Eversholt Fault Tree Description Language

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: Eversholt 1.8
November 2008

081124@21288

Contents

Preface

The *Eversholt Fault Tree Description Language* defines and describes the eversholt language. The eversholt language describes fault trees in the Sun Fault Management Architecture (FMA).

Who Should Use This Book

This document is intended for systems software engineers who are creating fault trees for the Sun fault management system.

Before You Read This Book

Before you read this document, read "Sun Fault Management Architecture I/O Fault Services" in Chapter 13, "Hardening Solaris Drivers" in "Designing Device Drivers for the Solaris Platform" in *Writing Device Drivers*.

For more information, see the Fault Management community site (`http://www.opensolaris.org/os/community/fm/`) on the OpenSolaris web site (`http://www.opensolaris.org/os/`). One document available on the Fault Management community site is the *Fault Management Daemon Programmer's Reference Manual* (FMD PRM).

How This Book Is Organized

This document is organized into the following chapters:

Chapter 1, "Introduction," defines eversholt as the computer language for describing fault trees. This chapter gives an overview of the eversholt language and provides a sample source file that shows many of the common eversholt constructs in context.

Chapter 2, "The Eversholt Language," describes the eversholt language in detail. This chapter describes every construct that is part of the eversholt language, and the syntax and semantics of the construct.

Chapter 3, "Eversholt Compiler," gives a high-level overview of eversholt language features, shows a small sample source file, and then describes the command line interface of the eversholt compiler.

Appendix A, "Eversholt Language Syntax," provides a summary of the eversholt language syntax. This appendix lists the eversholt reserved words and shows the formal grammar.

Comments and Questions

To submit comments on this book or comments or questions on the eversholt language or Sun fault management, use the `fm:discuss` forum on the OpenSolaris web site (`http://www.opensolaris.org/jive/forum.jspa?forumID=49`).

CHAPTER 1
Introduction

The name eversholt refers to all of the following tools:

- The nomenclature and methods for recording and analyzing fault propagation trees
- The language for describing fault trees concisely
- The compiler that compiles that language
- The diagnosis algorithm that performs fault diagnosis based on a fault tree

In this document, eversholt is the computer language for describing fault trees. For clarity, the compiler and diagnosis engine are referred to explicitly.

1.1 Eversholt Language Overview

Figure 1–1 shows the flow between components of a fault management system. This chapter concentrates on the language used in the `.esc` files that are the input to the eversholt compiler. For information on the compiler command, see Chapter 3, "Eversholt Compiler."

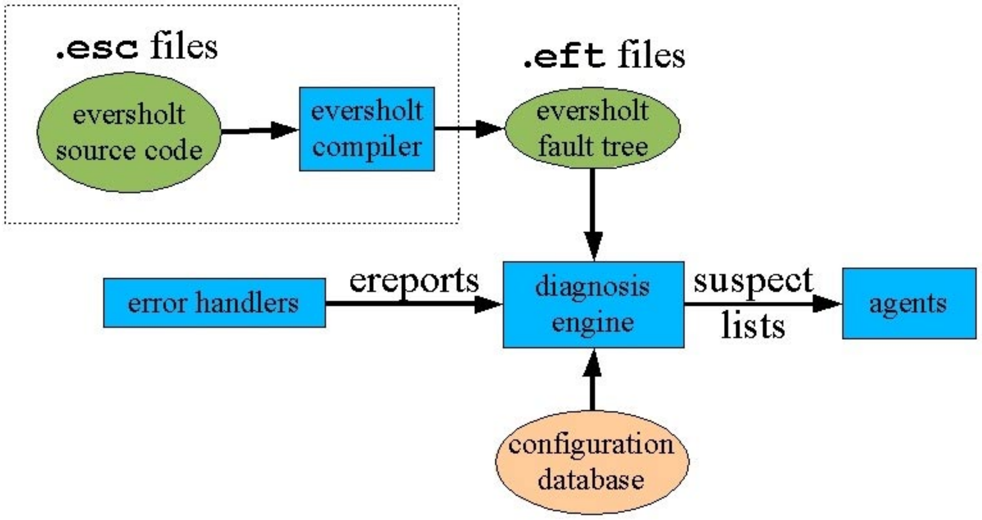

FIGURE 1–1 Eversholt Information Flow

Eversholt is not a general purpose programming language. Eversholt does not include loops or if-then-else statements, and compiling eversholt source does not produce executable code. Instead, eversholt is a language for concisely describing fault propagation. The binary file produced by the eversholt compiler (the `.eft` fault tree file) contains the result of any static analysis the compiler could perform. That binary file drives the eversholt diagnosis engine, telling it how to react to ereports and configuration information at run-time.

1.2 Eversholt Sample Source File

The following sample eversholt source file contains many of the common constructs in the language and serves as a quick example of how to use eversholt.

```
/*
 * sample.esc: Sample eversholt source code
 *
 * The language syntax description here is brief -- more of a
 * reminder of what the language manual has to say about it. Use
 * this file as an eversholt "crib sheet".
 */

/*
 * The common way to embed SCCS information in .esc files
 * (and therefore the resulting .eft files) is to use the
 * following line (tabs shown as \t for readability):
```

```
 * #pragma ident\t"%Z%%M%\t%I%\t%E% SMI"
 */

#pragma ident "@(#)sample.esc 1.1 03/07/31 SMI"

/*
 * Events are written as:
 * <event-class> @ <component-path> { <constraint> }
 * where <event-class> is a dot-separated FMA event class string:
 * fault.something...
 * upset.something...
 * defect.something...
 * error.something...
 * ereport.something...
 * and <component-path> is a slash separated path without instance numbers:
 * somethingA/somethingB...
 * and <constraint> is an expression.
 * When constraint is false, that event is elided from the statement.
 * For declarations, elided events mean the declaration goes away and
 * all uses of that event elsewhere are also elided. For prop statements,
 * elided events mean the list of event gets smaller and if no events
 * remain on the one or both sides of the propagation arrow ->, then
 * the prop statement itself goes away.
 *
 * See examples of events in the following declarations.
 */

/* Declare SERD engines
 * "N" is required, value is integer
 * "T" is required, value is timeval
 * "trip" is optional, value is event
 */
engine serd.fluctuating@acside/pm, N=3, T=1day;

/*
 * Declare faults
 * no required properties
 * "FITrate" is optional, value is an integer
 * "engine" is optional, value is a SERD engine name
 * "count" is optional, value is a STAT engine name
 * "message" is optional, value is 0 or 1
 * "retire" is optional, value is 0 or 1
 * "response" is optional, value is 0 or 1
 */
event fault.power.output_short@acside/pm, FITrate=10;
event fault.power.input_short@acside/pdb, FITrate=10;
event fault.power.fluctuating@acside/pdb, engine=serd.fluctuating@acside/pm;
```

```
/*
 * Declare upsets
 * no required properties
 * "engine" property is optional, value is a SERD engine name
 */
event upset.power.glitch@acside/pm;

/*
 * Declare defects
 * no required properties
 * "engine" property is optional, value is a SERD engine name
 */
event defect.powermgmt.configbug@acside/pm;

/*
 * Declare errors
 * no required properties
 */
event error.power.overcurrent@acside/dcbus;
event error.power.output_overcurrent@acside/pm;

/*
 * Declare ereports
 * no required properties
 * "poller" is optional, value is name of external poller
 * "discard_if_config_unknown" is optional, value is 0 or 1
 */
event ereport.power.output_overcurrent@acside/pm;
event ereport.power.output_undervoltage@acside/pm;
event ereport.power.fluctuating@acside/pm;
event ereport.power.configbug@acside/pm;
event ereport.power.glitch@acside/pm;

/*
 * Propagations
 * <list-of-events> (N)->(K) <list-of-events>
 * default N is 1, default K is 1, so the arrow
 * ->
 * is short for
 * (1)->(1)
 * the letter A may be used to stand for "all".
 *
 * Nothing must propagate to a problem (fault, upset, or defect).
 * Nothing must propagate from an ereport.
 * Errors may propagate to errors, but cycles in the
 * propagation tree are disallowed unless specificly enabled with
 * the allow_cycles pragma.
 *
 * Iterators can be implicit, like "sidenum" and "pmnum":
 * prop error.power.output_overcurrent@acside[sidenum]/pm[pmnum]
 * -> ereport.power.output_undervoltage@acside[sidenum]/pm[pmnum];
 *
 * or they can be implicit, the following statement means the same thing:
 *
 * prop error.power.output_overcurrent@acside/pm
 * -> ereport.power.output_undervoltage@acside/pm;
 *
 * Normally iterators are expanded vertically, so each matching instance
 * implies another complete prop statement. Iterators surrounded by <>
 * are expanded horizontally, so each matching instance expands the list
 * in the current prop statement:
 *
 * prop fault.power.output_short@acside/pm
 * -> error.power.overcurrent@acside/dcbus<>{within(1us)};
 */
prop fault.power.output_short@acside/pm ->
    error.power.overcurrent@acside/dcbus{within(1us)};
prop fault.power.input_short@acside/pdb ->
    error.power.overcurrent@acside/dcbus{within(1us)};
prop error.power.overcurrent@acside/dcbus (A)->
    error.power.output_overcurrent@acside/pm<>;
prop error.power.output_overcurrent@acside/pm ->
    ereport.power.output_overcurrent@acside/pm{within(10ms)};
prop error.power.output_overcurrent@acside/pm ->
    ereport.power.output_undervoltage@acside/pm{within(1ms)};

prop fault.power.fluctuating@acside/pdb ->
    ereport.power.fluctuating@acside/pm<>{within(1us)};

prop defect.powermgmt.configbug@acside/pm ->
    ereport.power.configbug@acside/pm;

prop upset.power.glitch@acside/pm ->
    ereport.power.glitch@acside/pm;
```

