A Failure Detection and Handling Mechanism for the SR Distributed Programming Language

By

Daniel Tri Huang
B.S. (San Francisco State University), 1986

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

____

____

____

Committee in Charge

1989

CHAPTER 1

Introduction

Reliable software is an integral part of system design. In a sequential programming environment, an executing program may encounter some unexpected conditions such as memory overflow or divide by zero. A program is robust or fault-tolerant if it can recover from such system or software errors. The notion of unexpected conditions may be generalized to include any condition that is beyond the control of a normal execution flow. In this thesis, such unexpected conditions are called *exceptional conditions* or simply *exceptions.*

The issue of reliability is even more critical in distributed software since robustness is one of the major design goals. In a distributed program, different modules of the program can execute concurrently on different machines or nodes; each module has its own address space and communicates with the other modules via messages. Since network or node failures are possible, a failure may sever intermodule communication. A robust distributed program should be able to handle these unexpected underlying system failures and may treat these failures just as other exceptions.

This thesis investigates issues related to the robustness of distributed programs. In particular, it concentrates on language design issues and solutions that support implementation of robust distributed programs. To illustrate the practical realization of these solutions, this work defines a failure detection and handling mechanism—henceforth, the Mechanism—for the SR distributed programming language [Andr88]. The extended SR language with the Mechanism is called SR_e.

1.1. Failure Detection and Handling

A distributed program may fail due to hardware failures, software failures, or communication failures. In a typical execution, one module—the invoker—may activate another module—the server—to perform some tasks; the server may in turn invoke other modules thus continuing the chain of invocation. Since an invoker and its server may reside on different nodes, a failure in the invoker's node or the server's node may only affect modules running on that node; the remaining modules of the program may still be able to continue their execution.

This thesis assumes that distributed programs are executed on top of fail-stop processors [Schl83], i.e., processors that will either perform according to their specifications or simply stop functioning. If a node containing an invoker crashes, or a network partition severs further communication between an invoker and its server, the server becomes an *orphan.* If a node containing a server crashes, or a network partition prevents an invoker from receiving the result from its server, the invocation fails. The primary function of a failure detection and handling mechanism is to locate orphans and failed invocations and to assist in the recovery process.

Primitives provided by a programming language to assist in failure detection and handling of robust programs can be divided into three classes. They are used

- to detect failures and the affected executions;
- to assist reincarnation of modules residing in failed nodes;

- to assist recovery of surviving modules after failures of part of the program.

1.2. Exception Handling Models

The recovery of surviving modules after a crash can be handled by an exception handling mechanism. In this thesis, a *signaling* statement is used to raise an exception explicitly. The construct that immediately contains the signaling statement is called the *signaler* and the construct that services the raised exception is called the *handler.*

Depending on the interaction between a signaler and its handler, exception handling supported by a language can be classified as belonging to a *termination* model, *strict-resumption* model, *resumption* model, or *combination* model [1]. In the termination model, a signaler is terminated after raising an exception. In the strict-resumption model, a signaler must be resumed after the raised exception is handled. In the resumption model, a signaler may be resumed after the raised exception is handled. In the combination model, a signaler can decide which of the other signaling conventions it wants to follow when it raises an exception. Thus a handler may react only passively in the termination model and the strict-resumption model, and has complete control over further execution of the signaler in the resumption model. A handler's response in the combination model may or may not have any effect on the signaler.

---

[1] The terms termination model and resumption model are from CLU [Lisk79]. The terms strict-resumption model and combination model are the special and extended cases, respectively.

1.3. Overview of the Mechanism

The goal of the Mechanism is to provide basic primitives so that programmers may develop further recovery and handling strategies. The Mechanism has two design constraints. First, it should be flexible and incorporate most of the desired features as proposed in earlier work. Second, it should be well integrated into SR, i.e., it should exploit the existing language primitives as much as possible, rather than adding an entirely new set of features to the language. However, the semantics of these exploited primitives must be preserved in the Mechanism.

SR is intended for high-level, distributed system programming and for distributed user applications. The language emphasizes simplicity, expressiveness, and efficiency; yet it provides a rich set of communication primitives. Since the functionality required of failure detection and handling is similar to that already provided by SR’s operation invocation and service, the Mechanism uses SR’s communication primitives as the basis for exception signaling and handling.

Failure detection is performed by SR’s run-time support system. The Mechanism provides a timer based primitive so that an invocation can be associated with a deadline. The Mechanism detects abnormal invocation terminations due to communication or node failures, invocations that are not started or completed within a certain amount of time, and servers that are not invoked or that complete their execution during a specified amount of time. When one of these conditions occurs, the run-time support raises an exception.

The Mechanism does not propose new primitives to assist in the reincarnation of modules resided at failed node; it does, however, provide a rich set of primitives to assist in the recovery of surviving modules after failures. Failure handling is the focus of this research, and is based on the combination model. Exceptions may be parameterized and are compile-time checkable. They may be raised explicitly or implicitly, and synchronously or asynchronously. Besides resuming or terminating the signaler, each handler may re-execute the construct with which it is associated and replace the return value of an enclosing construct.

One novel aspect of using SR communication primitives as the basis for failure handling is that the Mechanism introduces the notion of *asynchronous exception signaling and handling*. This aspect has not been addressed in any current language or proposal.

1.4. Organization of the Thesis

The remaining chapters of this thesis are organized as following. Chapter 2 presents background on language design issues and solutions to support failure detection and handling in distributed software. In particular, it gives an in-depth review of issues relating to the design of exception handling in programming languages. Chapter 3 describes the SR language, which is used as the base language on which the Mechanism is defined. Chapter 4 defines the Mechanism. It shows how the Mechanism is integrated into SR, and describes some features of SR_e. Chapter 5 contains several examples to illustrate how SR_e can be used to implement a robust program. Chapter 6 shows how to transform the Mechanism into standard SR and outlines a possible implementation. Chapter 7 evaluates the Mechanism with the failure detection and handling mechanisms surveyed in Chapter 2, and assesses the SR_e language. The final chapter, Chapter 8, summarizes the work reported in this thesis along with some suggestions for future research.

CHAPTER 2

Background

Various failure detection and handling mechanisms have been proposed in the literature as well as in language implementations. This chapter reviews some of these works and provides the necessary background for the material in later chapters. Exception handling features are also reviewed since they are the basis of the failure handling of this thesis.

2.1. Failure Detection and Handling

In a distributed program, different modules of the program can execute concurrently on different nodes; each module has its own address space and communicates with the other modules via messages. Since an invoker and its server may reside on different nodes, communication failures are possible. The following subsections discuss issues and linguistic designs related to failure detection and handling. Three different linguistic designs are investigated: the event-based model [Schl87], Argus [Lisk83] and Rajdoot [Panz88].

2.1.1. Issues

During a typical invocation, an invoker activates its server to perform some tasks; the server may in turn invoke other servers thus continuing the chain of invocation. A failure in the invoker's node or the server's node affects only modules running on that node; the remaining modules of the program may still be able to continue their execution.

In case of communication failures, the server may still be executing on the invocation while the invoker is unable to obtain the result. Such continuing execution of orphans wastes resources and may interfere with the integrity of other servers. The primary goal of failure detection and handling is to support fault tolerance of invocations, in particular, the detection and elimination of orphans and the graceful recovery of failed invokers.

A language assists failure detection and handling by providing primitives to ensure that

- all servers along the invocation chain are terminated if an ancestor caller terminates abnormally,
- all invocations issued after a failure are serviced only after all related orphans have terminated their execution, and
- all orphans are eventually detected and eliminated.

2.1.2. Event-based Model

Schlichting et al [Schl87] propose that failures can be treated and handled as system events. They suggest a linguistic extension for the SR language to cope with events such as *failures* and *restarts* of processors. A restart of a processor after a failure reestablishes an appropriate execution environment. *Event descriptions* such as **crash**(*machine_denotation*) and **restart**(*machine_denotation*) describe a failure and a restart of the processor denoted by the *machine_denotation* respectively.

Association of a given event with its handler is via a *binding_variable*, which is declared as

var *binding_variable* : **binding**

A value assigned to a *binding_variable* has two components: an event description and a *handler*. For example, the statement

binding_variable := **when** *event_description* **send** *handler* (*arguments*)

specifies that the operation denoted by the handler is to be invoked asynchronously by the extended language’s run-time support when the corresponding event description occurs. The handler and arguments are evaluated when the constructor is evaluated. A program may also revoke the monitoring request by assigning a **null** value to the binding variable.

