symbol)

Sends a value *EventValue* for the event defined by *EventName* and *TargetObject*, diagnoses its root causes, and predicts its impacts. This procedure also invokes the procedure specified by *PostProcessingProcedure* at the completion of event propagation. *Sender* and *Window* are optional arguments, as described in the introduction to this section.

You can create the procedure from the palettes by choosing View > Toolbox - Fault Modeling > User-Defined Procedures and Methods and creating a Send Event With Post Processing Procedure. The procedure's signature is automatically populated from its signature attribute.

cdg-send-event-with-post-processing

(*TargetObject*: class grtl-domain-object, *EventName*: text, *EventValue*: text, *UserDefinedData*: sequence, *Sender*: class grtl-domain-object, *PostProcessingProcedure*: symbol*, Client*: class object)

Same as the previous except allows you to send user-defined data with the event.

Diagnosing Events

cdg-diagnose-event

(*TargetObject*: class grtl-domain-object, *EventName*: text, *EventValue*: text, *Sender*: class grtl-domain-object, *Window*: class g2-window)

Sends a value *EventValue* for the event defined by *EventName* and *TargetObject* and diagnoses its causes. *Sender* and *Window* are optional arguments, as described in the introduction to this section.

cdg-diagnose-event
(*TargetObject*: class grtl-domain-object, *EventName*: text, *EventValue*: text, *UserDefinedData*: sequence, *Sender*: class grtl-domain-object, *Client*: class object)

Same as the previous except allows you to send user-defined data with the event.

cdg-diagnose-event-with-post-processing
(*TargetObject*: class grtl-domain-object, *EventName*: text, *EventValue*: text, *Sender*: class grtl-domain-object, *Window*: class g2-window, *PostProcessingProcedure*: symbol)

Sends a value *EventValue* for the event defined by *EventName* and *TargetObject*, diagnoses its causes, and predicts its impact. This procedure also invokes the procedure specified by *PostProcessingProcedure* at the completion of event propagation. *Sender* and *Window* are optional arguments, as described in the introduction to this section.

cdg-diagnose-event-with-post-processing
(*TargetObject*: class grtl-domain-object, *EventName*: text, *EventValue*: text, *UserDefinedData*: sequence, *Sender*: class grtl-domain-object, *PostProcessingProcedure*: symbol, *Client*: class object)

Same as the previous except allows you to send user-defined data with the event.

Predicting Events

cdg-predict-event
(*TargetObject*: class grtl-domain-object, *EventName*: text, *EventValue*: text, *Sender*: class grtl-domain-object, *Window*: class g2-window)

Sends a value *EventValue* for the event defined by *EventName* and *TargetObject*, and predicts its impact. *Sender* and *Window* are optional arguments, as described in the introduction to this section.

cdg-predict-event
(*TargetObject*: class grtl-domain-object, *EventName*: text, *EventValue*: text, *UserDefinedData*: sequence, *Sender*: class grtl-domain-object, *Client*: class object)

Same as the previous except allows you to send user-defined data with the event.

cdg-predict-event-with-post-processing
(*TargetObject*: class grtl-domain-object, *EventName*: text, *EventValue*: text, *Sender*: class grtl-domain-object, *Window*: class g2-window, *PostProcessingProcedure*: symbol)

Sends a value *EventValue* for the event defined by *EventName* and *TargetObject*, diagnoses its causes, and predicts its impact. This procedure also invokes the procedure specified by *PostProcessingProcedure* at the completion of event propagation. *Sender* and *Window* are optional arguments, as described in the introduction to this section.

cdg-predict-event-with-post-processing
(*TargetObject*: class grtl-domain-object, *EventName*: text, *EventValue*: text, *UserDefinedData*: sequence, *Sender*: class grtl-domain-object, *PostProcessingProcedure*: symbol, *Client*: class object)

Same as the previous except allows you to send user-defined data with the event.

Updating Events

cdg-update-event-value
(*SpecificEvent*: class cdg-specific-event, *NewValue*: text,
Window: class ui-client-item)

Updates the value of *SpecificEvent* to *NewValue*, where *NewValue* is "true", "false", "suspect", or "unknown". *Window* is an optional argument, as described in the introduction to Chapter 8 "Application Programmers' Interface" in the *SymCure User's Guide*.

cdg-update-event-value
(*SpecificEvent*: class cdg-specific-event, *NewValue*: text, *Sender*: class object, *Window*: class ui-client-item)

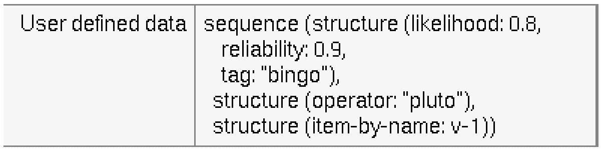
Updates the value of *SpecificEvent* to *NewValue*, where *NewValue* is "true", "false", "suspect", or "unknown". *Sender* and *Window* are optional arguments, as described in the introduction to Chapter 8 "Application Programmers' Interface" in the *SymCure User's Guide*.

Sending User-Defined Data

When sending, diagnosing, or predicting events with user-defined data, the sequence can include any combination of structures, values, and items. SymCure can use structures in this sequence to display them as if they are attributes.

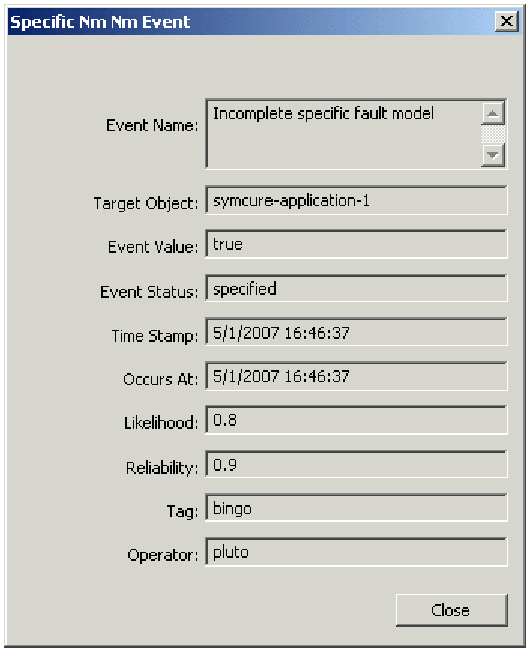
For example, the following procedure creates the specific event with the following user-defined-data attribute:

```
start cdg-send-event
    (symcure-application-1, "Incomplete specific fault model", "true",
    sequence
        (structure(likelihood: 0.8, reliability: 0.9, tag: "bingo"),
        structure(operator: "pluto"),
        structure(item-by-name: symcure-application-1)),
    symcure-application-1, this window)
```



User defined data	sequence (structure (likelihood: 0.8, reliability: 0.9, tag: "bingo"), structure (operator: "pluto"), structure (item-by-name: v-1))

The properties dialog for the specific event display automatically includes all the user-defined attributes:



Event Name: Incomplete specific fault model

Target Object: symcure-application-1

Event Value: true

Event Status: specified

Time Stamp: 5/1/2007 16:46:37

Occurs At: 5/1/2007 16:46:37

Likelihood: 0.8

Reliability: 0.9

Tag: bingo

Operator: pluto

Use the following APIs to access to the values of any structures in the user-defined data:

cdg-get-user-defined-datum
  (*SpecificEvent*: class cdg-specific-event, *AttributeName*: symbol)
  -> *attribute-value*: value

cdg-set-user-defined-datum
  (*SpecificEvent*: class cdg-specific-event, *AttributeName*: symbol,
  AttributeValue: value)

Root Causes

A root cause is a cdg-specific-event that is an underlying cause for a specific event. A root cause cannot itself be caused by any other event. A root cause can either be a:

- Known root cause, in which case its value is either known to be true or known to be false.
- Plausible root cause, in which case its value is true, false or suspect, depending on the value of the corresponding alarm. If the alarm is true, all suspect or true root cause are plausible root causes for explaining the alarm's value. If the alarm is false, all false root causes are plausible explanations. If the alarm is unknown, all unknown root causes are plausible explanations.

You can get the known and plausible root causes of a specific event, and all root causes for a diagnosis manager. You can also get the **effects** of a root cause, which are the specific events that are caused by the root cause.

You can also get the known and plausible explanations of a specific event, using cdg-get-root-causes and cdg-get-plausible-root-causes-of-event, respectively. The explanation of a specific event is the path from known or plausible root causes to an alarm. For a comparison of known and plausible root causes, and known and plausible explanations, see Getting Explanations and Evidence for Specific Events.

cdg-get-root-causes-of-event
(*SpecificEvent*: class cdg-specific-event)
-> *RootCauses*: sequence

Returns a sequence of cdg-specific-event objects, which are known root causes of *SpecificEvent*.

cdg-get-plausible-root-causes-of-event
(*SpecificEvent*: class cdg-specific-event)
-> *PlausibleRootCauses*: sequence

Returns a sequence of cdg-specific-event objects, which are suspected root causes of *SpecificEvent*. If *SpecificEvent* is true, it returns the root causes that are known to be true or are suspect. If *SpecificEvent* is false, it returns the root causes that are known to be false. If *SpecificEvent* is unknown, it returns the root causes that are unknown.

cdg-get-root-causes-of-diagnosis-manager
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *RootCauses*: sequence

Returns a sequence of cdg-specific-event objects, which are root causes of *DiagnosisManager* that are known to be true.

cdg-get-effects-of-event
(*SpecificEvent*: class cdg-specific-event)
-> *Effects*: sequence

Returns a sequence of cdg-specific-event objects, which are effects of *SpecificEvent*.

Diagnosis Managers

A diagnosis manager is a cdg-diagnosis-manager, which manages a set of correlated specific events and their specific external actions. You obtain a diagnosis manager from a specific event.