If the above file is named `sample.esc`, the eversholt compiler command to produce the binary file required by the diagnosis engine would be:

```
esc -o sample.eft sample.esc
```

Consult Chapter 3, "Eversholt Compiler," for a full description of compiler command-line options.

CHAPTER 2

The Eversholt Language

This chapter describes the eversholt language in detail. This chapter describes every construct that is part of the eversholt language, and the syntax and semantics of the construct.

This chapter describes the following language features:

2.1 Eversholt Language Definitions

Although eversholt is not a general purpose programming language that compiles into executable code, eversholt can still be discussed in terms of *compile-time*, *load-time*, and *run-time*.

compile-time	The conversion of eversholt source `.esc` files into binary fault tree `.eft` files.
load-time	The selection of the appropriate `.eft` files for a specific platform. This selection typically happens at the time the eversholt diagnosis engine begins execution on a platform.
run-time	The eversholt diagnosis engine applying the fault tree in a `.eft` file to a specific platform configuration and timeline of ereports.

2.2 Eversholt Source Files

Eversholt source files typically have the extension .esc. The source for a specific platform might be organized as a single file, as a set of any number of files, or as a hierarchy of #include files. The eversholt compiler takes whatever files are presented and produces a single .eft binary file as output. The fault management architect for a platform (the person who designs the eversholt rules for that platform) decides how many source files to use. The fault management architect decides whether many eversholt source files should be compiled individually (delivering many .eft binary files) or whether the source files should be compiled together (into a single .eft file). The resulting diagnosis engine behavior is the same if the same eversholt rules are loaded (using multiple source files or a single source file).

2.2.1 Preprocessor

All eversholt source files are processed through the C preprocessor, cpp before compilation. This allows directives such as #include and #ifdef to be used as they would be used in a C language file. Remember that all C preprocessor directives must start with a pound sign # in the first column of the line. The same is true for #pragma statements, which are not, strictly speaking, directives to cpp. See "2.9 Eversholt Language Pragmas" on page 37 for more information about #pragma statements.

When eversholt source is processed through cpp, the compiler provides the appropriate options to cpp so that the output contains no predefined symbols or include directories. For example, the following code does not work the same way it works with a typical C compiler:

```
/* Probably NOT what you want! */
#ifdef sparc
      prop error.a -> error.b;
#else
      prop error.a -> error.c;
#endif
```

The reason the above fails in eversholt is that sparc is not defined automatically by cpp as it is with C files. The eversholt compiler does allow you to define cpp variables by using the -D command-line option just as you do with C compilers. Still, constructs such as the above example should be used with great care. The #ifdef is expanded at compile-time, so it might not reflect information about the platform that the eversholt diagnosis engine is running on. The next section describes a potential solution to this issue.

2.2.2 Identifying Platform Eversholt Files

A directory structure similar to `/usr/platform` on Solaris machines is recommended for `.eft` files. Eversholt source files can then be compiled into binary `.eft` files that are grouped by instruction set, platform, and OS type. Typically, the framework surrounding the eversholt diagnosis engine allows it to search for the appropriate `.eft` binaries on start-up. This behavior is analogous to the way device drivers are loaded by the Solaris kernel.

2.3 Eversholt Language Constructs

This section describes each construct of the eversholt language. Since the eversholt language describes fault propagation, the `prop` statement is the most complex eversholt language statement.

2.3.1 Statements

Eversholt statements are always terminated with a semicolon. There are two classes of eversholt statements: declarations (see “2.4 Eversholt Language Declarations” on page 20) and propagation relationships (see “2.5 Eversholt Language Propagation Relationships” on page 26). The order in which the statements appear in the source file is not significant, except that an event must be declared before it is used.

2.3.2 Constants

Eversholt supports three types of constants:

- Numbers (integers)
- Timevals (integers with accompanying time units)
- Strings

Numbers follow the C language syntax, including octal and hexadecimal formats. For example, the numbers 123, 0173, and 0x7b all refer to the same value: decimal 123.

Note – There is no way to specify a floating point value in eversholt.

Timevals are integers followed by a word that defines the time units being used. For example, 1day, 24 hours, and 86400s all refer to the same value: a period of time equal to a full day. The following words are supported as time units:

```
us microsecond microseconds
ms millisecond milliseconds
s second seconds
min mins minute minutes
hour hours
day days
week weeks
month months
year years
```

In addition, the integer `0` (zero) and the word `infinity` are allowed as timevals by themselves. These two timevals are the only timevals that are not a number/name pair. The value `infinity` is intended for use with the `within` statement to provide a way to specify infinite propagation delays. See "2.6.2 Propagation Delay Constraints" on page 36 for more information about `within` statements.

Note – Timeval units are not considered reserved words in the eversholt language. Therefore, timeval units can be used in event names. For example, the component path `board/s/cpu` is allowable even though `s` means seconds when used in a timeval. The special meaning of these words only applies when they follow a number to create a timeval.

Strings are simply arbitrary text surrounded by double quotes.