Since the language’s run-time support begins monitoring an event only after the completion of the *binding_variable* assignment, the proposal provides a predefined *state* function to retrieve the status of a monitored component as of the assignment occurrence. The proposal also allows variables to be declared as *stable* so that their latest values can be restored during a reincarnation of a failed processor. A stable variable is declared as

var *variable_name* : **stable** *variable_type*

2.1.3. Argus

Argus [Lisk83] is a distributed programming language and is intended for transaction processing. Each module written in Argus is called a *guardian* and runs at a single node. A guardian provides its services via *handlers* and is resilient to failures. During the reincarnation of a previously crashed guardian, the language support system re-creates the guardian and restores the values of stable objects. Then a *recovery* procedure, if specified, is executed to re-create other objects that were lost during the crash. A

'x'

recovery procedure has the form

> **recover** *body* **end**

Failure handling in Argus is based on the exception handling mechanism of CLU [Lisk79], which is presented in detail in section 2.2.6. The header of a handler definition must name all possible exceptions. The exception handling is based on the termination model. Argus also contains an orphan detection facility [Walk84, Lisk87] and requires an extensive language run-time support.

2.1.4. Rajdoot

Panzieri and Shrivastava [Panz88] introduce an RPC (Remote Procedure Call) mechanism called Rajdoot that contains fault tolerant capabilities. Rajdoot has the form

> *rpc* (*server*:...; *call*:...; *timeout*:...; *retry*:...; **var** *reply*:...; **var** *rpc_status*:...)

The *timeout* parameter indicates the maximum duration available for execution. When the timer has expired and the server is executing, the server aborts its execution and the invocation terminates abnormally. The *retry* parameter indicates the number of times the invocation is executed before it is aborted.

In order to facilitate orphan elimination, each node maintains a stable counter—*crashcount*—that is incremented immediately after it recovers from a crash. A node also maintains crashcount values of its clients. Each invocation contains the client's crashcount value. If this value is greater than the value stored at the server, then orphans at the server are aborted before the invocation is serviced. Every node has a *terminator* process that occasionally checks the crashcount values of other nodes via messages and aborts any orphans when it detects crashes.

2.2. Exception Handling

Exception handling has been widely studied. However, issues involved in integrating an exception handling mechanism into a language are still debatable. The following subsections discuss exception handling design issues and present the proposals given by Goodenough [Good75], Cocco and Dulli [Cocc82], and Yemini and Berry [Yemi85]. The exception handling mechanism for Ada [Ada83], CLU [Lisk79], LISP [Wile84], and Mesa [Mesa85] are also briefly presented.

2.2.1. Issues

An exception handling design must resolve issues related to *exception declarations*, *signaling exceptions*, and *handling exceptions.* Depending on the solution, additional language primitives may be required and the design can be classified as one of the four models described in Chapter 1. Figure 2.1 summarizes these issues and possible solutions.

2.2.1.1. Exception Declarations

Exceptions can be declared explicitly by programmers or implicitly by languages. They can be parameterized and typed to support the interactions between signalers and handlers. In order to integrate the exception handling mechanism into a base language, the scope rule of exceptions should conform to that of the base language.

Categories	Issues	Possible solutions
Exception Declarations	declaration	explicit implicit
	type	multiple exception types single exception type
	parameterization	permits parameter passing parameterless
Signaling Exceptions	signaling	explicit implicit
	discipline	single-level multilevel
		termination model strict-resumption model resumption model
	concurrency	synchronous asynchronous
Handling Exceptions	association	static with syntactic constructs dynamic with signaler
	placement	control execution flow independence of execution flow
	handler activation	handling only one exception servicing multiple exceptions
	handler responses	terminate signaler resume signaler re-execute an associated construct replace some construct’s result propagate the exception transfer control
	handler deactivation	by a handler by other construct
	default handling	programmable predefined
		propagate same exception convert to default exception

Figure 2.1 — Exception Handling Mechanism Design Issues.

2.2.1.2. Signaling Exceptions

A signaling statement can raise an exception explicitly and the language’s run-time support can raise an exception implicitly. Depending on the circumstances, an exception can be raised according to the termination, strict-resumption, or resumption models. An exception can be raised either directly to its handler or propagated in some hierarchical order—along invocation chain or lexically nested constructs. It follows a multilevel discipline in the former case and a single-level discipline in the latter case.

If the signaler and its handler are located in different nodes, a raised exception may not reach its destination. Signaling an exception is very similar to an invocation, thus synchronous and asynchronous exception signalings are possible and multiple exception signaling can also occur.

2.2.1.3. Handling Exceptions

The placement of handlers can affect the flow of control after an exception is handled. In order to alter the flow of execution, a handler can be associated with a syntactic construct so that termination of the handler’s execution also terminates the execution of the construct. Depending on the interaction with its signaler, a handler may

- terminate the signaler
- resume the signaler
- retry the signaler or some construct
- replace the return value of an invocation or expression
- propagate the exception

- transfer control to another construct.

An unhandled exception may be automatically propagated along a hierarchical path or must be explicitly propagated. The issues of handler activation and deactivation must be addressed if multiple exception signaling and handling are allowed. An exception handling mechanism can provide default handling capability and an override facility to enhance its flexibility.

2.2.2. Goodenough’s Proposal

Goodenough [Good75] defines a set of language primitives to support a combination model for handling exceptions of invocations. Figure 2.2 summarizes Goodenough’s proposal.

2.2.2.1. Exception Declarations

Exceptions are declared with procedure headings and have the general form

[*exception* : *type* **optional**]

where *exception* is parameterless and the *type* of an exception can be *escape*, *signal*, or *notify*. The escape exception supports the termination model, whereas the signal exception supports the resumption model and the notify exception supports the strict-resumption model. The keyword **optional**, if present, denotes that a programmer-defined handler is not required.

Categories	Features	Language Primitives
Exception Declarations	explicit declaration optional handler multiple exception types parameterless	escape, notify, and signal types
Signaling Exceptions	explicit signaling single level discipline combination model	escape, notify, and signal commands
Handling Exceptions	static association with syntactic construct	handler construct
	terminate signaler	escape, exit, and return commands
	resume signaler	resume statement
	terminate enclosing operation	return statement
	propagate the exception	escape statement
	transfer control	exit and return statements
	normal termination handler	ended exception
	cleanup capability	cleanup exception
	default handler	default exception

Figure 2.2 — Exception Handling Mechanism Proposed by Goodenough.

2.2.2.2. Signaling Exceptions

Exceptions can be signaled explicitly via *escape*, *signal*, and *notify* commands, or implicitly by subroutine calls and language-defined operators. A signaling command has the form

command exception_name

where the type of the *exception_name* must correspond with the *command.* Exceptions must be raised explicitly along the invocation chain.

2.2.2.3. Handling Exceptions

Exception handlers are statically associated with syntactic units such as call statement, assignment statement, loop statement, expression, and procedure definition. A handler has the form

[*exception* : *body*]

Execution of a handler may terminate via *exit*, *resume*, *return* or *escape* commands. The exit command terminates the signaler and the syntactic unit to which the handler is attached. The resume command returns control to the subroutine or language-defined operation raising the exception. The return command terminates the handler and returns control from the subroutine containing the handler. The proposal also supports *default handlers* that can be invoked implicitly if no handler is provided or explicitly via **resume** (**default**).

In addition to the above facility, Goodenough's proposal also provides three special exceptions: *ended*, *cleanup*, and *default.* The ended exception provides the ability to define an exception handler for normal termination of an operation. The cleanup exception allows a signaler to clean up before terminating its operation. The default exception permits a signaler to designate a default handler if the exception raised by the signaler is not handled.

2.2.3. Cocco and Dulli's Proposal

Cocco and Dulli [Cocc82] introduce an exception handling mechanism for modular languages. Similar to Goodenough's proposal, the mechanism supports the combination model. Figure 2.3 summarizes Cocco and Dulli's proposal.

Categories	Features	Language Primitives
Exception Declarations	explicit declaration associated with unit multiple exception types parameterized exception	escape, notify, signal, and retry types
Signaling Exceptions	explicit signaling single-level discipline combination model	raise and exit-raise statements
Handling Exceptions	associated with unit	place at end of unit
	group according to type	
	terminate signaler	exit statement
	resume signaler	by default
	retry signaler	restart statement
	terminate enclosing unit	exit statement
	explicit propagation	exit-raise statements

Figure 2.3 — Exception Handling Mechanism proposed by Cocco and Dulli.

2.2.3.1. Exception Declarations

Exceptions and handlers are explicitly declared and associated with a *unit.* A programming unit has the form

’xviii’

```
unit
   {unit external interface}

   except
      exception_names; {Exc}

unit-body

   {local declarations}

   exceptions
      escape_exception_names : escape;
      signal_exception_names : signal;
      retry_exception_names : retry;
      notify_exception_names : notify;

   executable part of the unit;

   escape-handlers
   escape_exception_names : handler_block;

   signal-handlers
   signal_exception_names : handler_block;

   retry-handlers
   retry_exception_names : handler_block;

   notify-handlers
   notify_exception_names : handler_block;

end-unit;
```

Besides the three exception types as proposed by Goodenough, Cocco and Dulli introduce *retry* exceptions. The retry exception reinitializes the operation raising the exception until some condition is met.

2.2.3.2. Signaling Exceptions

Exceptions can be explicitly signaled by a *raise* statement or by an *exit-raised* statement. The raise statement signals an exception for local handling, and the exit-raise statement signals an exception to the invoker. An exit-raised exception handler cannot resume its signaler.