You can get the following information from a diagnosis manager:

- Known and suspected root causes, alarms, observed symptoms, effects, as well as all specific events managed by the diagnosis manager.
- Known and candidate tests, as well as all specific external actions managed by the diagnosis manager.
- The diagnosis status, which is a combination of the progress status, build status, and explanation status of the diagnosis manager.
- The chronological sequence of specific events and specific actions associated with the diagnosis manager.

The diagnosis status of the diagnosis manager depends on a combination of these factors:

- Is there an explanation, that is, a root cause for every observed event? In other words, does the set of known root causes explain the set of observed symptoms?
- Has the entire specific fault model been built, that is, have all possible causes of all observed events been built?
- Have all tests and repair actions that could provide additional information to the diagnostic reasoning process finished execution?

Getting and Deleting the Diagnosis Manager

cdg-get-diagnosis-manager
(*SpecificEvent*: class cdg-specific-event)
-> *DiagnosisManager*: class cdg-diagnosis-manager

Returns the cdg-diagnosis-manager for *SpecificEvent*.

cdg-delete-diagnosis-manager
(*DiagnosisManager*: class cdg-diagnosis-manager)

Deletes *DiagnosisManager* and all the events and actions that it manages.

Getting Specific Events

cdg-get-suspect-root-causes-for-diagnosis-manager
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *SuspectedRootCauses*: sequence

Returns a sequence of cdg-specific-event objects for *DiagnosisManager* that are suspected root causes, that is, root causes whose value is suspect.

cdg-get-known-root-causes-for-diagnosis-manager
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *KnownRootCauses*: sequence

Returns a sequence of cdg-specific-event objects for *DiagnosisManager* that are known root causes, that is, known to be "true" or known to be "false".

cdg-get-known-symptoms-for-diagnosis-manager
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *KnownSymptoms*: sequence

Returns a sequence of cdg-specific-event objects for *DiagnosisManager* that are observed symptoms, that is, effects, whose value has been specified at some point during the diagnostic session. Note that the current value of the specific event might have become false, based on downstream inference.

cdg-get-alarms-for-diagnosis-manager
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *Alarms*: sequence

Returns a sequence of cdg-specific-event objects for *DiagnosisManager* that are alarms, that is, specific events whose type is alarm.

cdg-get-known-effects-for-diagnosis-manager
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *KnownEffects*: sequence

Returns a sequence of cdg-specific-event objects for *DiagnosisManager* that are effects, that is, specific events that are not root causes.

cdg-get-specific-events
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *SpecificEvents*: sequence

Returns a sequence of all cdg-specific-event objects for *DiagnosisManager*.

cdg-get-number-of-unprocessed-incoming-events
( )
-> *Result*: integer

Returns the number of incoming events that are waiting to be propagated.

cdg-get-overridden-specified-events
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *specific-events*: sequence

Returns a sequence of specific events associated with *DiagnosisManager* whose last specified values differ from their current event values.

cdg-get-overridden-specified-events
(*DiagnosisManager*: class cdg-diagnosis-manager,
OverriddenValues: sequence)
-> *specific-events*: sequence

Returns a sequence of specific events associated with *DiagnosisManager* whose last specified values are a member of *OverriddenValues* and differ from their current event values. *OverridenValues* must be a subset of sequence("true", "false", "suspect"). This API allows you, for instance, to get all events that were specified to be true but have their values overridden by diagnostic inference by setting *OverriddenValues* to sequence("true"). Using all three values in the *OverriddenValues* sequence makes this API identical to the one without the *OverriddenValues* argument.

Getting Specific Actions

cdg-get-known-tests-for-diagnosis-manager
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *CompletedTests*: sequence

Returns a sequence of cdg-specific-test-action objects for *DiagnosisManager* that are completed tests.

cdg-get-candidate-tests-for-diagnosis-manager
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *CandidateTests*: sequence

Returns a sequence of cdg-specific-test-action objects for *DiagnosisManager* that are candidate tests, that is, tests that can be used to resolve suspected root causes.

cdg-get-specific-actions
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *SpecificActions*: sequence

Returns a sequence of all cdg-specific-action objects for *DiagnosisManager*.

Getting Diagnosis Information

cdg-get-diagnosis-completion-status
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *DiagnosisStatus*: symbol

Returns the diagnosis-status for *DiagnosisManager*, which can be complete or incomplete. The diagnosis is complete if any of the following conditions is satisfied:

- All observed events are fully explained by the set of known root causes, and the specific fault model managed by the diagnosis manager is fully built.
- All observed events are fully explained by the set of known root causes, and the cdg-terminate-diagnosis-early initialization parameter of the diagnosis manager is true.
- The diagnosis manager is fully built, and no further progress is possible without the arrival of external information.

cdg-get-diagnosis-explanation-status
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *ExplanationStatus*: truth-value

Returns true if *DiagnosisManager* has identified at least one known root cause that explains each observed symptom, false otherwise.

cdg-get-build-status
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *BuildStatus*: truth-value

Returns true if the specific fault model managed by *DiagnosisManager* has been fully built, that is, every event that can be associated with the diagnosis manager is included in the specific fault model.

cdg-get-diagnosis-progress-status
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *ProgressStatus*: truth-value

Returns true if no further progress is possible, false otherwise. The progress status is true if there are no scheduled actions associated with the diagnosis manager.

cdg-get-diagnosis-start-time
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *StartTime*: quantity

Returns the time at which *DiagnosisManager* was created.

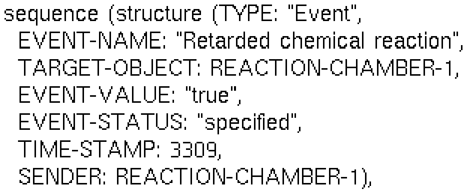
Re-Creating Event and Action Sequences for a Diagnosis Manager

cdg-recreate-events-sequence
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *EventsAndRootCauses*: sequence)

Re-creates the sequence in which incoming events arrive and root causes get their values. The sequence consists of structures with this syntax:

structure(type: *EventType*, event-name: *EventName*,
target-object: *TargetObject*, event-value: *EventValue*,
event-status: *EventStatus*, time-stamp: *TimeStamp*, sender: *Sender*)

For example, here is part of a returned event sequence:



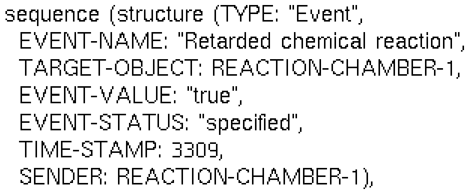
```
sequence (structure (TYPE: "Event",
  EVENT-NAME: "Retarded chemical reaction",
  TARGET-OBJECT: REACTION-CHAMBER-1,
  EVENT-VALUE: "true",
  EVENT-STATUS: "specified",
  TIME-STAMP: 3309,
  SENDER: REACTION-CHAMBER-1),
```

cdg-get-incoming-events-sequence
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *Events*: sequence)

Re-creates the sequence in which events arrive. The sequence consists of structures with this syntax:

structure(type: *EventType*, event-name: *EventName*,
target-object: *TargetObject*, event-value: *EventValue*,
event-status: *EventStatus*, time-stamp: *TimeStamp*, sender: *Sender*)

For example, here is part of a returned event sequence:



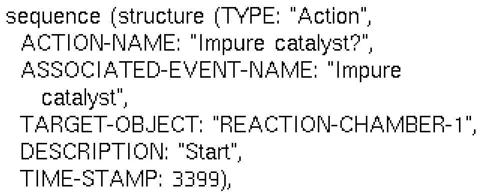
```
sequence (structure (TYPE: "Event",
  EVENT-NAME: "Retarded chemical reaction",
  TARGET-OBJECT: REACTION-CHAMBER-1,
  EVENT-VALUE: "true",
  EVENT-STATUS: "specified",
  TIME-STAMP: 3309,
  SENDER: REACTION-CHAMBER-1),
```

cdg-get-external-actions-sequence
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *Actions*: sequence)

Re-creates the sequence in which specific actions are initiated and completed. The sequence consists of structures with this syntax:

structure(type: *ActionType*, action-name: *ActionName*,
associated-event-name: *EventName*, description: *Description*,
time-stamp: *TimeStamp*)

For example, here is part of a returned event sequence:



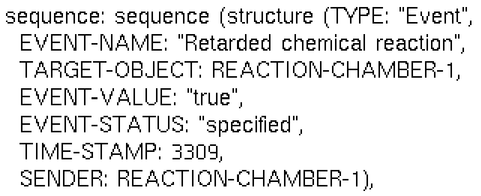
```
sequence (structure (TYPE: "Action",
  ACTION-NAME: "Impure catalyst?",
  ASSOCIATED-EVENT-NAME: "Impure
    catalyst",
  TARGET-OBJECT: "REACTION-CHAMBER-1",
  DESCRIPTION: "Start",
  TIME-STAMP: 3399),
```

cdg-recreate-events-and-actions-sequence
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *EventsActionsAndRootCauses*: sequence)

Re-creates the sequence in which incoming events arrive, actions are performed, and root causes get their values. The sequence consists of structures with this syntax:

structure(type: *Type*, event-name: *EventName*, target-object: *TargetObject*,
event-value: *EventValue*, event-status: *EventStatus*,
time-stamp: *TimeStamp*, sender: *Sender*)

For example:



```
sequence: sequence (structure (TYPE: "Event",
  EVENT-NAME: "Retarded chemical reaction",
  TARGET-OBJECT: REACTION-CHAMBER-1,
  EVENT-VALUE: "true",
  EVENT-STATUS: "specified",
  TIME-STAMP: 3309,
  SENDER: REACTION-CHAMBER-1),
```

cdg-get-root-cause-events-sequence
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *RootCauses*: sequence)