2.3.3 Expressions

The term *expression* in eversholt is used broadly to mean any of the following list of things. Note that the context of an expression might disallow some of the variations listed below. For example, declarations can define *property=value* pairs where the value is broadly defined as being an expression. However, specific properties often require specific types, such as the `N` and `T` properties on a SERD engine. These `N` and `T` properties are required to be integers; any other type of expression will generate a compile-time error.

Constants
: Constants are valid expressions, and are described in "2.3.2 Constants" on page 15.

Variables or iterators
: The terms *variable* and *iterator* are used interchangeably in this document. A variable applies only to a single semicolon-terminated statement. Any mention of the same variable in another statement refers to a completely unrelated variable whose scope is limited to that statement. Variable names start with a letter and then contain any combination of letters, numbers, or underscores. Variable names look like C variable names.

Functions

The term *function* in eversholt refers more to the syntax than the semantics. Eversholt function calls look like C function calls, but all eversholt functions are built-in functions, defined by the language. The eversholt functions are:

`confprop(`*path*`,` *propname*`)`
: Returns the value of the indicated property of a resource within the current configuration.

`confprop_defined(`*path*`,` *propname*`)`
: Returns true if the indicated property of the resource exists within the current configuration.

`count(`*enginename*`)`
: Returns the current value of the given `stat` engine.

`has_fault(`*path*`,` *class*`)`
: The `has_fault(`*path*`,` *class*`)` function returns true if the resource at address *path* has already been diagnosed with a fault of class *class*.

`is_under(`*path1*`,` *path2*`)`
: Returns true if the second argument represents a component path that is *under* the component path represented by the first argument. That is, when both paths are fully expanded, `is_under()` returns true when the first path exactly matches the beginning of the second path.

`payloadprop(`*propname*`)`
: Returns the value of the indicated property in the payload of an ereport.

`payloadprop_contains(`*propname*`,` *expression*`)`
: Returns true if the given payload property is equal to the given expression. Also returns true if the payload property is an array and has an element equal to the expression.

  You can match a property or array element to a component path in the propagation (see "2.3.4 Events" on page 19). To tell the eversholt compiler that the expression should be interpreted as a component path, use the sub-function `asru()`. For example, the following is true for each component path contained in the property array `resource`.

```
prop a@path -> b@path { payloadprop_contains("resource", asru(path)) };
```

`payloadprop_defined(`*propname*`)`
: Returns true if the given payload name is defined in a received ereport, regardless of the type or value of the payload property.

`setpayloadprop(`*propname*`,` *propval*`)`
: Sets a payload value that will be added to the suspect list if the diagnosis algorithm determines that the associated event has occurred.

  Use the `setpayloadprop()` function only on propagations from fault events. If that fault is in the suspect list, the payload is added to the `hc-specific` area of that fault at the end of the diagnosis. If multiple propagations attempt to use `setpayloadprop()` for propagations from the same fault event, then the behavior is undefined.

`setserdincrement(`*count*`)`
Use `setserdincrement()` only on propagations from fault or upset events that have an associated SERD engine. The `setserdincrement()` function allows the particular instance of the SERD engine to be incremented multiple times as specified by the argument. This allows different weightings to be given to different events that feed into a single SERD engine.

`setserdn(`*count*`)`
Use `setserdn()` only on propagations from fault or upset events that have an associated SERD engine. The `setserdn()` function allows the particular instance of the SERD engine to have its *N* value overridden by the specified value.

`setserdsuffix(`*string*`)`
Use `setserdsuffix()` only on propagations from fault or upset events that have an associated SERD engine. The `setserdsuffix()` function allows the particular instance of the SERD engine to have the specified suffix appended to its name. This allows multiple SERD engines to be associated with the same engine name and component path.

`setserdt(`*timeval*`)`
Use `setserdt()` only on propagations from fault or upset events that have an associated SERD engine. The `setserdt()` function allows the particular instance of the SERD engine to have its *T* value overridden by the specified value.

`within(`*timeval*`)`
Returns propagation delays. See "2.6.2 Propagation Delay Constraints" on page 36.

Boolean expressions
Like the C language, an eversholt expression is false when it evaluates to zero and is true otherwise. Expressions can be constructed with parenthesis and with the operators ==, `!=`, &&, `||`, `!`, and `?`. All of these operators have the same meaning that they have in the C language.

Integer expressions
Constants, variables, and functions that have integer values can be combined using the operators +, `-`, `%`, `div`, `*`, `^`, &, `|`, >>, <<, <, <=, >, and >=. These have the same meaning as in the C language except that `div` is used instead of `/`.

Events and component paths
Events and component paths as described in "2.3.4 Events" on page 19 are valid eversholt expressions, although they typically are not combined with other expressions. For example, adding an integer to an event would make no sense and would produce a compile-time error.

When restrictions exist on what types of expressions are allowed in a context, those restrictions are described in the section of this manual that describes that specific context. For example, the restriction on what types of expressions can be used for the fault event properties appears in "2.4.1.1 Fault Events" on page 22.

2.3.4 Events

Events are made up of several parts, as shown in the figure below. Only the *event class* is required, but most hardware related events contain a *component path* as well. The *constraint* part of an event is not allowed in all the places where an event can appear. See "2.6 Eversholt Language Constraints" on page 35 for more information.

```
fault.power.output_short @ acside/pm { confprop(acside/pm, "ON") == "true" }
|      event class      | |component|  |          constraints             |
|                       | |  path   |  |                                  |
```

Note – The "at sign" `@` is not used anywhere else in the eversholt language. The `@` symbol always separates an event class name from a component path name. Likewise, the curly braces `{}` always surround constraints and are not used for anything else in eversholt.

When an event is defined in eversholt, it always has an event class. It might or might not have a component path. For example, an event that represents a fatal software bug in the `sendmail` program might be named:

```
defect.sendmail.fatal
```

In a system where only a single `sendmail` program can be running at any given time, the above definition does not require a component path because it is clear which `sendmail` is meant. But in a system where many `sendmail` programs can be running, more information is necessary to qualify which event is meant:

```
defect.sendmail.fatal@sendmail_daemon
```

The idea behind the above example is that eversholt will fill in the instance number after `sendmail_daemon` when diagnosing that defect.

In practice, eversholt events almost always have both event class names and component path names. Since the event declaration contains both names, it follows that eversholt considers two declarations with the same event class and different component paths as different declarations. Likewise, two declarations with different event class names but the same component paths are also considered different declarations. In this context, different declarations means the events are unrelated as far as eversholt in concerned. The events can be used in `prop` statements as if they are completely separate events.

2.3.4.1 Event Class

The *event class* is a name containing dots that join the most general class name (leftmost) to more and more specific class names (moving left to right). These names are described in the Events Registry OpenSolaris project.

The top-level (leftmost) class name is the only component interpreted by eversholt. Eversholt requires the top-level class to be one of the following names:

```
fault
upset
defect
error
ereport
```

The definition of each of the above top-level classes and the properties eversholt associates with each one are described in "2.4.1 `event` Statements" on page 21.

Note – An event class is the name of an event, not the name of any hardware or software components. Event class names do not contain slashes `/` or instance numbers as component paths do.