2.2.3.3. Handling Exceptions

Handlers are placed at the end of the corresponding unit and are grouped according to their types. Exceptions can be handled locally within the unit that raises the exceptions or propagated to an external unit. Propagation of an exception first terminates the actual unit and re-raises the exception at another unit.

An *exit* statement and a *restart* statement can be executed within a handler. The exit statement terminates the unit while the restart statement re-executes the unit's body.

2.2.4. Yemini and Berry's Proposal

Yemini and Berry [Yemi85] propose an expression-oriented exception handling mechanism called the *replacement* model that is similar to the resumption model. The mechanism relies on procedure call as the basis for exception signaling and handling. The proposal has been applied to Algol 68 language extended with Ada packages. Figure 2.4 summarizes Yemini and Berry's proposal.

’xx’

Categories	Features	Language Primitives
Exception Declarations	explicit declaration with syntactic construct single exception type parameterized exception	signals clause
Signaling Exceptions	explicit signaling single-level discipline resumption model	procedure call
Handling Exceptions	placement of handlers does not affect execution flow of the attached construct	handler construct
	terminate signaler	replace statement
	resume signaler	by default
	retry operation	procedure call replace statement
	explicit propagation	procedure call replace statement
	transfer control	goto statement

Figure 2.4 — Exception Handling Mechanism proposed by Yemini and Berry.

2.2.4.1. Exception Declarations

Exceptions are declared using a *signals* clause, which can be attached to a procedure declaration or a closed syntactic construct. The signal clause has the form

signals *exception_list*

where each of the exception in the *exception_list* is

(**exc**(*argument_types*)(*resumption_type*,*replacement_type*) *exception_name*)

The *argument_types* specify types of parameters that must be passed with a raised exception. A handler may either return a value of *resumption_type* to resume the execution of its signaler or return a value of *replacement_type* to be used as the final value of the construct to which the signal clause is attached.

2.2.4.2. Signaling Exceptions

Signaling an exception is the same as invoking a procedure. The immediate invoker is responsible for handling exceptions that are declared at a procedure heading.

2.2.4.3. Handling Exceptions

Handlers can be attached to some closed construct and have the form

on *handler_list* **no**

Each handler in the *handler_list* is constructed similarly to a procedure except that it has two return value types instead of only one. The placement of exception handlers is independent of the flow of execution after an exception is handled; it serves only to delimit the scope of the handlers.

A handler in the replacement model can resume the signaler, terminate the signaler, retry some operations, propagate the exception, or transfer control to another construct. A handler can terminate its signaler by returning a replacement type value to the signaler.

2.2.5. Ada

Ada's exception handling mechanism [Ada83] follows the termination model. Figure 2.5 summarizes the exception handling mechanism in Ada.

Categories	Features	Language Primitives
Exception Declarations	user-defined language-defined parameterless single exception type	exception declaration
Signaling Exceptions	explicit signaling single level discipline	raise statement
	implicit signaling	language’s run-time support
Handling Exceptions	terminate the handler	return and raise statement
	explicit propagation	raise statement
	implicit propagation	same exception is raised
	transfer control	return and raise statements

Figure 2.5 — Exception Handling Mechanism in Ada.

2.2.5.1. Exception Declarations

Ada supports both explicit and implicit exception declarations. Exceptions have a single type and are parameterless. They are declared through

exception_list : **exception**;

Implicit exceptions are predefined by the language.

2.2.5.2. Exception Signaling

Exceptions are signaled according to the termination model. A *raise* statement signals an exception explicitly while Ada’s run-time support signals an exception implicitly. A raise statement has the form

raise *exception_name*

If the *exception_name* is omitted, the raise statement propagates the same exception that is being handled to an enclosing frame (see below) or to an invoker.

2.2.5.3. Handling Exceptions

A handler is associated with a frame, which is delimited by **begin** and **end.** Exception handlers are placed at the end of a frame prefixed by **exception** and has one of the following forms

when *exception_names* => *body*
others => *body*

If the *exception_names* match the raised exception, the corresponding *body* is executed. The **others** clause is selected if the raised exception does not match with any **when** clause. A handler terminates via a raise statement or a *return* statement. Termination of a handler also terminates the frame to which the handler is attached.

Exceptions are automatically propagated along the invocation chain until a corresponding handler is located. If a visited frame does not contain a handler, the frame is terminated and the same exception is also raised at the smallest enclosing frame.

2.2.6. CLU

CLU [Lisk79] is a sequential procedure-oriented language. Exception handling in CLU also follows the termination model. Figure 2.6 summarizes the exception handling mechanism in CLU.

Categories	Features	Language Primitives
Exception Declarations	explicit declaration single exception type parameterized exception procedure only	signals clause
Signaling Exceptions	explicit signaling single level discipline termination model	signal and exit statements
handling exceptions	end of handling terminates attached construct	exception statement
	group exceptions with different parameter types in one handler	(*) argument
	explicit propagation	resignal, raise, and exit statements
	implicit propagation	convert to *fail* exception
	transfer control	return, signal, and exit statements
	default handler	others clause

Figure 2.6 — Exception Handling Mechanism in CLU.

2.2.6.1. Exception Declarations

Exceptions are declared in procedure heading and can be parameterized. For example,

> *proc_name* = **proc** (*argument_list*) **returns** (*type*) **signals** (*exception_list*)

specifies an implementation of the procedure *proc_name* may signal one of the exceptions in the *exception_list*.

2.2.6.2. Signaling Exceptions

Exceptions are raised explicitly by a signal statement or by an exit statement. After an exception is raised, the signaler terminates its execution. The signal statement raises an exception to an invoker and has the form

> **signal** *exception_name* (*argument_list*)

The exit statement raises an exception to an enclosing construct of the same procedure and has the form

> **exit** *exception_name* (*argument_list*)

2.2.6.3. Handling Exceptions

Handlers are statically associated with invocations and may be attached only to statements. Handlers are delimited by **except** and **end.** A handler has one of the following forms

> **when** *exception_names* : *body*
> **when** *exception_names* (*argument_list*) : *body*
> **when** *exception_names* (*) : *body*
> **others** (*exception_name* : *string*) : *body*

If one of the *exception_names* matches the signaled exception, the corresponding *body* is executed. The *argument_list* allows a signaler to return parameters to its handler while the "*" notation suppresses the argument list referencing; thus, exceptions that are declared with different parameters can be grouped into one **when** clause. The **others** clause serves as a default handler if none of the exception names matches the raised exception. If the **others** clause is not provided, an uncaught exception is automatically turned into a *failure* exception. A handler terminates when its body returns or signals an exception. Termination of a handler also terminates the construct to which it is attached.

Exceptions must be explicitly propagated along the invocation chain. A *resignal* statement is a short form for a handler that simply re-raises the same exception to another construct up the invocation chain.

2.2.7. LISP

LISP's exception handling mechanism [Wile84] also follows the termination model. Figure 2.7 summarizes the exception handling mechanism in LISP.

2.2.7.1. Exception Declarations

LISP does not support exception declarations.

Categories	Features	Language Primitives
Exception Declarations	not provided	not provided
Signaling Exceptions	multilevel discipline explicit signaling termination model	throw function
Handling Exceptions	terminate the invocation chain	catch function

Figure 2.7 — Exception Handling Mechanism in LISP.

2.2.7.2. Exception Signaling

LISP's exception handling mechanism uses a throw function to support multilevel return of function calls. The throw function has the general form

(**throw** *exp tag*)

where *tag* is optional. If *tag* is specified, the return value of *exp* is delivered to the first **catch** with a matching *tag* (see below) along the invocation chain. Otherwise, the return value of *exp* is delivered to the most recent **catch** that does not specify a *tag*.

2.2.7.3. Handling Exceptions

The catch function has the general form

(**catch** *exp tag*)

where *tag* is optional. If a throw function is not executed during the evaluation of *exp*, the return value of the catch function is the return value of *exp*. Otherwise, the return value of the catch function is the return value of the matching throw function, and other function calls along the invocation chain of *exp* are aborted.

2.2.8. Mesa

Mesa's exception handling mechanism [Mesa85] follows the combination model. It requires many language primitives and an extensive run-time support. Exception signaling and handling is a variation of Mesa's procedure construct. Figure 2.8 summarizes the exception handling mechanism in Mesa.

Categories	Features	Language Primitives
Exception Declarations	explicit declaration associated with frame multiple exception types parameterized exception	signal and error declarations
Signaling Exceptions	explicit signaling single-level discipline combination model	signal, error, and return with error
Handling Exceptions	terminate signaler	unwind statement
	resume signaler	resume statement
	retry associated construct	retry statement
	explicit propagation	reject statement
	transfer control	goto, exit, loop, retry, and continue statements
	explicit deactivation	continue statement
	default handling	any catch
	cleanup capability	*unwind* exception

Figure 2.8 — Exception Handling Mechanism in Mesa.