Re-creates the sequence in which root causes get their values. The sequence consists of structures with this syntax:

structure(type: *Type*, event-name: *EventName*, target-object: *TargetObject*,
event-value: *EventValue*, event-status: *EventStatus*,
time-stamp: *TimeStamp*)

For example, here is part of a returned event sequence:

```
sequence (structure (TYPE: "Event",
  EVENT-NAME: "Insufficient reagent",
  TARGET-OBJECT: REACTION-CHAMBER-1,
  EVENT-VALUE: "suspect",
  EVENT-STATUS: "upstream inferred",
  TIME-STAMP: 3309),
```

cdg-get-alarms-sequence
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *Alarms*: sequence)

Re-creates the sequence in which alarms occur and are predicted. The sequence consists of structures with this syntax:

structure(type: *Type*, event-name: *EventName*, target-object: *TargetObject*,
event-value: *EventValue*, event-status: *EventStatus*,
time-stamp: *TimeStamp*)

For example, here is part of a returned event sequence:

```
sequence (structure (TYPE: "Event",
  EVENT-NAME: "Impure product",
  TARGET-OBJECT: REACTION-CHAMBER-1,
  EVENT-VALUE: "true",
  EVENT-STATUS: "downstream inferred",
  TIME-STAMP: 3309),
```

Performing a Topological Sort

In graph theory, a topological sort of a directed acyclic graph (DAG) is a linear ordering of its nodes, which is compatible with the partial order R induced on the nodes where x comes before y (xRy) if there's a directed path from x to y in the DAG. An equivalent definition is that each node comes before all nodes to which it has edges. Every DAG has at least one topological sort and may have many.

The following API allows you to perform a topological sort of a specific fault model:

cdg-topological-sort
(*DiagnosisManager*: class cdg-diagnosis-manager,
UserDefinedProcedureName: symbol)

Traverses a fault model in the order of causality, that is, the traversal starts from the set of root causes, then goes to their immediate effects, then the next immediate effects, and so on. At each event, you can perform custom operations by applying a user-defined procedure. This ability complements an event's event-changed procedure, which is run only if an event changes, in that the user-defined procedure will be executed for every event managed by the diagnosis manager.

The user-defined procedure is applied in topological order to each event in the diagnosis manager and has this signature:

```
my-cdg-topological-sort-event-proc
   (Event: class cdg-specific-event)
```

The User Defined Procedures palette of the Fault Models toolbox contains an instance of this user-defined procedure called Graph Traversal Procedure/Method.

The topological sort API will not work on cyclical graphs. In practice, this limitation is likely to be insignificant.

Generic and Specific Events

An event is either a:

- Generic event, which is a cdg-generic-event that is identified by its target class and event name.
- Specific event, which is a cdg-specific-event that is identified by its target object and event name, or by its target object and generic event.

You can get a generic event from a specific event. You can get a specific event from its target object and event name, or from its target object and generic event.

You can get the following information for a specific event:

- Event name, description, event status, event value, target object, and sender.
- Current event state, previous event state, event state at a particular point in time, and event state history. Event state includes the value of the event-status and event-value of an event.
- Known root causes, plausible root causes, and evidence for root causes of an event. **Evidence** includes the symptoms, predicted events, and tests of an event, where a symptom is a specific event that has been observed and a predicted event is a specific event that is predicted but not observed.
- External test, repair, recovery, and mitigation actions of an event.

You can get the following information for a generic event:

- The procedure to call when the event-value changes for a corresponding specific event.
- The procedure to call when the event-value for a corresponding specific event does not change over a predetermined amount of time.

Getting Generic Event Information

cdg-get-generic-event
(*TargetClass*: symbol, *EventName*: text)
-> *GenericEvent*: item-or-value

Returns the cdg-generic-event defined by *EventName* and *TargetClass*. If the generic event does not exist, it returns the symbol none.

cdg-get-generic-event-for-specific-event
(*SpecificEvent*: class cdg-specific-event)
-> *GenericEvent*: class cdg-generic-event

Returns the cdg-generic-event corresponding with *SpecificEvent*.

cdg-get-specific-event-for-generic-event
(*GenericEvent*: class cdg-generic-event,
TargetObject: class grtl-domain-object)
-> *SpecificEvent*: item-or-value

Returns the cdg-specific-event defined by *GenericEvent* and *TargetObject*. If no such event exists, it returns the symbol none.

cdg-collect-generic-events-for-class
(*TargetClass*: symbol)
-> *GenericEvents*: sequence

Returns a sequence of cdg-generic-event objects defined for *TargetClass*.

cdg-collect-generic-events-for-view
(*GenericEventView*: cdg-generic-event-view)
-> *GenericEvents*: sequence

Returns a sequence of cdg-generic-event objects corresponding to *GenericEventView*.

cdg-collect-generic-events-for-action
(*GenericAction*: cdg-generic-action)
-> *GenericEvents*: sequence

Returns a sequence of cdg-generic-event objects that are acted on by *GenericAction*.

cdg-collect-generic-actions-for-event
(*GenericEvent*: cdg-generic-event)
-> *GenericActions*: sequence

Returns a sequence of cdg-generic-action objects assigned to *GenericEvent*.

cdg-collect-mutually-exclusive-events
(*GenericEvent*: cdg-generic-event)
-> *GenericEvents*: sequence

Returns a sequence of cdg-generic-event objects that are mutually exclusive to *GenericEvent*.

cdg-get-event-changed-procedure
(*GenericEvent*: class cdg-generic-event)
-> *EventChangedProcedure*: symbol

Returns the name of the event-changed-proc for *GenericEvent*.

cdg-get-event-unchanged-procedure
(*GenericEvent*: class cdg-generic-event)
-> *EventUnchangedProcedure*: symbol

Returns the event-unchanged-proc for *GenericEvent*.

Getting Specific Event Information

cdg-get-specific-event
(*TargetObject*: class grtl-domain-object, *EventName*: text)
-> *SpecificEvent*: item-or-value

Returns the cdg-specific-event defined by *TargetObject* and *EventName*. If the event does not exist, it returns the symbol none.

cdg-get-targetted-events
(*TargetObject*: class grtl-domain-object)
-> *SpecificEvents*: sequence

Returns a sequence of all cdg-specific-event objects associated with *TargetObject*.

cdg-get-event-name
(*SpecificEvent*: class cdg-specific-event)
-> *EventName*: text

Returns the event-name of *SpecificEvent*.

cdg-get-event-description
(*SpecificEvent*: class cdg-specific-event
-> *Description*: text

Returns the description of the cdg-generic-event that corresponds with *SpecificEvent*.

cdg-get-event-type
(*SpecificEvent*: class cdg-specific-event
-> *Type*: symbol

Returns the event-type of the cdg-generic-event that corresponds with *SpecificEvent*. The event type can be alarm, root-cause, or unspecified.

cdg-get-event-status
(*SpecificEvent*: class cdg-specific-event)
-> *EventStatus*: text

Returns the event-status of *SpecificEvent*. The status can be "specified", "upstream inferred", "downstream inferred", or "mutually exclusive".

cdg-get-event-value
(*SpecificEvent*: class cdg-specific-event)
-> *EventValue*: text

Returns the event-value of *SpecificEvent*. The value can be "true", "false", "suspect", or "unknown".

cdg-get-event-value-at-time
(*SpecificEvent*: class cdg-specific-event, *TimeStamp*: integer)
-> *EventValue*: text

Returns the event-value of *SpecificEvent* at *TimeStamp*.

cdg-get-last-specified-value
(*SpecificEvent*: class cdg-specific-event)
-> *EventValue*: text

Returns the last specified value of *SpecificEvent* ("true", "false", "suspect", or "unknown"). The default last specified value of an event is "unknown" if no value has been specified for that event.

cdg-get-last-specified-time-stamp
(*SpecificEvent*: class cdg-specific-event)
-> *TimeStamp*: quantity

Returns the timestamp of *SpecificEvent* when its value was last specified. Default = 0.0 if no value has been specified for the event.

cdg-get-event-target
(*SpecificEvent*: class cdg-specific-event)
-> *TargetObject*: class grtl-domain-object

Returns the target-object for *SpecificEvent*.

cdg-get-event-sender
(*SpecificEvent*: class cdg-specific-event)
-> *Sender*: item-or-value

Returns the event source for *SpecificEvent* or the symbol none.

cdg-get-event-state-changes
(*SpecificEvent*: class cdg-specific-event, *TimePeriod*: quantity)
-> *state-changes*: sequence

Returns a sequence of event state changes of an event within a time period ending at the current time. A large number of such changes in a very short time could indicate an unstable and oscillating system. Such behavior can occur if the fault model and the data are inconsistent. You can use this API to detect such behavior.

The return value is a sequence of structures, ordered in non-decreasing order based on the timestamp, that is, from lowest to highest. Each structure has these attributes:

```
structure
    (EVENT-VALUE: text,
    EVENT-STATUS: text,
    TIME-STAMP: quantity)
```

cdg-get-event-value-changes
(*SpecificEvent*: class cdg-specific-event, *TimePeriod*: quantity)
-> *value-changes*: sequence

Returns a sequence of event value changes of an event within a time period ending at the current time. The return value is the same as for cdg-get-event-state-changes.

Getting Specific Event Message Information

cdg-get-specific-event-message-text
(*SpecificEvent*: class cdg-specific-event)
-> *Message*: text

Returns the message text of *SpecificEvent* with text substitutions.

cdg-get-specific-event-message-detail
(*SpecificEvent*: class cdg-specific-event):
-> *Detail*: text

Returns the message detail of *SpecificEvent* with text substitutions.

cdg-get-specific-event-message-advice
(*SpecificEvent*: class cdg-specific-event)
-> *Advice*: text

Returns the message advice of *SpecificEvent* with text substitutions.