2.3.4.2 Component Path

The *component path* is a slash-separated path similar to the *device tree* in the Solaris OS, except that instance numbers are omitted in eversholt. This allows eversholt to be configuration independent. The instance numbers are filled in only when the eversholt diagnosis engine needs them, and they can be gleaned from the machine configuration at the time of the fault management exercise. For example, a component path representing a CPU on a "system board" might look like this:

```
sb/cpu
```

In some contexts, components might contain iterators, such as in the following example:

```
sb[sbn]/cpu[cpun] /* vertical iterators */
sb<sbn>/cpu<cpun> /* horizontal iterators */
```

See "2.5.1 `prop` Statements" on page 26 for more information on iterators.

2.4 Eversholt Language Declarations

All eversholt *declarations* follow the same general pattern:

reserved-word name-being-declared `[` *property* `=` *value* `[,` *property* `=` *value...*`]]` `;`

reserved-word	Either `event` or `engine`.
name-being-declared	An event.

property/value pairs	After the name, the syntax allows for an arbitrary number of *property*=*value* pairs. Each type of declaration has a specific list of required properties and a specific list of optional properties, as described in the subsections below.

2.4.1 `event` Statements

An `event` statement declares an event. Declaring an event serves three purposes in eversholt:

1. Adding to the list of known events

   Declaring an event allows that event to be used in other eversholt statements. Using an event before it is declared is not allowed, but the declaration can be repeated later in the source file to add additional properties to the event (see "Associating properties with events"). This allows event names to be declared through `#include` files, enables checking for typographical errors in event names, but still allows eversholt source files to add properties to events.
2. Associating properties with events

   The name/value pairs in an event statement allow properties to be associated with an event. Property names cannot be arbitrary strings. Each type of event requires certain properties to be defined and allows certain optional properties. The list of required and optional properties appears in the description of each type of event in the subsections below.
3. Associating global constraints with events

   An `event` statement must name an event, but the constraint part of the event is optional. Currently the only global constraint supported is the `within()` function. When a constraint appears, it is evaluated at run time during an eversholt fault management exercise as if that constraint were typed every place the event was typed. In this way, a constraint specified in the event declaration applies *globally*. When the constraint evaluates to false (see "2.6 Eversholt Language Constraints" on page 35), every place that event is mentioned is elided, as if the declaration and all uses of that event never happened. This is distinctly different from when a constraint appears on an event in a `prop` statement. In those cases, the constraint is only used to elide the event from that statement.

When declaring an event, the top-level event class must be recognized by eversholt. See "2.3.4.1 Event Class" on page 19. There are three types of events:

Problems	There are three types of problem events:	
	Fault	Something is broken
	Upset	Cause of a soft error
	Defect	Design fault
Errors	A signal or datum that is wrong	

Error reports	The detection of an error by an error detector

The top-level event classes corresponding to the above events types are:

```
fault
upset
defect
error
ereport
```

All event declarations must use one of these top-level events.

```
event fault.fan.dead@tray/fan, /* allowed */
      FITrate=500;

event alarm.fan.dead@tray/fan, /* NOT allowed */
      FITrate=500;
```

The type of event also determines how it can be used in `prop` and `mask` statements. Nothing is allowed to propagate to a problem, and nothing is allowed to propagate from an error report.

2.4.1.1 Fault Events

Fault event declarations consist of the reserved word `event`, followed by an event class that begins with `fault`, followed by any component path and constraint information in the syntax described in "2.3.4 Events" on page 19, followed by a *property*=*value* list. The following properties are used with fault event declarations. Note that the names of the properties are case-sensitive.

TABLE 2–1 Fault Event Properties

Property	Required or Optional	Allowed Types
`FITrate`	Optional	Integer, Function
`engine`	Optional	SERD engine name
`retire`	Optional	Integer
`response`	Optional	Integer
`count`	Optional	`stat` engine name
`message`	Optional	Integer

`message`	When given the value zero, tells the run-time environment that the fault does not need to appear in messages to the system administrator. For example, on the

	Solaris OS, setting `message=0` hides the fault in the default output of `fmdump` or `fmadm faulty` and causes it to only show up under `fmdump -a`.
`count`	Specifies a `stat` engine name to increment when the corresponding fault is diagnosed.
`engine`	Specifies a SERD engine name to use if the fault is diagnosed as a valid suspect. If the SERD engine trips, then the fault is added to the suspect list. If the SERD engine does not trip, then the fault is discarded.
`retire`	Setting `retire=0` tells any retire agents that the suspect should not be retired or isolated ecause of this fault.
`response`	Setting `response=0` tells any FRU-based agents that responses such as lighting fault LEDs should not be carried out because of this fault.

Since a fault event defines a problem, nothing is allowed to propagate to a fault event. As described in "2.6 Eversholt Language Constraints" on page 35, using a constraint on a fault event declaration causes that constraint to apply to that event every time that event is used in a `prop` statement.

```
/* Example fault event declaration */
event fault.cpu.ultrasparcIII.overtemp@sb/cpu,
      FITrate=20;
```

2.4.1.2 Upset Events

Upset event declarations consist of the reserved word `event`, followed by an event class that begins with `upset.`, followed by any component path and constraint information in the syntax described in "2.3.4 Events" on page 19, followed by a *property*=*value* list. The following properties are used with upset event declarations. Note that the names of the properties are case-sensitive.

TABLE 2–2 Upset Event Properties

Property	Required or Optional	Allowed Types
`engine`	Optional	Previously declared engine name

If present, the `engine` property specifies a SERD engine name to use if the upset is diagnosed as a valid suspect. If the SERD engine trips, then any trip ereport defined by the SERD engine is generated. Whether or not there is an `engine` property, the upset is then discarded and not added to the suspect list.

Since an upset event defines a problem, nothing is allowed to propagate to an upset event. As described in "2.6 Eversholt Language Constraints" on page 35, using a constraint on an upset event declaration causes that constraint to apply to that event every time that event is used in a `prop` statement.

```
/* Example upset event declaration */
event upset.mem.bitflip@sb/dimm/chip,
      engine=serd.mem.bitflip@sb/dimm/chip;
```

2.4.1.3 Defect Events

Defect event declarations consist of the reserved word `event`, followed by an event class that begins with `defect.`, followed by any component path and constraint information in the syntax described in "2.3.4 Events" on page 19. No properties are defined for defect events.

Since a defect event defines a problem, nothing is allowed to propagate to a defect event. As described in "2.6 Eversholt Language Constraints" on page 35, using a constraint on a defect event declaration causes that constraint to apply to that event every time that event is used in a `prop` statement.

```
/* Example defect event declaration */
event defect.OS.datacorruption@os;
```

2.4.1.4 Error Events

Error event declarations consist of the reserved word `event`, followed by an event class that begins with `error.`, followed by any component path and constraint information in the syntax described in "2.3.4 Events" on page 19

As described in "2.6 Eversholt Language Constraints" on page 35, using a constraint on an error event declaration causes that constraint to apply to that event every time that event is used in a `prop` statement.

```
/* Example error event declaration */
event error.power.overcurrent@acside/pm;
```

Note – Error events in eversholt refer to the event that a signal or datum is wrong, not the detection by an error detector. Detection by an error detector is known as an ereport event.

2.4.1.5 Error Report Events

Error report event declarations consist of the reserved word `event`, followed by an event class that begins with `ereport.`, followed by any component path and constraint information in the syntax described in "2.3.4 Events" on page 19, followed by a *property*=*value* list. The following properties are used with error report event declarations. Note that the names of the properties are case-sensitive.