2.2.8.1. Exception Declarations

Exceptions are called *signals* and are declared in one of the following forms:

exception_name : **signal** *parameter_list* *return_clause*
exception_name : **error** *parameter_list* *return_clause*

where *parameter_list* and *return_clause* are optional. Only programmer-defined exceptions need to be declared. An exception that is declared as **error** is raised by an *ErrorCall* and an exception that is declared as **signal** is raised by a *SignalCall* (see below).

2.2.8.2. Signaling Exceptions

An *ErrorCall* and a *SignalCall* raise an exception explicitly. An *ErrorCall* has one of the following forms

> **return with error** *exception_name argument_list*
> **error** *exception_name argument_list*
> **error**

while a *SignalCall* can be either an *ErrorCall* or has the form

> **signal** *exception_name argument_list*

where the *argument_list* is optional. Thus, the *SignalCall* supports the combination model while the ErrorCall supports the termination model.

2.2.8.3. Handling Exceptions

In Mesa, exceptions are handled by a *catch phrase*, which contains one or more *catches*. A catch has one of the following forms

> *expression_list* => *body*
> **any** => *body*

A catch phrase may appear at the end of an argument list preceded by "!" or be appended to the beginning of a block or a loop statement via an **enable** clause. If the *expression_list* contains more than one exception or if a catch is labeled **any**, the arguments of a signaled exception cannot be accessed in the corresponding body of the catch.

Exceptions are propagated dynamically along the invocation chain. When a catch phrase is visited, one of its catches that matches the raised exception is selected to service the exception. The **any** catch, if specified, is selected if no other catch in the catch phrase

'xxx'

matches the raised exception. If no catch phrase in the routine accepts the signal, the exception is propagated to the next level in the call hierarchy.

A catch phrase can also execute *reject*, *resume*, *retry*, *continue*, and *unwind* statements. The reject statement terminates execution of the catch phrase and propagates the exception to an enclosing catch phrase. The resume statement returns values from a catch phrase to the routine that generated the exception. The retry statement re-executes the statement to which this catch phrase is attached. The continue statement transfers control to the statement immediately following the statement to which the catch phrase is attached. The unwind statement allows each activation record on the dynamic chain to perform cleanup operations (by providing an **unwind** catch) before the catch phrase that issues the unwind statement makes an unconditional jump.

CHAPTER 3

The SR Language

This chapter describes portions of the SR distributed programming language that are directly related to the work of this thesis. The description is primarily extracted from [Andr88] and [Andr89] [1].

3.1. System Components

An SR program is composed from three kinds of separately-compiled *components*: resource specifications, resource bodies, and globals. Resources are the main building block; they are the unit of abstraction and encapsulation. Globals contain declarations of constants and types shared by resources; they have the form

> **global** identifier
>   declarations of constants and types
> **end** identifier

A resource is a parameterized pattern, instances of which are created dynamically. Resources define operations and are implemented by one or more processes that execute in the same address space. Processes interact by means of operations; processes in the same resource may also share variables.

---

[1] [Andr88] defines failure handling features for SR. However, they have not been implemented and are unrelated to the work in this thesis.

3.2. Resources

A resource has a *specification* and a *body.* The specification identifies other components the resource uses and declares operations, constants, and types exported by the resource. The basic form of a resource specification is:

> **resource** identifier
>   **import** identifier list
>   declarations of operations, constants, and types
> **end** identifier

All objects declared in the specification are exported from the resource; they may be used in other resources that import this resource.

The body of a resource contains the processes that implement the resource, declarations of objects shared by those processes, and initialization and finalization code. The general form of a body is:

> **body** identifier(formal parameters)
>   **import** identifier list
>   declarations of shared objects
>   processes
>   **initial** block **end**
>   **final** block **end**
> **end** identifier

A body inherits all objects declared in or imported into the correspondingly named specification. As indicated, the body of a resource is parameterized; actual parameters are assigned during resource creation (see below). The shared objects may be constants, types, variables, and additional, local operations; none of these objects are visible outside the body. A *block* is a sequence of declarations and statements. The specification and body of a resource may be combined when it is not necessary to compile them separately.

3.3. Resource Creation

Instances of resources are created dynamically by means of the **create** statement. Execution of

variable := **create** *resource_identifier*(arguments)

causes a new instance of the named resource to be created. Arguments are passed by value to the new instance and then the resource’s initialization code, if any, is executed (as a process). Execution of **create** terminates when the initialization code terminates. A *resource capability* is returned by **create** and assigned to *variable.* This capability can be used to invoke the operations exported by the resource, can be copied and passed to other resources, and can be used in a **destroy** statement to destroy the instance. The **destroy** statement terminates after the resource’s finalization code (if any) terminates and space allocated to the resource has been freed. An example illustrating resources and the use of the **create** statement appears in section 3.4.6.

By default, execution of **create** places a new resource in the same virtual machine (address space) as that of the resource on which **create** is executed. It is also possible to cause an instance to be placed in a different virtual machine by appending “**on** *vmcap*” to the **create** statement, where *vmcap* is a capability for a virtual machine. New virtual machines are generated by creating instances of the *vm* pseudo-resource in a manner analogous to creating a resource. A virtual machine can be placed on a specific physical machine (network node) by appending “**on** machine_id” to the **create** statement.

In addition to capabilities for resources, which are generated by **create** statements, SR provides capabilities for individual operations, such as one declared in a resource’s specification or local to a process (see below). Capabilities for individual operations provide a mechanism similar to procedural parameters. They are just pointers to operations and are used to support finer-grained control with regard to the resource capabilities over the communication paths between resources. Finally, capability variables and individual fields within capabilities for resources can be set to the special values **null** or **noop**; **null** is the initial value of each capability variable.

3.4. Operations and Communication Primitives

Operations are patterns for actions on objects. They are declared in **op** declarations, which can appear in resource specifications, within resources, or within processes.

Operations are invoked by explicit **call** or **send** statements, or by implicit calls; they are serviced by procedures (**proc**) or input (**in**) statements. In the four possible combinations, these primitives have the following effects:

Invocation	*Service*	*Effect*
call	**proc**	procedure call (possibly remote or recursive)
call	**in**	rendezvous
send	**proc**	dynamic process creation
send	**in**	message passing

3.4.1. Operation Declaration and Capabilities

An operation declaration defines a simple or subscripted operation header. It has the general form

op operation (parameters) **returns** return_value operation_restriction

Every operation must be declared in an **op** declaration before being invoked or implemented.

An operation may have parameters and/or a return value. Parameters are passed by copying, as is the return value. Value parameters (**val**) are copied in, result parameters (**res**) and the return value are copied out, and variable parameters (**var**) are copied both in and out. The default parameter discipline is **val.**

The operation restriction indicates whether the invocation can be by **call**, by **send**, or by either. The default is no restriction.

3.4.2. Invocation Statements

Invocation statements have the forms

call operation(arguments)
send operation(arguments)

where operation is a field of a resource capability, an operation capability, or the name of an operation declared in the scope of the invocation statement. A **call** statement terminates when the operation has been serviced and results have been returned. A **send** statement terminates when the arguments have been stored on the machine on which the resource that services the operation resides. Thus, **call** is *synchronous* whereas **send** is *semi-synchronous* [Bern87].

Execution of an invocation statement causes a run-time error if the denoted operation is a capability with value **null.** In general, execution of an invocation statement has no effect if the denoted operation is a capability with value **noop**; however, if the invocation appears in an expression, the return value is undefined.

3.4.3. Servicing Operations

An operation is serviced either by a **proc** or by one or more **in** statements. A **proc** is a generalization of a procedure: it is declared like a procedure and may be called like a procedure but has the semantics of a process. Its general form is

proc operation_identifier(formal_identifiers) **returns** result_identifier
    block
end operation_identifier

where the operation identifier is the same as that in an **op** declaration in the resource containing the **proc.** Whenever that operation is invoked, an instance of the **proc** is created. This instance executes as a process and uses the formal and result identifiers (if any) to access the arguments of the invocation and to construct the result. If the operation was called, the caller waits for the instance to terminate. However, if the operation was invoked by **send**, the sender and instance of the **proc** execute concurrently; the effect in this case is like forking a process.

The other way to service operations is to employ **in** statements, which have the general form

in operation_command [] ... [] operation_command [] else_command
ni

Each operation command is structurally like a **proc** except it may also contain a synchronization and scheduling expression:

operation_identifier(formal_identifiers) **returns** result_identifier
and synchronization_expression **by** scheduling_expression → block

An **in** statement delays the executing process until some invocation is selectable. An invocation is *selectable* if the boolean-valued synchronization expression in the corresponding operation guard is true. The synchronization expression is optional; if omitted it is implicitly true.

In general, the oldest selectable invocation is serviced. However, if the corresponding operation command contains a scheduling expression, the invocation that is serviced is the oldest one that is selectable and also minimizes the scheduling expression. An invocation is serviced by executing the block in the corresponding operation command, with the formal identifiers bound to the arguments of the invocation. Both synchronization and scheduling expressions can reference invocation arguments since they are in the scope of the formal identifiers in an operation command. Hence selection can be based on invocation arguments in addition to other variables. The optional else command is selected if none of the other arms of the **in** statement is selectable.