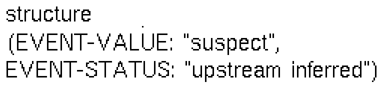
Getting the State of Specific Events

cdg-get-state-of-event
(*TargetObject*: class grtl-domain-object, *EventName*: text)
-> *EventState*: structure

Returns a structure containing the value and status of the cdg-specific-event defined by *EventName* and *TargetObject*. If the event does not exist, it returns structure(event-value: "", event-status: ""). The sequence consists of structures with this syntax:

structure(event-value: *EventValue*, event-status: *EventStatus*)

For example:



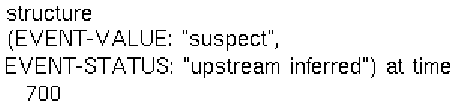
```
structure
 (EVENT-VALUE: "suspect",
 EVENT-STATUS: "upstream inferred")
```

cdg-get-state-of-event-at-time
(*TargetObject*: class grtl-domain-object, *EventName*: text, *TimeStamp*: integer)
-> *EventState*: structure

Returns a structure containing the value and status of the cdg-specific-event defined by *EventName* and *TargetObject* at *TimeStamp*. If the event does not exist, it returns structure(event-value: "", event-status: ""). The sequence consists of structures with this syntax:

structure(event-value: *EventValue*, event-status: *EventStatus*)

For example:



```
structure
(EVENT-VALUE: "suspect",
EVENT-STATUS: "upstream inferred") at time
  700
```

cdg-get-previous-state-of-event-at-time
(*TargetObject*: class grtl-domain-object, *EventName*: text, *TimeStamp*: integer)
-> *EventState*: structure

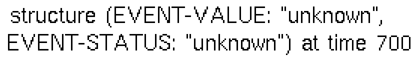
Returns a structure containing the previous value and status of the cdg-specific-event defined by *EventName* and *TargetObject* at *TimeStamp*. The sequence consists of structures with this syntax:

structure(event-value: *EventValue*, event-status: *EventStatus*)

If the event does not exist, it returns:

structure(event-value: "", event-status: "")

For example:



```
structure (EVENT-VALUE: "unknown",
EVENT-STATUS: "unknown") at time 700
```

cdg-get-event-status-at-time
(*SpecificEvent*: class cdg-specific-event, *TimeStamp*: integer)
-> *EventStatus*: text

Returns the event-status of *SpecificEvent* at *TimeStamp*.

cdg-get-previous-event-status
(*SpecificEvent*: class cdg-specific-event)
-> *EventStatus*: text

Returns the event-status of the previous state of *SpecificEvent*.

cdg-get-previous-event-value
(*SpecificEvent*: class cdg-specific-event)
-> *EventValue*: text

Returns the event-value of the previous state of *SpecificEvent*.

cdg-get-event-history
(*SpecificEvent*: class cdg-specific-event)
-> *StateChangeHistory*: sequence

Returns a sequence of state changes for *SpecificEvent*. The history is a sequence of structures that mark the changes in the state of the event. Each structure consists of one of the following, depending on *PreviousStatus*:

- If *PreviousStatus* = "specified"

  structure(event-value: *PreviousValue*, event-status: *PreviousStatus*, time-stamp: the time-stamp of *SpecificEvent*, sender: *PreviousSender*)

- If *PreviousStatus* /= "specified"

  structure(event-value: *PreviousValue*, event-status: *PreviousStatus*, time-stamp: the time-stamp of *SpecificEvent*)

where:

PreviousStatus is the value of the event-status of the previous event, which is a text string.

PreviousValue is the value of the event-value of the previous event, which is a text string.

The event history does not include information on the current event.

For example:

```
sequence
 (structure (EVENT-VALUE: "suspect",
   EVENT-STATUS: "upstream inferred",
   TIME-STAMP: 3309))
```

Getting Fraction of Causes and Effects

The following APIs can be used to compute the fraction (or ratio) of causes or effects of an event that have a particular value. The results of the API may be interpreted by an application as the degree of degradation or the level of confidence in the event, as needed.

cdg-get-fraction-of-true-causes
(*SpecificEvent*: class cdg-specific-event)
-> *Fraction*: float

Returns the ratio of causes of *SpecificEvent* that are true to all the causes of *SpecificEvent*.

cdg-get-fraction-of-true-effects
(*SpecificEvent*: class cdg-specific-event)
-> *Fraction*: float

Returns the ratio of effects of *SpecificEvent* that are true to all the effects of *SpecificEvent*.

cdg-get-fraction-of-false-causes
(*SpecificEvent*: class cdg-specific-event)
-> *Fraction*: float

Returns the ratio of causes of *SpecificEvent* that are false to all the causes of *SpecificEvent*.

cdg-get-fraction-of-false-effects
SpecificEvent: class cdg-specific-event)
-> *Fraction*: float

Returns the ratio of effects of *SpecificEvent* that are false to all the effects of *SpecificEvent*.

cdg-get-fraction-of-suspect-causes
(*SpecificEvent*: class cdg-specific-event)
-> *Fraction*: float

Returns the ratio of suspected causes of *SpecificEvent* that are true to all the suspected causes of *SpecificEvent*.

cdg-get-fraction-of-suspect-effects
(*SpecificEvent*: class cdg-specific-event)
-> *Fraction*: float

Returns the ratio of suspected effects of *SpecificEvent* that are true to all the suspected effects of *SpecificEvent*.

cdg-get-fraction-of-unknown-causes
(*SpecificEvent*: class cdg-specific-event)
-> *Fraction*: float

Returns the ratio of unknown causes of *SpecificEvent* that are true to all the unknown causes of *SpecificEvent*.

cdg-get-fraction-of-unknown-effects
(*SpecificEvent*: class cdg-specific-event)
-> *Fraction*: float

Returns the ratio of unknown effects of *SpecificEvent* that are true to all the unknown effects of *SpecificEvent*.

Getting and Setting User-Defined Data

Specific events might need to store user-defined data, for example, the likelihood of an event or the degree of degradation of an event. This feature is necessary because SymCure cannot predict beforehand what attributes should be provided, and SymCure does not allow sub-classing of specific events.

User-defined attributes can be displayed automatically in the properties dialogs of specific events provided that the user-defined data is populated as a sequence of structures and that the attributes of the structures are quantitative, symbolic, or text values. SymCure's import and export capabilities also handle these attributes automatically. Currently, user-defined attributes are imported as text values.

To support this feature, specific events provide the following APIs for getting and setting custom information for a specific event:

cdg-set-user-defined-data
(*SpecificEvent*: class cdg-specific-event, *UserDefinedData*: sequence)

cdg-get-user-defined-data
(*SpecificEvent*: class cdg-specific-event)
-> *Data*: sequence

Typically, you populate the sequence with structures, for example, sequence(structure(likelihood: 0.1, degree-of-degradation: 0.9)).

Getting Explanations and Evidence for Specific Events

You can get the following information for a specific event:

- Known explanations, which is a path from a known root cause to an alarm, where the value of the root cause justifies the value of the alarm. An event is known if its value is true or false.
- Plausible explanations, which is a path from a plausible root cause to an alarm. Plausible root causes have a value of true, false or suspect, depending on the value of the corresponding alarm. If the alarm is true, all suspect or true root cause are plausible root causes for explaining the alarm's value. If the alarm is false, all false root causes are plausible explanations. If the alarm is unknown, all unknown root causes are included in its plausible explanations.

Evidence, which includes the symptoms, predicted events, and tests for a specific event.

Getting known and plausible explanations is similar to getting known and plausible root causes, using cdg-get-root-causes and cdg-get-plausible-root-causes-of-event, respectively. For a description of these APIs, see Root Causes.

The following examples provide a comparison between known and plausible root causes, and known and plausible explanations, using the following four APIs:

1. cdg-get-root-causes-of-event
2. cdg-get-plausible-root-causes-of-event
3. cdg-get-known-explanations
4. cdg-get-plausible-explanations

Suppose you have the following specific fault model:

A->B, B->C, C->D

where C is an alarm and A and D are root causes.

Case 1. Suppose that C is true, and A and D are suspect.

Using C as the argument, the four APIs return the following results:

1. {} as there are no known causes for C
2. {A, D} each of which is a plausible cause for C
3. {} as there are no known causes for C
4. {A->B->C, D->C}, which specify the paths from all the plausible causes of C to itself

Case 2. Suppose that C is true, and A is true while D is suspect.

Using C as the argument, the four APIs will return the following results:

1. {A}
2. {A, D} each of which is a plausible cause for C
3. {A->B->C}
4. {A->B->C, D->C}, which specify the paths from all the plausible causes of C to itself

Case 3. Suppose that C is false (true), A is false (true), D is false (true).

1. {A, D} each of which is a known cause for C
2. {A, D} each of which is a plausible cause for C
3. {A->B->C, D->C}
4. {A->B->C, D->C}

Thus, plausible root causes and plausible explanations are always supersets of known root causes and known explanations.

Note that the path A->B->C in the examples above is represented by the following structure in the APIs that follow:

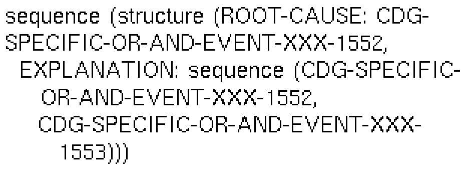
structure(Root-cause: A, Explanation: sequence(A, B, C)).

cdg-get-known-explanations
(*SpecificEvent*: class cdg-specific-event)
-> *KnownExplanations*: sequence

Returns a sequence of cdg-specific-event objects that are known explanations of *SpecificEvent*. Each explanation is a sequence that defines the path from a root cause to *SpecificEvent*. The sequence consists of structures with this syntax:

structure(root-cause: *RootCause*,
explanation: (sequence *SpecificEvent*[, ...]))