TABLE 2–3 Error Report Event Properties

Property	Required or Optional	Allowed Types
`discard_if_config_unknown`	Optional	Integer

If the `discard_if_config_unknown` property is set to the value 1, this tells the system to ignore any ereports of that class whose path does not match an existing path in the configuration database (instead of reporting this as an "undiagnosable" defect).

As described in "2.6 Eversholt Language Constraints" on page 35, using a constraint on an error report event declaration causes that constraint to apply to that event every time that event is used in a `prop` statement.

```
/* Example error report event declaration */
event ereport.cpu.ultrasparcIII.ce@sb/cpu;
```

2.4.2 `engine` Statements

Engine declarations consist of the reserved word `engine`, followed by an engine name, followed by a *property*=*value* list. Engine names look very much like event names. They consist of a dot-separated engine class name, which must have a top-level name that is recognized by eversholt. The following subsections describe the engine types recognized by eversholt. The engine name also can contain a component path, appended to the engine class name with an at sign `@` similar to the way events are formed.

2.4.2.1 SERD Engines

SERD (Soft Error Rate Discrimination) is represented by the top-level engine class name `serd`. The following properties are used with SERD engine declarations. Note that the names of the properties are case-sensitive.

TABLE 2–4 SERD Engine Properties

Property	Required or Optional	Allowed Types
`N`	Required	Integer, Function
`T`	Required	Timeval, Function
`trip`	Optional	Event

The SERD algorithm is a thresholding algorithm that uses the above properties to decide when to "trip" and issue the specified event. The properties `N` and `T` describe the threshold as a density of events in time (`N` events within time `T`). The `trip` property specifies which event to issue when the SERD engine detects that the events exceed the threshold.

```
/* Example SERD engine declaration */
engine serd.cpu.ultrasparcIII.ce@sb/cpu,
      N=10,
      T=12 hours,
      trip=ereport.cpu.ultrasparcIII.too_many_ce@sb/cpu;
```

2.4.2.2 stat Engines

A simple engine for counting events is represented by the top-level engine class name `stat`. No properties are defined for `stat` engine declarations. A `stat` engine is incremented when a fault is diagnosed whose declaration contains a `count` property that references the `stat` engine. See "2.4.1.1 Fault Events" on page 22. The current value of a `stat` engine can be used in constraints on propagations using the `count()` function. See "2.3.3 Expressions" on page 16.

```
/* Example stat engine declaration */
engine stat.page_fault@dimm;
```

2.5 Eversholt Language Propagation Relationships

As the name implies, a `prop` statement defines a relationship where the events on the left side of the arrow (`->`) can cause the events on the right side of the arrow.

2.5.1 `prop` Statements

2.5.1.1 `prop` Statement Syntax

The syntax of a `prop` statement is built around the arrow `->`, which shows cause and effect. Events on the left side of the arrow cause events on the right side of the arrow as shown below.

```
prop fault.something -> error.something ;
     |   cause     |    |   effect    |
     | event list  |    | event list  |
```

2.5.1.2 Showing and Translating Fault Trees

The eversholt technology includes a nomenclature for depicting fault trees graphically for illustration or white-board discussion. The eversholt language provides a concise way to translate those pictures into machine-readable form. Figure 2–1 shows a trivial fault tree with a single fault, error, and ereport in a straightforward propagation relationship.

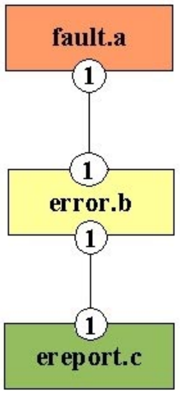

FIGURE 2–1 Trivial Fault Tree

You have two ways to translate the fault tree in the above figure to the eversholt language. The first method is to use the following two `prop` statements.

```
prop fault.a -> error.b;
prop error.b -> ereport.c;
```

The second method to express Figure 2–1 in eversholt is to use a *cascading* `prop` statement:

```
prop fault.a -> error.b -> ereport.c;
```

The above two examples are functionally equivalent. Using separate `prop` statements or a single cascading `prop` statement produces the exact same result. In some cases, the above two methods do not produce the same result because of the way iterator scoping works. Iterator scoping is discussed in detail later in this section.

As the picture of the fault tree implies, nothing is expected to propagate *to* a fault. The same is true for an upset or a defect. Similarly, nothing is expected to propagate *from* an ereport. These restrictions are enforced by the eversholt compiler as described in "2.4.1 `event` Statements" on page 21.

2.5.1.3 `prop` Statement Component Paths

This section discusses component names, including implicit and explicit iterators, vertical and horizontal expansion, explicit instance numbers, and explicit iterator names.

Same Component Paths

The diagram below shows a `prop` statement where the event on the left side and the event on the right side have the same component paths.

```
prop fault.a@x/y -> error.b@x/y ;
     |  cause   |   |  effect  |
     |event list|   |event list|
```

Component paths do not contain instance numbers in eversholt. This enables the propagation rules to be configuration independent. At run-time, the eversholt diagnosis engine detects identical component names on the left side and right side of a `prop` statement such as the one shown above and matches up their instance numbers. This inference process also occurs when only the first part of the path matches. Thus if the left side contains `x/y/z` and the right side contains `x/foo`, eversholt matches the instance numbers associated with `x` as those words match, then continues to expand the `y/z` and `foo` parts. Note that the match must be exact, including any explicit iterator names, so `x[i]/y[j]` does not match `x[k]/y[l]`, nor does it match `x/y`. This inference process usually achieves what the eversholt writer wants, but more complex relationships can be specified using explicit iterators.

Explicit Iterators

The diagram below shows a `prop` statement with explicit iterators.

```
prop fault.a@x[xnum]/y[ynum] -> error.b@x[xnum]/y[ynum] ;
     |       cause         |    |        effect       |
     |     event list      |    |      event list     |
```

Since the iterators in the above diagram match up between the left side and the right side, the above statement is functionally equivalent to the statement shown in "Same Component Paths" on page 27, which used implicit iterators instead of explicit iterators. When the iterators do not match up, you have the situation shown in the following section.

Vertical Expansion

```
prop fault.a@x/y[iterleft] -> error.b@x/y[iterright] ;
     |      cause        |    |       effect       |
     |    event list     |    |     event list     |
```

The `prop` statement in the above diagram shows a propagation relationship between `fault.a@x/y` and `error.b@x/y` where the instance number for `x` must match but the instance number for `y` does not need to match. The fact that the iterators `iterleft` and `iterright` are different words means they will not be matched up by eversholt at run-time. The result will be the cross product of all instance numbers for `y` found in the current configuration, where the `x` instance numbers match up. For example, assume the following components are in the configuration database:

```
x0/y0
x0/y1
x1/y0
```

In this example, the run-time expansion of the preceding diagram would result in the following propagation relationships:

```
fault.a@x0/y0 -> error.b@x0/y0
fault.a@x0/y0 -> error.b@x0/y1
fault.a@x0/y1 -> error.b@x0/y0
fault.a@x0/y1 -> error.b@x0/y1
fault.a@x1/y0 -> error.b@x1/y0
```

The above expansion is called *vertical expansion* because propagation relationships are expanded vertically as if individual `prop` statements were added for each matching instance in the configuration database. Vertical expansion is like copying the eversholt `prop` statement and replicating it once for each match in the current configuration. Iterators surrounded by the square brackets `[]` denote vertical expansion iterators. The implicit iterators shown in "Same Component Paths" on page 27 also denote vertical expansion.