An **in** statement terminates when the selected block terminates, or when a **return** statement (see below) is executed within the block. Executing **exit** or **next** also terminates execution of an **in** statement that is within an iterative statement. If the invocation serviced by an **in** statement was called, results are returned to the caller when the **in** statement terminates.

3.4.4. Receive Statement

The **receive** statement has the general form:

receive *operation*(v_1, ..., v_N)

It is a useful abbreviation for an **in** statement that waits for an invocation of *operation* and then assigns the values of the formal parameters to variables v_1, ..., v_N. Together with the **send** form of invocation, **receive** supports asynchronous message passing in a familiar way. Synchronous message passing is supported by **receive** together with the **call** form of invocation.

3.4.5. Invocation Terminations

SR provides two mechanisms to terminate an invocation, **return** and **reply.** Executing a **return** or **reply** statement causes the invocation being serviced by the smallest enclosing operation command or **proc** to terminate, if it is called. Executing **return** also causes the smallest enclosing **in** statement or **proc** to terminate. In contrast, a process that executes **reply** continues executing with the statement following **reply.** Such a process may continue to reference the formals until it leaves their scope; however, no subsequent change to formal parameters or to the return value is reflected back to the caller. A **reply** to an invocation for which a **reply** has already been executed has no effect.

Resource initialization and finalization components may also contain **return** or **reply.** The effects are similar to those for a **proc.** Executing a **return** statement causes the component to terminate early and terminates the corresponding **create** or **destroy** statement. Executing a **reply** statement terminates the corresponding **create** or **destroy** statement, but lets the initialization or finalization component continue executing. Again, executing an extra **reply** has no effect.

3.4.6. Example

To illustrate how resources and operations fit together, consider the bounded buffer resource shown in Figure 3.1. This resource provides two operations: *insert* and *remove*; *insert* places a value in the next available slot if one exists, while *remove* returns the oldest value from the buffer. A calling process is delayed should the buffer be full when an *insert* is attempted or empty when a *remove* is attempted. Note that the resource has a parameter *size*; its value determines the number of slots in the buffer. The use of resource parameters in this way allows instances to be created from the same pattern, yet still vary

```
resource BoundedBuffer
    op insert(item : int)
    op remove() returns item : int
body BoundedBuffer(size : int)
    op bb()
    # create an instance of bb when BoundedBuffer is created.
    # see text for a better way to achieve the same effect.
    initial  send bb()  end
    proc bb()
        var store[0:size−1] : int
        var front, rear, count : int := 0, 0, 0
        do true →
            in insert(item) and count<size →
                    store[rear] := item;  rear := (rear+1)%size;  count++
             [] remove() returns item and count>0 →
                    item := store[front];  front := (front+1)%size; count−−
            ni
        od
    end
end BoundedBuffer
```

Figure 3.1 — A bounded buffer resource.

to a certain degree. Note also that executions of *insert* and *remove* are mutually exclusive since a single **in** statement services both. Finally, the **send** statement in the initialization code creates one instance of the main **proc** *bb.* Resources often contain “worker” processes such as *bb.* SR provides a **process** abbreviation to simplify programming such processes. For example, the above resource could be coded more compactly by deleting the declaration of the *bb* operation, deleting the initialization code, and replacing the line “**proc** *bb*()” by “**process** *bb*”.

Figure 3.2 outlines how another resource, *User*, might create and use an instance of the *BoundedBuffer* resource. As shown, *User* imports *BoundedBuffer.* Its *Main* (background) process declares a resource capability variable, *buf*, that can be used to refer to an instance of *BoundedBuffer. Main* then creates an instance of *BoundedBuffer* with 20 slots and calls the *insert* operation in that instance to place a 33 in the buffer. The interface part of *BoundedBuffer* defines the two operations *insert* and *remove.*

```
resource User
    import BoundedBuffer
body User()
    process Main
        var buf: cap BoundedBuffer
         ...
        buf := create BoundedBuffer(20)
        buf.insert(33)
         ...
    end
end
```

Figure 3.2 — A user of *BoundedBuffer.*

3.5. Concurrent Invocations

One additional communication primitive is the **co** statement, which supports concurrent invocations. The form of a **co** statement is

> **co** concurrent_invocation → post_processing
> // ...
> // concurrent_invocation → post_processing
> **oc**

A concurrent invocation is a **call** or **send** statement, or an assignment statement that invokes a user-defined function. The postprocessing blocks are optional. Execution of **co** first starts all invocations. Then, as each invocation terminates, the corresponding postprocessing block is executed (if there is one); postprocessing blocks are executed one at a time. Execution of **co** terminates when all invocations and postprocessing blocks have terminated or when some postprocessing block executes an **exit** statement. A concurrent invocation in **co** can also be preceded by a quantifier, which specifies multiple invocations, one for each iteration of the implied loop. For example, a replicated file might be updated by executing

> **co** (i := 1 **to** N) **call** *file*[i].*update*(values) **oc**

where *file* is an array of capabilities containing one entry for each file resource.

3.6. Types, Declarations, and Sequential Statements

SR provides a variety of data types and sequential statements similar to those found in other languages. However, their form and many of their details are different, largely to facilitate the integration of the sequential and concurrent components of SR. It also results from the desire to simplify the programming of commonly occurring algorithmic patterns.

In addition to the usual kinds of basic and structured types, SR provides capabilities and what is called an **optype** (operation type). An **optype** defines a pattern for an operation, i.e., the types of the parameters and return value. Such a type is used when the same operation pattern is used more than once, for example, when several resources implement the same type of operation such as file read. What makes capabilities and operation types especially useful is that type-checking is based on structural equivalence; hence an operation capability can be bound to any operation that has the specified pattern.

SR's **if** and **do** statements are based on Dijkstra's guarded commands [Dijk76] mainly so that their structure is similar to that of the **in** statement. However, if no boolean expression is true, **if** has no effect (unlike Dijkstra's **if**, which aborts in this situation). We also allow the last guard of **if**, **do**, and **in** to be **else**, which is interpreted as the conjunction of the negations of the other guards.

SR also provides a novel iterative statement, **fa** (for all), that employs quantifiers introduced earlier with the **co** statement. The form of **fa** is

> **fa** quantifier , ..., quantifier → block **af**

where quantifiers have the general form

> bound_variable := initial_value direction final_value **st** boolean_expression

For example, the single statement

fa *i* := lb(*a*) **to** ub(*a*)−1,
    j := *i*+1 **to** ub(*a*) **st** *a*[*i*]>*a*[*j*] →
        a[*i*] :=: *a*[*j*] # swap statement
af

sorts array *a* into ascending order. This statement employs the builtin array lower-bound ('lb') and upper-bound ('ub') functions. Note how the range of values for the second bound variable, *j*, depends on *i*. Also note the use of **st** (such that), which is used to limit execution of the body of **fa** to those values of *i* and *j* for which *a*[*i*]>*a*[*j*].

The two final sequential statements are **exit** and **next.** The **exit** statement is used to force early termination of the smallest enclosing iterative or **co** statement. The **next** statement is used to force return to the loop control of the smallest enclosing iterative statement; it can also be used within a postprocessing block in a **co** statement to force **co** to wait for another invocation to terminate. Note how these statements, like quantifiers, have consistent uses in support of both sequential and concurrent programming.

CHAPTER 4

The Mechanism and the SR_e Language

This chapter presents the definition of the Mechanism and the extended SR language, SR_e, that includes it. A complete syntactic synopsis of SR_e is presented in the Appendix. The syntax of the Mechanism is defined using a compact variant of BNF. Non-terminals are identifiers set in *italic* type. Keywords are set in **bold** type and delimiters are set in roman type. The symbol '#' is used within productions to separate alternatives. Superscript 'o' attached to a symbol indicates that it is optional. Superscript '+' on a non-terminal indicates that the non-terminal may be repeated one or more times.

4.1. The Mechanism

The failure detection and handling mechanism is based on the concept of *frames.* The Mechanism consists of *failure detection*, *failure recovery*, and *failure handling* components. Failure detection is based on a *timeout* primitive that allows programmers to specify the deadline of a frame and on a heartbeat-based facility. In the heartbeat-based facility, each machine periodically broadcasts its heartbeat messages [Andr88] to other machines so that the states of the process or resource instance can be monitored. Failure recovery is implemented by programmers. Failure handling is built from a general exception handling facility that uses SR communication primitives as the basis of signaling and handling.

Exception handling in SR_e follows the combination model defined in section 1.2, and supports multilevel synchronous and asynchronous signaling. All exceptions must be explicitly declared but need not be handled. Figure 4.1 summarizes the exception handling facility in SR_e.