For example:



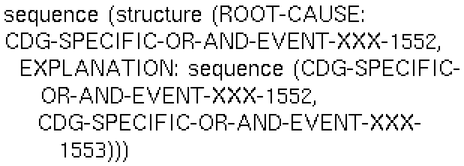
```
sequence (structure (ROOT-CAUSE: CDG-
SPECIFIC-OR-AND-EVENT-XXX-1552,
  EXPLANATION: sequence (CDG-SPECIFIC-
    OR-AND-EVENT-XXX-1552,
    CDG-SPECIFIC-OR-AND-EVENT-XXX-
      1553)))
```

cdg-get-plausible-explanations
(*SpecificEvent*: class cdg-specific-event)
-> *PlausibleExplanations*: sequence

Returns a sequence of cdg-specific-event objects that are suspected explanations of *SpecificEvent*. Each explanation is a sequence that defines the path from a root cause to *SpecificEvent*. The sequence consists of structures with this syntax:

structure(root-cause: *RootCause*,
explanation: (sequence *SpecificEvent*[, ...]))

For example:



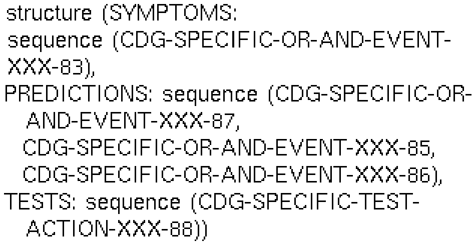
```
sequence (structure (ROOT-CAUSE:
CDG-SPECIFIC-OR-AND-EVENT-XXX-1552,
  EXPLANATION: sequence (CDG-SPECIFIC-
    OR-AND-EVENT-XXX-1552,
    CDG-SPECIFIC-OR-AND-EVENT-XXX-
      1553)))
```

cdg-get-evidence-for-root-cause
(*SpecificEvent*: class cdg-specific-event)
-> *Evidence*: structure

Returns a structure containing three sequences for the cdg-specific-event objects that are symptoms, predicted events, and tests of *SpecificEvent*. The structure has the following syntax:

structure(symptoms: (sequence *SymptomEvent* [, ...]),
predictions: (sequence *PredictedEvent*[, ...]),
tests: (sequence *SupportingTest*[, ...])

For example:



```
structure (SYMPTOMS:
sequence (CDG-SPECIFIC-OR-AND-EVENT-
XXX-83),
PREDICTIONS: sequence (CDG-SPECIFIC-OR-
   AND-EVENT-XXX-87,
  CDG-SPECIFIC-OR-AND-EVENT-XXX-85,
  CDG-SPECIFIC-OR-AND-EVENT-XXX-86),
TESTS: sequence (CDG-SPECIFIC-TEST-
   ACTION-XXX-88))
```

Getting Specific Actions of Specific Events

cdg-get-external-actions
(*SpecificEvent*: class cdg-specific-event)
-> *ExternalActions*: sequence

Returns a sequence of cdg-specific-action objects that are external actions of *SpecificEvent*, including test, mitigation, repair, and recovery actions.

cdg-get-external-test-actions
(*SpecificEvent*: class cdg-specific-event)
-> *Tests*: sequence

Returns a sequence of cdg-specific-test-action objects that are external test actions of *SpecificEvent*.

cdg-get-external-repair-actions
(*SpecificEvent*: class cdg-specific-event)
-> *RepairActions*: sequence

Returns a sequence of cdg-specific-repair-action objects that are external repair actions of *SpecificEvent*.

cdg-get-external-recovery-actions
(*SpecificEvent*: class cdg-specific-event)
-> *RecoveryActions*: sequence

Returns a sequence of cdg-specific-recovery-action objects that are external recovery actions of *SpecificEvent*.

cdg-get-external-mitigation-actions
(*SpecificEvent*: class cdg-specific-event)
-> *MitigationActions*: sequence

Returns a sequence of cdg-specific-mitigation-action objects that are external mitigation actions of *SpecificEvent*.

cdg-get-downstream-external-actions
(*SpecificEvent*: class cdg-specific-event)
-> *Actions*: sequence

Returns a sequence of all cdg-specific-action objects that are associated with events that are downstream of *SpecificEvent*.

cdg-get-downstream-tests
(*SpecificEvent*: class cdg-specific-event)
-> *DownstreamTests*: sequence

Returns a sequence of cdg-specific-test-action objects that are associated with specific events that are downstream of *SpecificEvent*, where *SpecificEvent* is assumed to be a root cause. The sequence includes tests for all downstream events, including those that might not be used to resolve the value of the particular specific event.

cdg-get-candidate-tests
(*SpecificEvent*: class cdg-specific-event)
-> <u>*CandidateTests*</u>: sequence

Returns a sequence of cdg-specific-test-action objects that are candidate tests for *SpecificEvent*, where *SpecificEvent* is assumed to be a root cause. The result includes the subset of the downstream tests that can be used to resolve the value of *SpecificEvent* and that have not yet been run.

Upgrading Message Attributes

cdg-upgrade-message-attributes-for-folder
(*DiagramFolder*: class cdg-diagram-folder, *EventValue*: text)

Copies the old message attributes of each generic event in *DiagramFolder*, which includes message text, message detail, and message advice, to the new value dependent attributes for the generic event as specified by *EventValue*, and enables operator messages for *EventValue*. This API is recursively applied to all fault model folders on the subworkspace of *DiagramFolder*.

For example, to upgrade events in fault model folder DF-1 to use the old message attributes when the corresponding specific events are false, call cdg-upgrade-message-attributes-for-folder(DF-1, "false").

cdg-upgrade-message-attributes
(*EventValue*: text)

Converts message attributes for all generic events in your application as specified by the *EventValue* argument, without regard to their fault model folder.

For example, to upgrade events to use the old message attributes for operator messages when the underlying specific event is true, call cdg-upgrade-message-attributes("true").

External Actions

An external action is a cdg-generic-action that defines a test, repair, recovery, or mitigation action associated with a specific event. An external action is either a:

- Generic action — A cdg-generic-action that is identified by its target class and action name.
- Specific action — A cdg-specific-action that is identified by its target object and action name.

An external action is automatically enabled when the value of the underlying event changes, depending on the type of action, as follows:

- For a test action, when the underlying event changes from any value to "suspect" or "unknown".
- For a repair action, when the underlying event value changes from any value to "true".
- For a recovery action, when the underlying event value changes from "true" to "false".
- For a mitigation action, when the underlying event changes from any value to "suspect" or "true".

If an external action is enabled, it executes either automatically or manually, depending on its activation-type. You can also execute an external action programmatically.

You can get specific external actions from generic actions. You can also get generic actions from specific actions, target classes, and generic events.

An external action defines the following information, which you can get:

- Execution information, which includes whether the action has been requested, that is, enabled, based on the status of the underlying event; the status of the action; and the execution history of the action.
- Known and plausible root causes associated with the action.
- Cost associated with executing the action.
- Unique tag.
- Underlying root causes.

You can also programmatically send the result of a test or repair action.

Getting Generic Actions

cdg-get-generic-action
(*TargetClass*: symbol , *ActionName*: text)
-> *GenericAction*: item-or-value

Returns the cdg-generic-action defined by *ActionName* for *TargetClass*. If the action does not exist, it returns the symbol none.

cdg-get-generic-action-for-specific-action
(*SpecificAction*: class cdg-specific-action)
-> *generic-action*: class cdg-generic-action

Returns the cdg-generic-action corresponding to *SpecificAction*.

cdg-collect-generic-actions-for-class
(*TargetClass*: symbol)
-> *GenericActions*: sequence

Returns a sequence of cdg-generic-action objects defined for *TargetClass*.

cdg-collect-generic-actions-for-event
(*GenericEvent*: cdg-generic-event)
-> *GenericActions*: sequence

Returns a sequence of cdg-generic-action objects associated with *GenericEvent*.

Getting Specific Actions and Information

cdg-get-specific-action
(*TargetObject*: grtl-domain-object, *ActionName*: text)
-> *SpecificAction*: item-or-value

Returns the cdg-specific-action defined by *TargetObject* and *ActionName*. If the action does not exist, it returns the symbol none.

cdg-get-associated-events
(*SpecificAction*: cdg-specific-action)
-> *SpecificEvents*: sequence

Returns a sequence of all cdg-specific-event objects associated with *SpecificAction*.

cdg-get-cost
(*SpecificAction*: class cdg-specific-action)
-> *Cost*: float

Returns the cost of *SpecificAction*.

cdg-get-underlying-root-causes
(*SpecificAction*: class cdg-specific-action)
-> *RootCauses*: sequence

Get the root causes that underlie the last execution of *SpecificAction*.

cdg-get-action-history
(SpecificAction: class cdg-specific-action)
-> *History*: text

Gets the history of all completed executions of *SpecificAction*.

cdg-set-tag
(*SpecificAction*: class cdg-specific-action, *Tag*: text)

Overwrites the default tag of *SpecificAction* with *Tag*.

Note: When using this API, you must ensure that each tag is unique.

cdg-get-tag
(*SpecificAction*: class cdg-specific-action)
-> *Tag*: text

Gets the tag of *SpecificAction*.