Horizontal Expansion

Sometimes it is useful to expand an event into a list based on the current configuration. This is called *horizontal expansion*. Instead of replicating the entire `prop` statement for each match in the configuration, a horizontally-expanded event is turned into a list of events that match, which is inserted back into the same `prop` statement. The following diagram shows a `prop` statement with both vertical and horizontal expansion.

```
prop fault.a@x[0]/y[0] -> error.b@x<>/y<> ;
     |   cause     |    |    effect     |
     | event list  |    |  event list   |
```

Explicit Instance Numbers

The left side of the diagram in "Horizontal Expansion" on page 29 illustrates another feature of the eversholt syntax: *explicit instance numbers*. Since the square brackets contain integers instead of iterator names, the eversholt diagnosis engine will only match the component path to instance numbers in the configuration database with matching instance numbers. Assume the following components are in the configuration database:

```
x0/y0
x0/y1
x1/y0
```

The run-time expansion of the diagram in "Horizontal Expansion" on page 29 would lead to the following propagation relationships:

```
fault.a@x0/y0 -> error.b@x0/y0, error.b@x0/y1, error.b@x1/y0
```

Note that the vertical expansion in the above example only resulted in a single `prop` statement because the explicit instance numbers on the left side only matched `x0/y0` in the configuration. The angle brackets <> on the right side of the diagram in “Horizontal Expansion” on page 29 expanded to all three lines in the configuration database. Since angle brackets mean horizontal expansion, the result was a list of events.

Finally, note that the angle brackets in the diagram in “Horizontal Expansion” on page 29 did not contain iterator names. Explicit names are allowed, but there was no use for them in this example since the iterator names were not needed elsewhere in the statement. Even with implicit iterator names, the empty angle brackets are required to show horizontal expansion is desired (vertical expansion is the default).

2.5.1.4 Cause and Effect Bubbles

The following figure shows a single propagation graphically. The “bubbles” drawn on the events are referred to as cause and effect bubbles. Bubbles drawn on the bottom of the events are referred to as *cause bubbles*. The bubbles drawn on the top of the events are referred to as *effect bubbles*. The numbers in the cause and effect bubbles are referred to as N and K as shown in the following figure.

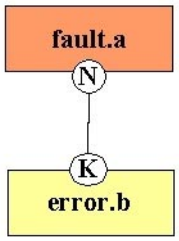

FIGURE 2–2 N and K Values

The N and K numbers have the following meaning:

N	Propagate to at least N events.
K	Propagate from at least K events.

As the following example shows, the N and K values are written in the eversholt language at the head and tail of the arrow, just as they would be in the graphic representation of the tree.

```
prop fault.power.output_short@acside/pm<> (0)->(1) error.power.overcurrent@acside/dcbus;
     |         cause event list         | (N)  (K) |       effect event list        |
```

The parenthesis `()` are required around the N and K values. The default value of 1 is assumed when an N or K value is omitted.

2.5.1.5 Propagation Relationships Constraints

Propagation relationships often have constraints associated with them, so that the propagation is only possible when the constraints evaluate as true. Constraints are a very powerful part of eversholt and are described further in "2.6 Eversholt Language Constraints" on page 35.

The most common constraint used in writing eversholt is the `within()` clause, which is used to specify a propagation delay. This constraint typically appears on events on the right side of a `prop` statement, as shown in the following example.

```
prop error.power.output_overcurrent@acside/pm
-> ereport.power.output_overcurrent@acside/pm{within(10ms)};
```

See "2.6.2 Propagation Delay Constraints" on page 36 for details on the `within()` constraint.

2.5.1.6 Some Common Fault Tree Patterns

A common method for writing eversholt is to first draw the fault tree graphically and then translate that drawing into the eversholt language. As seen in Figure 2–1, this translation can be fairly straightforward. The next few examples show how to translate some other fault tree patterns.

The following figure shows a fault tree where N is greater than 1.

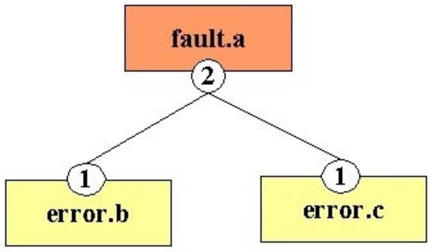

FIGURE 2–3 N Greater Than 1

The following eversholt code represents the above figure.

```
prop fault.a (2)-> error.b, error.c;
```

The following eversholt code is another way to represent the above figure. In this example, the letter `A` is a special expression used to mean *all* events.

```
prop fault.a (A)-> error.b, error.c;
```

The following figure shows a fault tree where K is greater than 1.

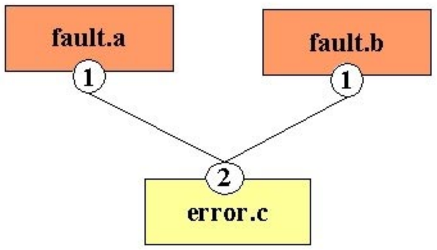

FIGURE 2–4 K Greater Than 1

The following eversholt code represents the above figure.

```
prop fault.a, fault.b ->(2) error.c;
```

The following eversholt code is another way to represent the above figure. In this example, the letter `A` is a special expression used to mean *all* events.

```
prop fault.a, fault.b ->(A) error.c;
```

When both N and K are greater than 1 the graphical representation becomes a bit more complex, as shown in the following figure.

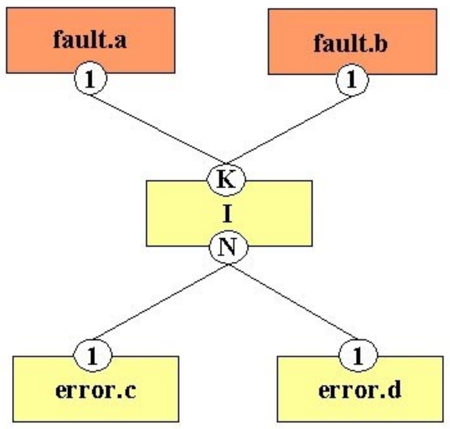

FIGURE 2–5 N and K Greater Than 1

Note the need for a new, intermediate event I in the above figure. The following eversholt code represents the above figure.

```
prop fault.a, fault.b (N)->(K) error.c, error.d;
```

The eversholt compiler automatically inserts internal intermediate events such as the I event in the above figure as necessary.

The following figure shows a graphical representation that does not translate directly into eversholt without modification.

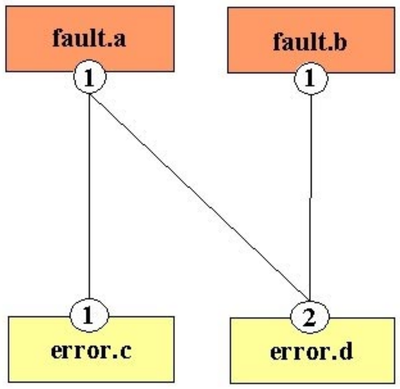

FIGURE 2–6 No Direct Translation

The following eversholt code is *not* a correct translation of the above figure.

```
/* WRONG! No K Value Given... */
prop fault.a -> error.c, error.d;
prop fault.b -> error.d;
```

The above translation is not correct because it does not specify the K value. The list that contains `fault.a` and `fault.b` does not appear anywhere. The solution is to introduce an intermediate event in the eversholt code, as shown in the following figure.