Categories	Features	Language Primitives
Exception Declarations	explicit declaration	{ op_declaration }
	associated with frame	be statement
	simulate multiple exception types	op restriction
	parameterized exception	op declaration
Signaling Exceptions	explicit signaling multilevel discipline synchronous/asynchronous combination model	invocation statements
	implicit signaling single-level discipline termination model	language’s run-time support
Handling Exceptions	static association control execution flow	be statement
	service multiple exceptions	input statement
	terminate signaler	void statement
	resume signaler	reply and return statements
	retry the operation	restart statement
	terminate enclosing operation	replace and abort statements
	explicit propagation	send and call statements
	implicit propagation	convert to *fail* exception
	terminate frame	leave statement
	terminate handler block	deactivate statement
	cleanup capability	*cleanup* exception

Figure 4.1 — Exception Handling Mechanism in SR_e.

4.1.1. Frame

Each frame is defined by a *be* statement, which has a deadline and an exception handling facility. In this thesis, the terms frame and be statement are used interchangeably. An example of a frame is

```
# simulate nap (5) — sleep 5 units of time, then terminate the frame
var deadline := 5
be { op cleanup (reason : reasons) } timer deadline
   do true -> skip od
handlers
   in cleanup (reason) and reason = timeout -> leave ni
eb
```

4.1.1.1. Be statement

```
be_statement ::= be exception_declaration° timer_clause°
                                  block°
                                  handler_block°
                 eb
```

A be statement is delimited by **be** and **eb.** The *timer clause* specifies a timeout value. The *exception declaration* defines all possible exceptions that may be signaled to the frame and handled by the *handler block* component. The *block* component is an SR block, i.e., it consists of statements, declarations and imports, and may be nested with other be statements.

When control is at a be statement, the timer clause, if specified, is evaluated. If its value is less than or equal to zero, the be statement terminates. Otherwise, the block component is executed. An end of the block component's execution also terminates the frame. Control is then passed to the statement immediately following the be statement. If an exception is signaled during the execution of the block, the exception handling facility is entered (see section 4.1.4).

4.1.1.2. Frame Termination

leave_statement ::= **leave**

A frame terminates when (1) the end of the block component is reached, (2) the block component is suspended and all outstanding exceptions are handled, (3) the handler block component executes a *restart*, *replace*, or *abort* statements (see section 4.1.4.3.3), or (4) a *leave* statement is executed. The leave statement may only appear within a frame. Execution of **leave** terminates the smallest enclosing frame.

Example:

```
var key : int
be
   # obtain key from user until a valid input is entered
   do true ->
      writes ('enter key: ')
      read (key)
      if search (key) → leave
        [] else → write ("invalid key, re-enter.")
      fi
   od
eb
```

4.1.2. Failure Detection

The Mechanism supports failure detection through a timer clause and a *terminator* routine.

4.1.2.1. Timer Clause

timer_clause ::= **timer** expression

A timer clause specifies the maximum elapsed time (deadline) to execute the block component of a frame to which the timer clause is attached. When the deadline has expired, execution of the block component is suspended with reason **timeout.** The deadline is enforced as an approximated real clock time value, and has no effect on the handler block component.

The expression in the timer clause has type **int.** It is evaluated once when the be statement is encountered. An invocation executed within a frame’s block component must be completed within the remaining time to execute the frame. A server—a **proc** or an input statement—must complete the servicing of an invocation within the remaining waiting time of its invoker. The effective deadline of a frame is the minimum value among the timer clause value, the remaining time to execute an enclosing frame or server, and the remaining waiting time of the invoker. When the deadline has expired, the invocation is aborted.

Example:

```
be timer 5
  # foo must be completed within 5 units of time.
  call foo()
eb
```

4.1.2.2. Built-in Function

The Mechanism provides an *avail* function that returns the maximum remaining real clock time to execute the smallest enclosing be statement or server. The avail function is parameterless and has the form

avail ()

It is useful when a client needs to specify the deadline for an invocation or when a server needs to optimize its performance. For example, if the remaining time is less than a given threshold, a server can abort the invocation. The default deadline is infinity.

4.1.2.3. Terminator

The language's run-time support has a *terminator* routine to monitor the state of a program execution. The terminator is based on the heartbeat concept and is executed periodically. When the terminator detects an orphan, it suspends the execution of the orphaned server with reason **orphan.** When the terminator detects a failed invocation, it suspends the failed invoker with reason **fail.** When the terminator detects an orphaned resource, it implicitly executes a destroy statement at the resource. A resource is considered to be an orphan if its capability is no longer held by other resources and no process in the resource is active.

4.1.3. Failure Recovery

The Mechanism does not guarantee automatic recovery of crashed resources. Failure recovery can be implemented by programmers using SR's resource initialization routines. The stable storage facility that might be required by a failure recovery algorithm can be simulated using SR's input/output facility.

4.1.4. Failure Handling

Failure handling is built from a general exception handling facility that relies on SR communication primitives as the basis of signaling and handling. The exception handling facility consists of three parts: exception declarations, signaling exceptions, and handling 'l'

exceptions.

4.1.4.1. Exception Declarations

exception_declaration ::= { operation_declaration$^+$ }

Exceptions are declared similar to SR operations and are statically associated with a frame. There are two kinds of exceptional conditions: *programmer-defined* exceptions and a *language-defined* exception. A frame must explicitly declare an exception if it handles the exception. The scope of an exception is limited to the be statement. A declared exception must be serviced by the handler block component of a frame and not by the block component.

Each resource specification and operation declaration must define all possible exceptions that may be signaled by an instantiation of the resource and by an implementation of the operation, respectively. These exceptions are declared as parameters of the resource specification or operation declaration. An invoker is responsible to provide handler capabilities with the invocation. The matching between handlers and exceptions is via parameter passing.

The only language-defined exception is:

op *cleanup* (*reason* : **reasons**)

where **reasons** is a predefined enumerated type and is defined as

type reasons=enum(**fail**,**orphan**,**outer_orphan**,**timeout**,**outer_timeout**)

Example:

```
# declare two exceptions: cleanup and f
{ op cleanup (reason : reasons)   # language-defined exception
  op f () }                       # programmer-defined exception f
```

4.1.4.2. Signaling Exceptions

signaling_statement ::= invocation_statement

void_statement ::= **void**

Programmer-defined exceptions must be raised explicitly using *signaling* statements, which are just SR invocation statements, i.e., **send** and **call.** Hence, explicitly signaling an exception has the same semantics as invoking an operation. An invocation is identified as signaling an exception if the invoked operation is declared in an exception declaration clause. The *cleanup* exception may be raised explicitly by signaling statements or implicitly by the language's run-time support.

A *void* statement is semantically similar to a reply statement except that no return value is actually sent back to an invoker. Execution of a void statement terminates the invocation abnormally. The invoker continues to be blocked until either (1) its deadline has expired or the terminator is next executed, or (2) its frame is terminated by the handler block. A void statement can be placed in any program segment where a reply statement may appear.

When a server is suspended with reason **orphan**, each frame that is immediately enclosed by the server is suspended with reason **orphan.** All other enclosed frames are suspended with reason **outer_orphan.** When all enclosed frames have terminated, the server implicitly executes a void statement and then terminates its execution. If the server is encapsulated in the block component of a frame, the enclosing frame is suspended with reason **fail**; otherwise, if the server is enclosed by a **proc**, the **proc** implicitly executes a void statement and terminates its execution.

When a frame is suspended with reason **timeout**, each enclosed frame is suspended with reason **outer_timeout.** If the frame does not declare to handle a cleanup exception and is encapsulated in the block component of a frame, the enclosing frame is suspended with reason **fail**; otherwise, if the frame is enclosed by a **proc**, the **proc** implicitly executes a void statement and terminates its execution.

If an invoker is suspended with reason **fail** and is encapsulated in the block component of a frame, the enclosing frame is suspended with reason **fail**; otherwise, if the frame is enclosed by a **proc**, the **proc** implicitly executes a void statement and terminates its execution.

The language's run-time support terminates each suspended frame starting from the innermost frame of the nested block component. If the frame is declared to handle a cleanup exception, the language's run-time support signals a cleanup exception with the suspending reason to the frame. Otherwise, the frame is just terminated and the next enclosing frame is visited. This step is repeated until all suspended frames have terminated.

When a cleanup invocation is implicitly signaled to a frame, the block component aborts its execution. All embedded servers in the block component are terminated as if a void statement were executed. The handler block must terminate the frame.

Example:

```
proc foo ()

  be { op cleanup (reason : reasons) }
     var disaster_exception : cap (r : reasons)
     disaster_exception := cleanup
     in f () →
        be { op cleanup (reason : reasons) }
           call goo ()
        handlers
           in cleanup (reason) and reason = orphan →
                # activated if f is orphaned
            [] cleanup (reason) and reason = outer_orphan →
                # activated if foo is orphaned
            [] cleanup (reason) and reason = fail →
                # activated if goo failed
            [] else →
                # propagate fail to outer frame
                disaster_exception (fail)
           ni
        eb
     ni
  handlers
     in cleanup (reason) →
          # activated if a language-defined exception occurred
          abort
     ni
  eb
end
```

The undeliverable result of an invocation does not raise an exception at the server. If a frame is terminated by the language’s run-time support, all embedded servers are aborted and their invocations are terminated as if a **void** were executed for each invocation (see below).