Getting Action Execution Information

cdg-is-requested
(*SpecificAction*: class cdg-specific-action)
-> *Reqested*: truth-value

Returns true if *SpecificAction* has been requested for execution.

cdg-get-execution-status
(*SpecificAction*: class cdg-specific-action)
-> *ActionStatus*: symbol

Returns the action-status of *SpecificAction*. The action status is enabled, running or inactive.

cdg-reset-execution-status
(*SpecificAction*: class cdg-specific-action)

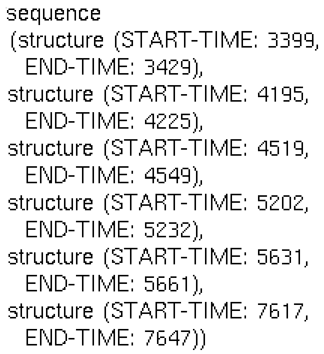
Resets the action-status of *SpecificAction*, that is, sets it to inactive.

cdg-get-action-history
(*SpecificAction*: class cdg-specific-action)
-> *History*: sequence

Returns a sequence of the history of *SpecificAction*. The history includes the time at which the action was started and completed. The sequence consists of structures with this syntax:

structure(start-time: *TimeStamp*, end-time: *TimeStamp*)

For example:



```
sequence
 (structure (START-TIME: 3399,
   END-TIME: 3429),
 structure (START-TIME: 4195,
   END-TIME: 4225),
 structure (START-TIME: 4519,
   END-TIME: 4549),
 structure (START-TIME: 5202,
   END-TIME: 5232),
 structure (START-TIME: 5631,
   END-TIME: 5661),
 structure (START-TIME: 7617,
   END-TIME: 7647))
```

cdg-get-action-result
(*SpecificAction*: class cdg-specific-action)
-> *return-value*: text

Gets the last result of executing *SpecificAction*.

Getting Specific Action Message Information

cdg-get-specific-action-message-text
(*SpecificAction*: class cdg-specific-action)
-> *Message*: text

Returns the message text for *SpecificAction*, with text substitutions.

cdg-get-specific-action-message-detail
(*SpecificAction*: class cdg-specific-action)
-> *Detail*: text

Returns the message detail for *SpecificAction*, with text substitutions.

cdg-get-specific-action-message-advice
(*SpecificAction*: class cdg-specific-action)
-> *Advice*: text

Returns the message advice for *SpecificAction*, with text substitutions.

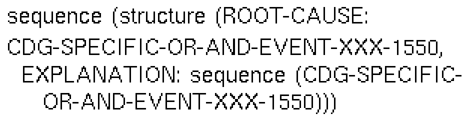
Getting Explanations for Actions

cdg-get-known-explanations
(*SpecificAction*: class cdg-specific-action)
-> sequence)

Returns a sequence of cdg-specific-event objects that are known explanations for *SpecificAction*. Each explanation is a sequence that defines a path from a root cause to the event associated with *SpecificAction*. The sequence consists of structures with this syntax:

structure(root-cause: *RootCause*,
explanation: (sequence *SpecificEvent*[, ...]))

For example:



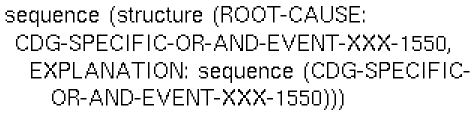
```
sequence (structure (ROOT-CAUSE:
CDG-SPECIFIC-OR-AND-EVENT-XXX-1550,
  EXPLANATION: sequence (CDG-SPECIFIC-
    OR-AND-EVENT-XXX-1550)))
```

cdg-get-plausible-explanations
(*SpecificAction*: class cdg-specific-action)
-> sequence)

Returns a sequence of cdg-specific-event objects that are suspected explanations for *SpecificAction*. Each explanation is a sequence that defines a path from a root cause to the event associated with *SpecificAction*. The sequence consists of structures with this syntax:

structure(root-cause: *RootCause*,
explanation: (sequence *SpecificEvent*[, ...]))

For example:



```
sequence (structure (ROOT-CAUSE:
  CDG-SPECIFIC-OR-AND-EVENT-XXX-1550,
    EXPLANATION: sequence (CDG-SPECIFIC-
      OR-AND-EVENT-XXX-1550)))
```

Setting Enabling Transitions

cdg-set-enabling-transitions
(*SpecificAction*: class cdg-specificaction, *EnablingTransitions*: sequence)

Disables enabling transitions for a specific action. *EnablingTransitions* is a sequence that is the new enabling transitions for *SpecificAction*, which is an empty sequence, by default.

Sending Action Results

Use these APIs to execute actions and send action results.

cdg-execute-action
(*SpecificAction*: class cdg-specific-action, *Window*: class g2-window)

Executes *SpecificAction* and displays any results on *Window*.

cdg-send-action-result
(*SpecificAction*: class cdg-specific-action, *Result*: text , *Cost*: value,
Win: class ui-client-item)

Sends the result of the action to each specific event associated with the action. *Result* is the resulting event value, which can be "true" or "false" for a test, or "false" for a repair action. If *Cost* is a quantity, the cost of the specific action is updated.

cdg-send-action-result
(*TargetObject*: class grtl-domain-object, *ActionName*: text, *Result*: text,
Cost: value, *Client*: class object)

Sends the result and cost for the action defined by *TargetObject* and *ActionName*.

Fault Model Folders

You create SymCure fault models on the subworkspace of a cdg-diagram-folder. You can get errors and messages for fault model folders.

cdg-get-diagram-folder-errors
(*DiagramFolder*: class cdg-diagram-folder)
-> *Errors*: sequence

Returns a sequence of errors for *DiagramFolder*. For a list of possible errors, see Viewing Errors.

cdg-get-diagram-folder-warnings
(*DiagramFolder*: class cdg-diagram-folder)
-> *Warnings*: sequence

Returns a sequence of warnings for *DiagramFolder*. For a list of possible warnings, see Viewing Warnings.

Debugging

cdg-enable-fault-model-debugging
(*enable*: truth-value)

Enables or disables the cdg-enable-debugging initialization parameter. The API has no effect if *enable* is the same as cdg-enable-debugging. When *enable* is true, debugging is enabled. This API procedure also enables the menu choice for accessing the debugging control panel. Any subsequent events processed by SymCure will be logged and therefore available for graphical debugging. When *enable* is false, debugging is disabled and the internal debugging log is emptied.

This procedure is the equivalent of choosing Project > Logic > Diagnose > Debug Specific Fault Models > Enable Debugging. For more information, see Debugging SymCure Fault Models.

Root Cause Episode Management

Use these APIs for starting and stopping episode management for any domain object:

cdg-start-root-cause-episode-management
(*DomainObject*: class grtl-domain-object,
EpisodePersistenceInterval: integer)

Initializes a root cause episode archive manager for *DomainObject* that manages all of its root cause episode archives. *EpisodePersistenceInterval* is the interval to persist all root cause episodes after their completion.

cdg-stop-root-cause-episode-management
(*DomainObject*: class grtl-domain-object)

Deletes all root cause episodes, archives, and stop root cause episode management for *DomainObject*. This API deletes all archives, without regard to the episode persistence interval of the Root Cause Episode Archive Manager.

Use these APIs for getting root cause episodes:

cdg-get-root-cause-episodes
(*DomainObject*: class grtl-domain-object, *RootCauseEventName*: text)
-> *episodes*: sequence

Gets all episodes of *RootCauseEventName* on *DomainObject*. Episodes are ordered in reverse chronology, i.e., from the latest to the earliest.

cdg-get-root-cause-episodes
(*DomainObject*: class grtl-domain-object, *RootCauseEventName*: text,
StartTime: quantity, *EndTime*: quantity)
-> *episodes*: sequence

Gets all episodes of *RootCauseEventName* on *DomainObject* that overlap with the duration specified by *StartTime* and *EndTime*. Episodes are ordered in reverse chronology, i.e., from the latest to the earliest. Note that if *EndTime* is less than *StartTime*, an empty sequence is returned.

Use these APIs to get episode information:

cdg-get-transitions
(*Episode*: class cdg-root-cause-episode)
-> *transitions*: sequence

Gets transitions for the episode. *Transitions* is a sequence of structures. Each structure includes the transition ("suspected, "exonerated", "detected", or "resolved") and the timestamp for the transition. *Transitions* is ordered chronologically according to timestamps. Note that oscillating transitions between "detected" and "suspected" are not stored in the episode.

An example of the transitions sequence is:

```
sequence
    (structure (TRANSITION: "suspected",
                TIME-STAMP: 669.82),
    structure (TRANSITION: "detected",
                TIME-STAMP: 737.498),
    structure (TRANSITION: "resolved",
            TIME-STAMP: 777.026))
```

When using these APIs, note the following:

- You can start root cause episode management at any time, even during an existing diagnostic session. Root cause episode management commences from that point onwards, but any diagnostic activity prior to the starting point is ignored.
- A domain object has at most one root cause episodes archive manager. SymCure wont let you create a new one for a domain object if one already exists.
- Garbage collection is performed at periodic intervals to delete any old episodes that may no longer be relevant to the fault management application. The interval in seconds between successive garbage collections is controlled by the following parameter in `config.txt`:

  ```
  CDG-EPISODE-DELETION-MONITOR-INTERVAL=86400
  ```

During garbage collection, any episodes that have been completed before the episode manager's persistence interval are deleted, thus freeing up memory for other tasks.

- Episodes, archives, and their managers are not designed to have iconic representations to optimize memory usage.