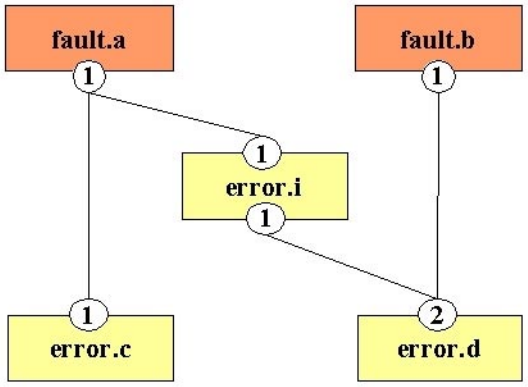

FIGURE 2–7 Using an Intermediate Event

After the insertion of the intermediate event, the tree in Figure 2-15 translates to the following eversholt code:

```
prop fault.a ->error.c, error.i;
prop error.i, fault.b ->(2) error.d;
```

2.6 Eversholt Language Constraints

Constraints are lists of boolean expressions. Constraints follow the name of the event they are constraining and are enclosed in curly braces `{}`.

Constraints can appear in two contexts:

- Events mentioned in declarations
- Events mentioned in `prop` statements

When a constraint is placed on an event mentioned in a declaration, the constraint causes the eversholt diagnosis engine to evaluate that constraint whenever that event is encountered (for example, in any propagation relationships) and elide the event when the evaluation is false. When a constraint is placed on an event mentioned in a `prop` statement, the eversholt diagnosis engine evaluates the constraint for just that particular event in that particular statement, and again it is elided if the constraint evaluates to false. When constraints appear in both contexts (for example, in a declaration and in a `prop` statement), both constraints are applied as if they were joined by a logical AND operator.

Constraints cause `prop` statements to get smaller, since constraints that evaluate to false remove events from those statements. If the resulting statement has no events on either the left or right side of the arrow `->`, the entire statement is ignored.

When a constraint contains a list of expressions, those expressions are logically ANDed together as if each expression were surrounded by parenthesis and connected to the next expression with &&. For example, the following two constraints are identical:

```
{i == j, confprop(x/y, "ON") == "true"}
{i == j && confprop(x/y, "ON") == "true"}
```

2.6.1 Boolean Expression Constraints

Constraints can contain boolean expressions that contain variable names, constants, functions, and the logical operators ==, `!=`, &&, `||`, `!`, `?`, and `:`. For example, the following propagation relationship only holds true when `x` and `y` have different instance numbers:

```
prop error.a@x[xn]/y[yn] -> error.b@x[xn]/y[yn]{xn != yn};
```

2.6.2 Propagation Delay Constraints

The most commonly used constraint is a propagation delay constraint, specified using the `within()` function. This function takes either one or two arguments, both of which must be timevals (see "2.3.2 Constants" on page 15).

In the one-argument form, shown in the following examples, the timeval given specifies the latest time the propagation to the event will occur. Propagation to the event could occur earlier, but it will occur no later than the time specified by the single argument. This argument value is also called the maximum propagation delay.

```
within(10ms);     /* propagates in 10ms or less */
within(5 hours);  /* propagates in 5 hours or less */
within(0);        /* propagates immediately */
within(infinity); /* might never propagate */
```

In the two-argument form of the `within()` function, shown in the following examples, the second argument is the same as in the one-argument form. The first argument specifies the earliest time the propagation to the event will occur. Propagation to the event could occur later, but it will occur no earlier than the time specified by the first argument. This first argument value is also called the minimum propagation delay.

```
within(2ms, 10ms);          /* propagates no earlier than 2ms, no later than 10ms */
within(1 minute, infinity); /* propagates no earlier than 1 minute, maybe never */
```

If the first argument is zero `0`, then the effect of the `within()` function is the same as if the second argument were the only argument. When both arguments are given, the value of the first argument must be less than or equal to the value of the second argument. The `infinity` value can be used only for the second argument.

When no propagation delay constraints are placed on a propagation, zero is assumed, and the event propagates immediately. However, if the N value given (see "2.5.1.4 Cause and Effect Bubbles" on page 30) specifies that propagations to some of the events might happen (rather than they all must happen), the propagation delays are effectively infinite. For example:

```
prop fault.a@x/y (0)-> error.b@x/y;  /* might propagate or might not propagate */
```

2.7 Eversholt Language config Statements

The `config` statement normally is not used in the typical eversholt application as shown in Figure 1–1. Normally, run-time configuration information is obtained from the platform configuration database. The `config` statement enables you to augment the platform configuration database by forcing some component paths permanently into the configuration.

The eversholt diagnosis engine uses the information provided in `config` statements as if that information had come from the platform configuration database.

Note – The `config` statement should be used only in very specific, rare situations. The configuration information provided in a `config` statement will be static and will override anything obtained from the platform configuration database.

The syntax of the `config` statement is:

`config` *instanced-path* `[`*property*=*value* `[,` *property*=*value* `...]] ;`

The *instanced-path* is the name of the component, including instance numbers. After the *instanced-path* is given, any number of properties are defined. One property is defined in the following example:

```
config sb0/cpu0 on=1;
```

2.8 Eversholt Language if Statements

Although the word `if` is a reserved word in eversholt, an "if statement" is not currently defined as part of the language. Typically a constraint as described in "2.3.2 Constants" on page 15 provides the conditionals needed. The word `if` is reserved for potential future use.

2.9 Eversholt Language Pragmas

Pragmas enable or disable specific features. The syntax of a `pragma` statement is:

`#pragma` *option-name*

The pound sign character `#` must be the first character of the line in a `pragma` statement. Although pragmas look like `cpp` directives, the C preprocessor simply passes the line through to the eversholt compiler. Any pragma that is not recognized by the eversholt compiler is ignored. This provides compatibility between newer eversholt source files and older compilers. Be careful to spell the option names correctly so that your pragma is not ignored. Use the `-v` option to the `esc` compile command to confirm whether pragmas are being correctly recognized by the compiler.

The following pragmas are defined:

`#pragma ident` *version-string*

The *version-string* typically is an SCCS or RCS revision string The *version-string* is embedded in the header of the resulting `.eft` (eversholt fault tree) binary file. This allows programs like `strings` or `what` to display the versions of the eversholt source files that went into constructing a particular `.eft` file.

`#pragma new_errors_only`

This pragma tells the compiler that new event declarations from that point of the file onward are expected to be limited to errors only. From this point onward, any `event` statements that declare faults, upsets, defects, or ereports should only be repeating previous declarations or adding properties or constraints to previous declarations, as described in "2.4.1 `event` Statements" on page 21. The definition of completely new fault, upsets, defects, or ereports is not expected and should produce a compile-time warning. This pragma is used to find mismatches between the events used in an eversholt source file and the Event Registry Tool where events are defined.

`#pragma trust_ereports`

This pragma tells the eversholt compiler to set a flag in the resulting `.eft` (eversholt fault tree) binary file telling the eversholt diagnosis engine that any component paths mentioned in ereports exist in the current machine configuration, whether or not those paths are found in the configuration database. This option can be used to avoid the need for configuration database entries for paths that are mentioned in ereports.

Note – There is a very small set of cases where using this pragma makes sense.

`#pragma allow_cycles`

This pragma tells the eversholt compiler that cyclic propagations should result in compile-time warnings instead of fatal compilation errors.