'liv'

Example 1:

```
be {op f ()}
   # signal a local programmer-defined exception
   f ()
handlers
   in f () -> ... ni
eb
```

Example 2:

```
be {op e ()}
   # invoke foo, passing handler capability e to it
   foo (e)
handlers
   # e is signaled by a server
   in e () -> ... ni
eb
```

Example 3:

```
# foo defines a possible exception exception_e
op foo (exception_e : cap ())

proc foo (exception_e)
   # foo signals exception_e to its invoker
   exception_e ()
end
```

4.1.4.3. Handling Exceptions

Exceptions are serviced by arms of input statements that are placed within the handler block component of a frame. The subsequent subsections present the *handler construct*, *handler activation*, *handler responses*, and *handler deactivation* issues.

4.1.4.3.1. Handler Construct

handler_block ::= **handlers** block

’lv’

A handler block is placed at the end of a frame to service exceptions raised to the frame. The block component of a handler block is constructed similarly to the block component of a frame.

Example:

```
var x,y : int
be {op cleanup (reason : reasons)}
   x := foo ()
   y := 10
handlers
   # x is set to zero if invocation of foo is not completed, then
   # execution is resumed after the be statement; the assignment to
   # y is not executed.
   in cleanup (reason) and reason = fail ->
        x := 0; leave
   ni
eb
```

4.1.4.3.2. Handler Activation

A handler block is activated if (1) its frame declares to handle the raised exception, (2) its frame is not terminated, and (3) it is not activated. A handler block services a raised exception by executing the corresponding arm of an input statement implemented inside the handler block.

4.1.4.3.3. Handler Responses

```
handler_response_statement ::= response_to_signaler_statement  #
                               response_to_frame_statement

response_to_signaler_statement ::= reply_statement  #
                                   return_statement #
                                   void_statement
```

response_to_frame_statement ::= restart_statement #
abort_statement #
replace_statement #
leave_statement #
deactivate_statement

restart_statement ::= **restart**

abort_statement ::= **abort**

replace_statement ::= **replace**

leave_statement ::= **leave**

deactivate_statement ::= **deactivate**

While servicing a raised exception, i.e., within an arm of input statement, a handler block may execute one of the *response to signaler* statements to resume or to terminate the signaler. These statements have the same semantics as SR invocations. *Response to frame* statements may only appear inside a handler block and their executions alter the control flow of the frame to which the handler block is attached.

The restart statement terminates the current instance of the smallest enclosing be statement. All embedded servers in the smallest enclosing frame are aborted and their invocations are terminated as if a **void** were executed for each invocation. Then, control is passed to the beginning of the be statement. Hence the same be statement is re-executed.

The replace statement and the abort statement terminate the invocation of the smallest enclosing server that contains the be statement. Execution of **replace** returns the result to the invoker whereas execution of **abort** does not. All embedded servers in the smallest enclosing frame are aborted and their invocations are terminated as if a **void** were executed for each invocation. Control is then passed to the immediate statement following the be statement.

The leave statement terminates the frame. All embedded servers in the smallest enclosing frame are aborted and their invocations are terminated as if a **void** were executed for each invocation. Then control is passed to the statement immediately following the frame.

The deactivate statement terminates the current instance of the smallest enclosing handler block. All embedded servers in the smallest enclosing handler block are aborted and their invocations are terminated as if a **void** were executed for each invocation. If the frame has an outstanding exception, another instance of the handler block is activated.

Example 1:

```
# sleep 5 units of time, then repeatly sleep 10 units of time
var deadline := 5
be {op cleanup (reason : reasons)} timers deadline
   do true -> skip od
handlers
   in cleanup (reason) and reason = timeout ->
        deadline := 10; restart
   ni
eb
```

Example 2:

```
op foo () returns i:int

proc foo () returns i

   # if f  is signaled, i is set to 10 and returned
   be {op f ()}
      ...
   handlers
      in f () → i := 10; replace  ni
   eb
end
```

Example 3:

```
op foo (num : int; bad_num : cap () returns new_num)

proc test ()

    var x := 5;

    # call foo, if bad_num exception is raised,
    # resume the signaler with new_num = 10
    be {op bad_num () returns new_num : int}
       foo (x,bad_num)
    handlers
       in bad_num () returns new_num -> new_num := 10  ni
    eb
end


proc foo (num,invalid)

     be { op bad_item () }
        # authenticate num, raise bad_item if num < 10
        if num < 10 -> bad_item ()  fi
        # the rest of the block
     handlers
        # num is not valid, raise invalid exception to invoker
        # for a new value, then re-execute the be statement
        in bad_item () -> num := invalid (); restart  ni
     eb
end
```

’lix’

4.1.4.3.4. Handler Deactivation

An executing handler block terminates when (1) a deactivate statement is executed inside the handler block, (2) a leave statement is executed within the frame, or (3) the end of the block or the handler block is reached.

Example:

```
proc foo (num,invalid)

   be { op bad_item () returns new_value : int }
      # authenticate num, raise bad_item if num < 10
      if num < 10 -> num := bad_item ()  fi
      # the rest of the block
   handlers
      # num is not valid, raise invalid exception to invoker
      # for a new value, then resume the block component
      # by executing the rest of the block and deactivate the
      # handler block component
      in bad_item () returns new_value -> new_value := invalid ()
      ni
   eb
end
```

4.2. The SR_e Language

SR_e is a version of the SR language augmented with the Mechanism. This section describes two minor semantic extensions to SR necessitated by incorporating the Mechanism.

4.2.1. Return, Exit, and Next Statements

In addition to terminating the smallest enclosing construct as defined in SR, execution of a return, exit, or next statement also terminates be statements that are embedded within the construct. All embedded servers in these frames terminate as if a **reply** were executed at each server.

’lx’

Example:

```
# repeatly service invocations of foo
do true ->
   in foo () returns x →
        be {op f () }
            # do something
            call f ()
            # the rest of block
        handlers
            # if f exception is raised, terminate foo with
            # x = 10 and go back to service next invocation.
            # The rest of block is not executed
            in f () → x := 10; next  ni
        eb
   ni
od
```

4.2.2. Destroy Statement

The semantics of a destroy statement is augmented as following. Execution of all **proc**s is aborted and each invocation is terminated as if a **void** were executed at the server. Then the finalization routine is activated and executed. Execution of a **destroy** (*myresource* ()) within the body of a finalization routine has no effect, which is similar to its current definition in SR.

CHAPTER 5

Examples

This chapter shows how SR_e can be used to implement fault tolerant programs. Section 5.1 presents an application of the exception handling facility to solve a control flow problem. Section 5.2 and 5.3 present examples of the resumption model and of the termination model, respectively. Section 5.4 presents an application of the be statement utilizing the timer clause.

5.1. Zahn's Construct

Zahn's construct [Zahn74, Knut74] is a control structure to solve a classical control flow problem and has the form

> **until** *event_list* **do**
>   *body*
> **then case**
>   *events* : *statements*

where the *event_list* specifies the *body*'s terminating events. When an event occurs, the body terminates and the *statements* body corresponding to the terminating event is executed. Then, control is passed to the statement immediately following the Zahn's construct.

Figure 5.1 shows an application of Zahn's construct. In this example, a *table* is searched for an entry matching the given *key*. If the *key* is found in the table, a corresponding *count_part* entry in the *frequency_table* is incremented; otherwise, the variable

```
# assume all variables and tables are properly declared and initialized
until found or not_found do
   for i := 1 to max  such that table[i].key_part = key  do
            position := i;
            found;
   end for;
   not_found;
then case
   found : frequency_table [position].count_part++;
   not_found: other_part++;
```

Figure 5.1 — An example of Zahn's construct.

other_count is incremented.

The use of Ada's exception handling facility to implement the above Zahn's construct example is discussed in [Ausn85]. Similar to that Ada solution, an SR_e frame is used to solve the above example as shown in figure 5.2. The events *found* and *not_found* are implemented as exceptions and are declared locally to the be statement. When an event occurs, a synchronous exception signaling statement—**call**—raises an exception corresponding to the event to the frame. Thus the execution of the body is blocked. The *statements* body associated with each event appears in the handler block component of the frame. The execution of **leave** terminates the be statement. Note that **leave** must appear in each arm of the input statement; otherwise, termination of an arm—handler—would terminate the signaling statement and thus incorrectly resume the block component.

The example is somewhat contrived and can be solved much more efficiently without using exceptions (see figure 5.3). However, it is good for illustrating the flexibility of the Mechanism.