Charting

cdg-show-chart
(*Target*: class grtl-domain-object, *Subsets*: sequence, *Title*: text, *Subtitle*: text, *XAxisLabel*: text, *YAxisLabel*: text, *Win*: class g2-window)

Plots a general chart, according to the specification. For example:

```
start cdg-show-chart
(reaction-chamber-1, sequence(structure(label: "Retarded chemical reaction",
data: sequence(5, 5, 7)), structure(label: "Impure reagent",
data: sequence(4, 0, 9)), structure(label: "Impure catalyst",
data: sequence(2, 2, 2), "Root Cause Episodes", "Downtime per week", "Weeks",
"Durations", this window)
```

The following APIs are used to plot aggregated durations of root cause episodes for a target object:

cdg-show-aggregated-duration-chart-from-start-time-to-end-time
(*Target*: class grtl-domain-object, *StartTime*: quantity, *EndTime*: quantity, *PeriodLength*: quantity, *Win*: class g2-window)

cdg-show-aggregated-duration-chart-from-start-time-with-lookahead
(*Target*: class grtl-domain-object, *StartTime*: quantity, *Lookahead*: quantity, *PeriodLength*: quantity, *Win*: class g2-window)

cdg-show-aggregated-duration-chart-from-end-time-with-lookback
(*Target*: class grtl-domain-object, *EndTime*: quantity, *Lookback*: quantity, *PeriodLength*: quantity, *Win*: class g2-window)

The following APIs are used to plot frequencies of root cause episodes for a target object:

cdg-show-frequency-chart-from-start-time-to-end-time
(*Target*: class grtl-domain-object, *StartTime*: quantity, *EndTime*: quantity, *PeriodLength*: quantity, *Win*: class g2-window)

cdg-show-frequency-chart-from-start-time-with-lookahead
(*Target*: class grtl-domain-object, *StartTime*: quantity, *Lookahead*: quantity, *PeriodLength*: quantity, *Win*: class g2-window)

cdg-show-frequency-chart-from-end-time-with-lookback
(*Target*: class grtl-domain-object, *EndTime*: quantity, *Lookback*: quantity, *PeriodLength*: quantity, *Win*: class g2-window)

Run-Time Behavior

cdg-pause
( )

Pauses event propagation. Following a call to this API, all incoming events are queued but not processed. The suspension of event propagation cannot occur in the middle of the propagation in response to an incoming event. Thus, if event propagation in response to an incoming event is in effect at the instant that cdg-pause is called, SymCure loops waiting for 1 second before attempting to pause event propagation again. Therefore, a call to cdg-pause might not take effect instantaneously.

cdg-resume
( )

Resumes event propagation. The incoming events queue is processed in the order that events arrived. If SymCure is not paused, a call to cdg-resume has no effect.

cdg-is-diagnostic-processing-active
( )
-> *Result*: truth-value

Returns true if event propagation is currently taking place; otherwise, returns false. This API procedure also returns false when SymCure is paused.

cdg-is-online
( )
-> *Result*: truth-value

Returns true if SymCure is ready to receive events; false if it is not yet ready.

cdg-take-offline
()

Takes SymCure offline. While SymCure is offline, all event propagation is suspended and any incoming events are ignored.

cdg-take-online
()

Takes SymCure online.

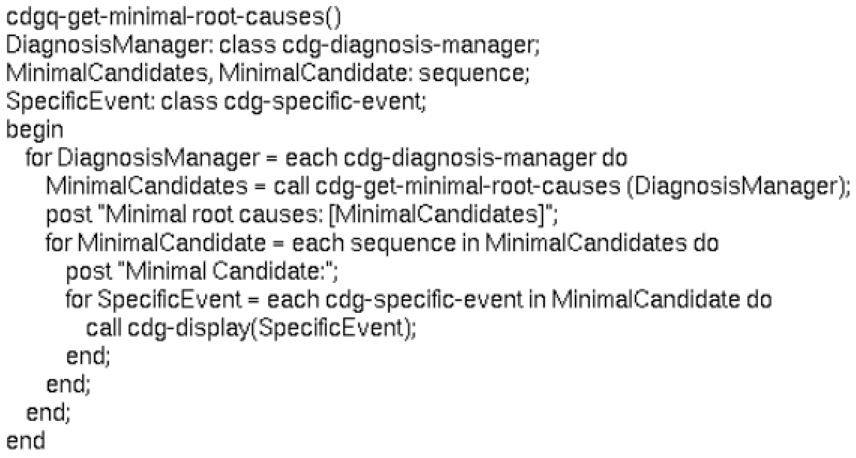
Minimal Candidates

A minimal candidate is the smallest set of root causes that must be true to explain all known symptoms. Any superset of a minimal candidate is not a minimal candidate. The notion of minimal candidates is particularly useful for analyzing a diagnosis problem when you have a number of suspected root causes, but you don't have enough information to resolve them. By extracting the minimal candidates for a diagnosis problem, you can identify the minimal combinations of root causes that must occur to explain your symptoms.

Here is the signature for the API:

cdg-get-minimal-root-causes
(*DiagnosisManager*: class cdg-diagnosis-manager)
-> *MinimalCandiates*: sequence

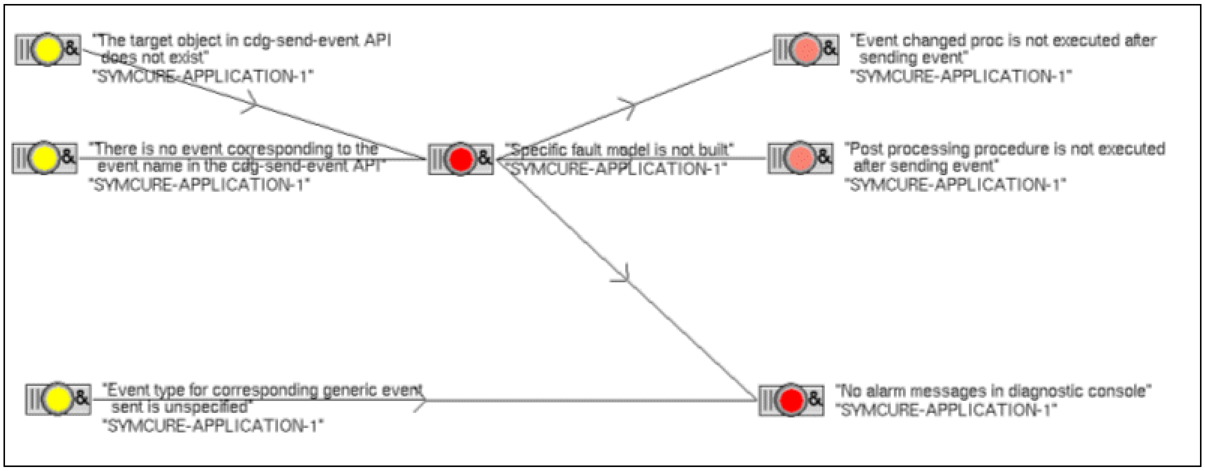
Here is an example of the use of this procedure:



```
cdgq-get-minimal-root-causes()
DiagnosisManager: class cdg-diagnosis-manager;
MinimalCandidates, MinimalCandidate: sequence;
SpecificEvent: class cdg-specific-event;
begin
  for DiagnosisManager = each cdg-diagnosis-manager do
    MinimalCandidates = call cdg-get-minimal-root-causes (DiagnosisManager);
    post "Minimal root causes: [MinimalCandidates]";
    for MinimalCandidate = each sequence in MinimalCandidates do
      post "Minimal Candidate:";
      for SpecificEvent = each cdg-specific-event in MinimalCandidate do
        call cdg-display(SpecificEvent);
      end;
    end;
  end;
end
```

Here is a specific fault model with two symptoms:

- "No alarm messages in diagnostic console" on SYMCURE-APPLICATION-1
- "Specific fault model is not built" on SYMCURE-APPLICATION-1



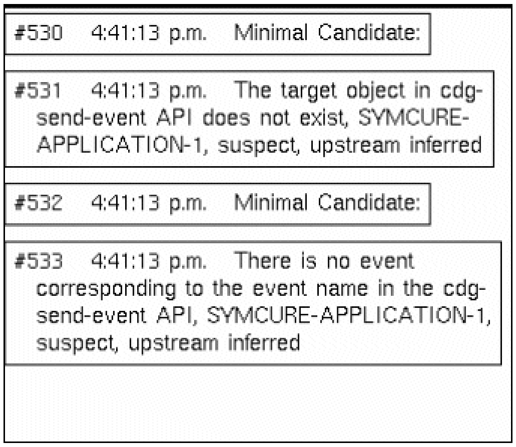
"The target object in cdg-send-event API does not exist" "SYMCURE-APPLICATION-1"
"There is no event corresponding to the event name in the cdg-send-event API" "SYMCURE-APPLICATION-1"
"Event type for corresponding generic event sent is unspecified" "SYMCURE-APPLICATION-1"
"Specific fault model is not built" "SYMCURE-APPLICATION-1"
"Event changed proc is not executed after sending event" "SYMCURE-APPLICATION-1"
"Post processing procedure is not executed after sending event" "SYMCURE-APPLICATION-1"
"No alarm messages in diagnostic console" "SYMCURE-APPLICATION-1"
&

There are three suspected root causes:

- "The target object in cdg-send-event API does not exist" on SYMCURE-APPLICATION-1.
- "There is no event corresponding to the event name in the cdg-send-event API" on SYMCURE-APPLICATION-1.
- "Event type for corresponding generic event send is unspecified" on SYMCURE-APPLICATION-1.

There are seven possible combinations of these three faults; either each by itself (three possibilities), all of them (one possibility), or any pair (three possibilities).

To determine which of these are minimal candidates, you call the cdgq-get-minimal-root-causes procedure, which produces the following results:



```
#530   4:41:13 p.m.   Minimal Candidate:

#531   4:41:13 p.m.   The target object in cdg-
  send-event API does not exist, SYMCURE-
  APPLICATION-1, suspect, upstream inferred

#532   4:41:13 p.m.   Minimal Candidate:

#533   4:41:13 p.m.   There is no event
  corresponding to the event name in the cdg-
  send-event API, SYMCURE-APPLICATION-1,
  suspect, upstream inferred
```

It should be clear from the specific fault model that "Event type for corresponding generic event send is unspecified" on SYMCURE-APPLICATION-1 cannot be a minimal candidate by itself, because any one of the other two root causes is required to explain "Specific fault model is not built" on SYMCURE-APPLICATION-1.

Either of the following root causes can explain all symptoms:

- "The target object in cdg-send-event API does not exist" on SYMCURE-APPLICATION-1.
- "There is no event corresponding to the event name in the cdg-send-event API" on SYMCURE-APPLICATION-1.