CHAPTER 3

Eversholt Compiler

This chapter gives a high-level overview of eversholt language features, shows a small sample source file, and then describes the command line interface of the eversholt compiler.

3.1 Eversholt Compiler Overview

The `esc` command is the eversholt compiler. Fault trees described by eversholt source code (typically contained in files ending with `.esc`) are compiled into a binary fault tree file that is then used by the eversholt diagnosis engine. The diagnosis engine is a full implementation of the RDA (Reconnoiter, Diagnose, Act) algorithm, which is driven by the compiled eversholt fault tree. Using the `rdasim` simulator, fault trees are verified by simulating incoming error reports and verifying the expected fault diagnoses.

The following list summarizes eversholt features that are described in detail in other chapters in this book:

- Eversholt source files can contain C preprocessor directives.
- Eversholt source files can contain C-style comments: `/* like this */`
- Semicolons are statement terminators.
- Extra whitespace between language tokens is ignored.
- Eversholt provides declarations for faults, errors, ereports, upsets, defects, and engines.
- Eversholt provides `prop` statements for specifying propagations.

You can build the eversholt compiler from source on the OpenSolaris web site. Go to `http://cvs.opensolaris.org/source/`, enter “eversholt” in the Full Search field, select the onnv Project, and click the Search button.

The following code is a sample from an Eversholt source file. This sample declares a single fault, error, and ereport. Then this sample shows the propagation from fault to error, and the propagation from error to ereport.

```
event fault.fuse_open@acside/pm, FITrate=1000;
event error.blackout@acside/pm;
event ereport.blackout@acside/pm;
prop fault.fuse_open@acside/pm -> error.blackout@acside/pm;
prop error.blackout@acside/pm -> ereport.blackout@acside/pm{within(20ms)};
```

Note that ereports correspond to what some other documents call Error Events. That is, an error report is typically packaged into an Error Event, and that in turn is sent to the diagnosis engine via some fault management framework. The diagnosis engine associates that Error Event with the appropriate ereport, and uses the incoming reports to infer a fault diagnosis.

3.2 Eversholt Compiler Command Line Interface

`esc [-YSdghpqvy] [-D`*name*`[=`*def*`]] [-I` *dir*`] [-U`*name*`] [-o` *outfile*`]` *esc-files*`...`

The `esc` command-line interface takes a list of eversholt source files and compiles them into a single internal fault table, as if the files were concatenated into a single file. Since each file is processed through the C preprocessor, `cpp`(1), eversholt source can contain statements that are supported by `cpp` such as `#define` and `#ifdef`. If no `-o` option is given, the compiler simply compiles the input files and exits. You can use this form of the compiler command to check syntax and semantics.

The following command line options are supported by the `esc` command line interface:

`-D` *name*`[=`*def*`]`	This flag is passed on to `cpp`(1). The `-D` option is useful for turning on `#ifdef` sections in eversholt source files so that a single eversholt file can contain platform-specific variations.
`-I` *dir*	This flag is passed on to `cpp`(1). It inserts *dir* into the include file search path.
`-U` *name*	This flag is passed to `cpp`(1).
`-Y`	This flag enables debug output from the compiler's parser.
`-S`	This flag causes the compiler to print various internal usage statistics on exit.
`-d`	This flag enables general debug output from the compiler.
`-g`	This flag causes the compiler to print any automatically-generated iterator names.
`-h`	The `-h` option prints a summary of the command line options. Inserting `-?` or any unknown option also causes the compiler to print a summary of the command line options.
`-p`	This flag causes the compiler to print the entire parse tree.
`-q`	This flag suppresses warnings.

`-v`	This flag enables verbose output.
`-y`	This flag enables debug output from the compiler's lexical engine.
`-o` *outfile*	This option writes the compiled table into a binary file named by *outfile*. The table can be later consumed by a diagnostic engine. The convention is to name the output file with the suffix `.eft` for “eversholt fault tree.”

APPENDIX A

Eversholt Language Syntax

This appendix contains a summary of the eversholt language syntax.

A.1 Eversholt Language Reserved Words

The following words are reserved in eversholt. These words are not allowed as iterator names, event classes, or component paths. These words can be used freely in literal strings and comments.

TABLE A–1 Eversholt Language Reserved Words

Reserved Word	More Information
`config`	"2.7 Eversholt Language config Statements" on page 36
`div`	"2.3.3 Expressions" on page 16
`engine`	"2.4.2 `engine` Statements" on page 25
`event`	"2.4.1 `event` Statements" on page 21
`if`	"2.8 Eversholt Language if Statements" on page 37
`infinity`	"2.3.2 Constants" on page 15
`prop`	"2.5.1 `prop` Statements" on page 26

In addition to the above reserved words, the following *partially reserved* words are defined. These partially reserved words have special meaning only when they follow a number. These words allow the specification of a *timeval*, which is described in "2.3.2 Constants" on page 15. The use of these words in event class names or component path names is allowed but discouraged.

```
day                 min                 second
days                mins                seconds
hour                minute              us
hours               minutes             week
microsecond         month               weeks
microseconds        months              year
millisecond         ms                  years
milliseconds        s
```

Theoretically, the class names of an event can be anything. However, several top-level class names are in common use, and the eversholt compiler expects those class names and assigns specific semantics to them. For example, when an event class name begins with the string `fault`, eversholt might have the property `FITrate` in the declaration. These special semantics tied to class names are described in "2.3.4 Events" on page 19. The following list shows the top-level class names that are recognized by eversholt.

```
defect
ereport
error
fault
upset
```

Any line that starts with the `#` character is either a C preprocessor statement (described in the `cpp` manual) or a `#pragma` statement, which is described in "2.2.1 Preprocessor" on page 14.

A.2 Eversholt Language Grammar

This summary of the eversholt language is intended more for aiding comprehension than as an exact statement of the language. For example, this description does not show a formal definition of an *id* or a *number* since these objects follow common C language practice.

statement:
> `event` *decl-body* `;`
> `engine` *decl-body* `;`
> `prop` *prop-body* `;`
> `config` *config-body* `;`

decl-body:
> `fullevent` *nvpairlist*

`fullevent`:
> *event-class*
> *event-class* `@` *component-path constraint*

```
event-class:
        id
        event-class . event-class

component-path:
        id
        id [ expression ]
        id < expression >
        component-path / component-path

constraint:
        empty
        { expression }

nvpairlist:
        id = expression
        nvpairlist , id = expression

prop-body:
        eventlist nork -> nork eventlist
        prop-body nork -> nork eventlist

eventlist:
        fullevent
        eventlist , eventlist

nork:
        empty
        ( number )
        ( A )

expression:
        id
        fullevent
        component-path
        ( expression )
        expression - expression
        expression + expression
        expression * expression
        expression div expression
        expression % expression
        expression == expression
        expression != expression
        expression && expression
        expression || expression
        ! expression
        expression ? expression : expression
        expression << expression
```

expression `>>` *expression*
expression `<` *expression*
expression `<=` *expression*
expression `>=` *expression*
expression `>` *expression*
expression `^` *expression*
expression `&` *expression*
expression `|` *expression*
function
timeval

function:
id `(` *exprlist* `)`

exprlist:
empty
expression
exprlist `,` *exprlist*

timeval:
number timeunits

timeunits:
`day`
`days`
`hour`
`hours`
…

config-body:
instanced-component-path nvpairlist

instanced-component-path:
id number
instanced-component-path `/` *instanced-component-path*