Thus, each of these root causes by itself, constitutes a minimal candidate.

Subclassing SymCure Events and Actions

You can subclass SymCure generic events and actions, as well as their associated display objects. You might want to subclass these objects to provide your own icons and class-specific attributes. In some cases, you must provide implementations of certain methods; in other cases, providing method implementations is optional.

Generic Event Subclasses

You can create a subclass of any of the following generic events:

- cdg-generic-or-and-event
- cdg-generic-and-and-event
- cdg-generic-or-or-event
- cdg-generic-nm-and-event
- cdg-generic-or-nm-event
- cdg-generic-if-and-event

You can create your own icons and add any class-specific attributes.

When subclassing generic events, you must provide implementations of the following methods to determine the associated generic and specific event display objects. In the method signatures described in this section, my-cdg-generic-or-and-event is an example of a user-defined generic event class.

cdg-get-display-class
(*MyGenericEvent*: class my-cdg-generic-or-and-event)
-> *generic-event-display*: symbol

Returns the symbol for the generic event display class.

cdg-get-specific-display-class
(*MyGenericEvent*: class my-cdg-generic-or-and-event)
-> *specific-event-display*: symbol

Returns the symbol for the specific event display class.

Generic Event Display Subclasses

You can create a subclass of any of the following generic event display objects. SymCure uses generic event display objects when displaying relations between events and external actions, and between events and view nodes.

- cdggrlb-generic-or-and-event-display
- cdggrlb-generic-and-and-event-display

- cdggrlb-generic-or-or-event-display
- cdggrlb-generic-nm-and-event-display
- cdggrlb-generic-or-nm-event-display
- cdggrlb-generic-if-and-event-display

You can create your own icons and add any class-specific attributes.

When subclassing generic event displays, you can optionally provide an implementation of the following method to copy class-specific attributes to the generic event display object. In the method signature, my-cdggrlb-generic-or-and-event-display is an example of a user-defined generic event display class.

cdggrlb-set-display-attributes
(*MyGenericEventDisplay*: class my-cdggrlb-generic-or-and-event-display, *MyGenericEvent*: class my-cdg-generic-or-and-event)

Within this method, you can use call next method to set the name of the event and the target class.

Specific Event Display Subclasses

You can create a subclass of any of the following specific event display objects:

- cdggrlb-specific-or-and-event-display
- cdggrlb-specific-and-and-event-display
- cdggrlb-specific-or-or-event-display
- cdggrlb-specific-nm-and-event-display
- cdggrlb-specific-or-nm-event-display
- cdggrlb-specific-if-and-event-display

You can create your own icons and add any class-specific attributes.

When subclassing specific event displays, you can optionally provide implementations of the following methods to copy class-specific attributes to the specific event display object or to modify the icon colors and text of its icon. In the method signatures, my-cdggrlb-specific-or-and-event-display is an example of a user-defined specific event display class.

cdggrlb-set-display-attributes
(*MySpecificEventDisplay*: class my-cdggrlb-specific-or-and-event-display, *MySpecificEvent*: class my-cdg-specific-or-and-event)

Within this method, you can use call next method to set the name of the event and the target object. If you need access to class-specific attributes from my-cdg-specific-or-and-event, use the cdg-get-generic-event-for-specific-event API to access the generic event.

cdggrlb-set-display-icon
(*MySpecificEventDisplay*: class my-cdggrlb-specific-or-and-event-display,
MySpecificEvent: class my-cdg-specific-or-and-event)

Within this method, you can use call next method only if your icon has the following regions:

- value-region
- value-text

Generic Action Subclasses

You can create a subclass of any of the following generic actions:

- cdg-generic-action
- cdg-generic-test-action
- cdg-generic-repair-action

You can create your own icons and add any class-specific attributes.

When subclassing generic actions, you must provide implementations of the following methods to determine the associated generic and specific action display objects. In the method signatures, my-cdg-generic-action is an example of a user-defined generic action class.

cdg-get-display-class
(*MyGenericAction*: class my-cdg-generic-action)
-> *generic-action-display*: symbol

Returns the symbol for the generic action display class.

cdg-get-specific-display-class
(*MyGenericEvent*: class my-cdg-generic-action)
-> *specific-action-display*: symbol

Return the symbol for the specific action display class.

Generic Action Display Subclasses

You can create a subclass of any of the following generic action display objects. SymCure uses generic action display objects to display relations between external actions and events.

- cdggrlb-generic-action-display
- cdggrlb-generic-test-action-display
- cdggrlb-generic-repair-action-display

You can create your own icons and add any class-specific attributes.

When subclassing generic action displays, you can optionally provide an implementation of the following method to copy any class-specific attributes to the generic action display object. In the method signatures, my-cdggrlb-action-display is an example of a user-defined generic action display class.

> cdggrlb-set-display-attributes
>
> (*MyGenericActionDisplay*: class my-cdggrlb-action-display, *MyGenericAction*: class my-cdg-generic-action)
>
> Within this method, you can use call next method to set the name of the action and the target class.

Specific Action Display Subclasses

You can create a subclass of any of the following specific action display objects:

- cdggrlb-specific-action-display
- cdggrlb-specific-test-action-display
- cdggrlb-specific-repair-action-display

You can create your own icons and add any class-specific attributes.

When subclassing specific action displays, you can optionally provide an implementation of the following method to copy any class-specific attributes to the generic event display object, or to modify the icon colors and text of its icon. In the method signature, my-cdggrlb-specific-action-display is an example of a user-defined specific action display class.

> cdggrlb-set-display-attributes
>
> (*MySpecificActionDisplay*: class my-cdggrlb-specific-action-display, *MySpecificAction*: class my-cdg-specific-action)
>
> Within this method, you can use call next method to set the name of the event and the target object. If you need access to class-specific attributes from my-cdg-specific-or-and-event, use the cdg-get-generic-action-for-specific-action API to access the generic action.

Exporting and Importing Fault Models

Use these API procedures to export and import specific and generic fault models:

cdg-persist-diagram-folder
(*DiagramFolder*: class cdg-diagram-folder, *Filename*: text
-> *xml-text*: text

Exports *DiagramFolder* to *Filename* and returns the XML text of *Filename*. If you do not provide the full path for filename, the g2-default-directory is used for storing the file.

cdg-parse-diagram-folder-xml-document
(*Filename*: text)

Imports diagram folder from *Filename*.

cdg-persist-diagnosis-manager
(*DiagnosisManager*: class cdg-diagnosis-manager, *Filename*: text)
-> *xml-text*: text

Archives *DiagnosisManager* in *Filename*. If you do not provide the full path for filename, the g2-default-directory is used for storing the file.

cdg-persist-diagnosis-managers
(*Directory*: text, *FilenamePrefix*: text, *FilenameSuffix*: text)

Stores all diagnosis managers in *Directory*. The *FilenamePrefix* and *FilenameSuffix* are added before and after the name of each diagnosis manager to generate unique filenames to avoid overwriting existing files. For example, the following API stores diagnosis-manager-1 as my-app-diagnosis-manager-1-275.xml in `C:\Temp`:

cdg-persist-diagnosis-managers("C:\Temp", "my-app-", "-[the current time]")

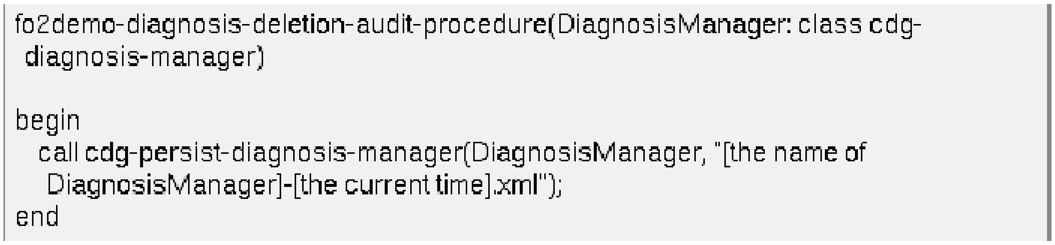
cdg-parse-diagnosis-manager-xml-document
(*Filename*: text)

Imports a diagnosis manager from the XML file specified by *Filename*. The API requires the full path to the file. It re-creates the diagnosis manager, and all associated specific actions and events. It requires the presence of the domain objects that are the targets of specific events in the XML file and the underlying generic fault models. Errors during parsing are posted on the message browser.

You can use the APIs for exporting specific fault models in the audit procedures specified by `config.txt` to automatically archive diagnosis managers and their actions and events.

For instance, to archive a diagnosis manager before it is deleted, create a cdg-audit-diagnosis-deletion-procedure, call the API cdg-persist-diagnosis-manager in this procedure, and assign the configuration parameter CDG-AUDIT-DIAGNOSIS-BEFORE-DELETION-PROCEDURE the name of this procedure.

Here is the text of a cdg-audit-diagnosis-deletion-procedure:



```
fo2demo-diagnosis-deletion-audit-procedure(DiagnosisManager: class cdg-
 diagnosis-manager)

begin
   call cdg-persist-diagnosis-manager(DiagnosisManager, "[the name of
    DiagnosisManager]-[the current time].xml");
end
```

Similarly, you can archive a specific fault model each time the status of a diagnosis manager is updated by using the configuration parameter CDG-AUDIT-DIAGNOSIS-STATUS-PROCEDURE.

Object lookup

cdg-domain-object-lookup
    (*TargetName*: text)
    -> *DomainObject*: item-or-value

Looks up and returns a domain object based on the key *TargetName*. If no such object could be found, the symbol none is returned.

TargetName could be either the:

- Symbolic name of a grtl-domain-object
- opfo-external-name of an opfo-domain-object
- Key of a grtl-object-with-key

Index