5.2. Convert Operation

The example in this section is a modified version of that given in [Yemi85]. This section gives three solutions that demonstrate the different models described in Chapter 1. The main operation in this example is the *convert* **proc**, which accepts an array of 10

```
# assume all variables and tables are properly declared and initialized
be { op found ()
     op not_found () }
   fa i := 1 to max  st table[i].key_part = key ->
              position := i
              found ()
   af
   not_found ()
handlers
   in found () → frequency_table [position].count_part++; leave
    [] not_found () → other_part++; leave
   ni
eb
```

Figure 5.2 — Zahn’s construct in SR_e.

```
# assume all variables and tables are properly declared and initialized
   fa i := 1 to max  st table[i].key_part = key ->
              position := i
              exit
   af
   if position ≤ max → frequency_table [position].count_part++
    [] else → other_part++
   fi
```

Figure 5.3 — A solution in SR_e without using exceptions.

integers. Each number is validated to ensure that its value is within the collating ordering. The conversion of an input number into its corresponding character in the collating set only occurs if the number is valid.

Figure 5.4 shows a solution to this problem that follows the strict-resumption model. In this solution, the *convert* operation is declared with an array of 10 integers and a

```
resource test
  op convert ( code[1:10] : int;
                  badcode : cap (i : int) returns n : int {call} )
       returns str[1:10] : char
  process main
    var array[1:10] : int
    writes ("input 10 integers: ")
    fa i := 1 to 10 → read (array[i])  af
    var s[1:10] : char
    be { op invalid (i : int) returns n : int {call} }
      s := convert (array,invalid)
    handlers
      in invalid (i) returns n →
           writes (i," is invalid, enter new number: ")
           read (n)
      ni
    eb
    write (s)
  end
  proc convert (code,badcode) returns str
    fa i := 1 to ub(code) →
       do code[i] < int(low(char)) or code[i] > int(high(char)) →
            code[i] := badcode (code[i])
       od
       str[i] := char(code[i])
    af
   end
end
```

Figure 5.4 — *convert* in the strict-resumption model.

badcode capability as parameters and an array of 10 characters as the return value. *badcode* is declared with the {**call**} restriction to indicate that the exception follows the resumption model.

The *main* process reads 10 integers from a keyboard. Then, it calls *convert* passing the list of the 10 numbers and an *invalid* exception capability. Upon returning from the call, it prints the resulting string. Note that the call is placed within a frame with the local exception *invalid.* If the *invalid* exception is signaled while the call is blocked, the handler block component is activated to service the exception. The offending value is printed and a new number is read, which in turn is passed back to the signaler. Then the handler block component is deactivated and the frame continues to be blocked pending the termination of the call.

The *convert* **proc**, when invoked, validates each integer. If an integer is not within the collating ordering, the *badcode* capability is used to signal an exception passing with it the offending value. The call of *badcode* also requests a substitute integer from its handler. When the call terminates, the new value is checked again. Only after a valid integer is entered, the **proc** converts the integer into its corresponding character in the collating set.

Figure 5.5 shows a solution that follows the resumption model. In this solution, when *convert* signals a *badcode* exception, the invoker's handler can either resume or terminate *convert.* The handler is nested within a loop statement so that the handler block component can keep track of the number of *invalid* exceptions that have been handled. When *max_try invalid* exceptions occur, the handler prints an error message and exits the frame to abort the invocation of *convert.* The body of *convert* does not have to be placed

```
resource test
  op convert ( code[1:10] : int;
                 badcode : cap (i : int) returns n : int {call} )
       returns str[1:10] : char
  process main
    var array[1:10] : int
    writes ("input 10 integers: ")
    fa i := 1 to 10 → read (array[i])  af
    var s[1:10] : char
    be { op invalid (i : int) returns n : int {call} }
      s := convert (array,invalid)
      write (s)
    handlers
      const max_try := 3
      var try := 0
      do true →
         in invalid (i) returns n →
              if try < max_try →
                   writes (i," is invalid, enter new number: ")
                                      read (n)
                                      try++
                                      next
              [] else →
                                    write ("abort: more than ",max_try, " errors.")
                                    leave
              fi
         ni
      od
    eb
  end
  proc convert (code,badcode) returns str
    fa i := 1 to ub(code) →
        do code[i] < int(low(char)) or code[i] > int(high(char)) →
              code[i] := badcode (code[i])

        od
        str[i] := char(code[i])
    af
  end
end
```

Figure 5.5 — *convert* in the resumption model.

within a frame since, by default, the language's run-time support will terminate *convert* without returning any result to its invoker if an invocation of *badcode* cannot be resumed.

Figure 5.6 shows another way to respond to an invalid input number that follows the termination model. In this solution, the body of *main* is re-executed when an error occurs. *badcode* is declared with the {**send**} restriction to indicate that asynchronous

```
resource test
  op convert ( code[1:10] : int;
                 badcode : cap (i : int) {send} )
       returns str[1:10] : char
  process main
    be { op invalid (i : int) {send} }
      var array[1:10] : int
      writes ("input 10 integers: ")
      fa i := 1 to 10 → read (array[i]) af
      var s[1:10] : char
      s := convert (array,invalid)
      write (s)
    handlers
      in invalid (i) →
           write ("abort: ", i," is invalid.")
           restart
      ni
    eb
  end
  proc convert (code,badcode) returns str
    fa i := 1 to ub(code) →
       if code[i] < int(low(char)) or code[i] > int(high(char)) →
            send badcode (code[i])
            void
            return
       [] else → str[i] := char(code[i])
       fi
    af
   end
end
```

Figure 5.6 — *convert* in the termination model.

signaling is used. When *convert* detects an invalid number, it just signals *badcode* and terminates the invocation by executing **void.** The void statement is needed to terminate the invocation without returning any result to its invoker. When the invoker's handler is activated, it prints an error message and re-executes the body of *main.* Note that the invocation of *main* is not terminated when **restart** is executed.

5.3. Sum Stream

This section shows how SR_e can support the termination model. The *sum stream* example and a CLU solution are originally given in [Lisk79]. Yemini and Berry also give a similar solution using the replacement model in [Yemi85]. In order to provide a comparison with other solutions, it is assumed that the type **stream** is available in SR_e, that type **string** is modified to conform with the **string** type in CLU, and that an integer overflow suspends the executing construct with reason **overflow.**

sum_stream is an operation that reads in signed decimal integers from a character stream and returns their sum. The input stream is viewed as containing a sequence of number fields separated by spaces and new lines. Each number field must consist of a nonempty sequence of digits, optionally preceded by a single minus sign. Figure 5.7 shows the calling sequence in an implementation of this example. *sum_stream* accepts a stream of numbers and returns their sum. *get_number* returns the value of the next number in an input stream. *get_field* returns the next number field in an input stream. *s2i* converts a number field to an integer. *getc* retrieves the next character in an input stream.

The SR_e solution to this example declares the *sum_stream* operation as

Figure 5.7 — Calling sequence of *sum stream* example.

```
proc sum_stream (s, ovfl, unint, bfmt) returns x
    be { op sum_eof () {send}
               op sum_unint (f : string) {send}
               op sum_bfmt (f : string)  {send}
               op cleanup (r : reasons) }
               x := 0
               do true →
              x := x + get_number(s,sum_eof,sum_unint,sum_bfmt)
               od
    handlers
          in sum_eof () → replace
           [] sum_unint (f) → send unint (f); abort
           [] sum_bfmt (f) → send bfmt (f); abort
           [] cleanup (r) and r = overflow → send ovfl (); abort
              ni
    eb
end
```

Figure 5.8 — *sum_stream* **proc.**

op *sum_stream* (*s* : **stream**;
integer_overflow : **cap** () {**send**};
unrepresentable_integer : **cap** (*f* : **string**) {**send**};
bad_format : **cap** (*f* : **string**) {**send**}) **returns** *x* : **int**

where *integer_overflow*, *unrepresentable_integer*, and *bad_format* are exceptions that *sum_stream* may signal to its invoker. *integer overflow* is raised if an integer overflow occurs in the body of the *sum_stream*. *unrepresentable_integer* is raised if the input string *s* contains an individual number that is outside the implemented range of integers. *bad_format* is raised if the input string *s* contains a field that is not an integer.

Figure 5.8 shows an implementation of the *sum_stream* **proc.** *sum_stream* invokes a *get_number* operation to retrieve individual numbers from the input stream. If *get_number* raises an *end_of_file* exception, *sum_stream* returns the accumulated result to its invoker. If *get_number* signals an *unrepresentable_integer* exception or a *bad_format* exception, or if an integer overflow occurs, then *sum_stream* just propagates the exception to its invoker. Note that different exception names are needed. Otherwise, the propagations only re-raise the exception to the same frame.

The external interface of *get_number* is

> **op** *get_number* (*s* : **stream**;
> *end_of_file* : cap () {**send**};
> *unrepresentable_integer* : **cap** (*f* : **string**) {**send**};
> *bad_format* : **cap** (*f* : **string**) {**send**}) **returns** *x* : **int**

get_number signals *end_of_file* if no more numbers are in the input stream *s*. It propagates *unrepresentable integer* if an extracted *field* represents an integer that is outside the implemented range of integers. It propagates *bad_format* if an extracted *field* contains a minus sign following a digit, more than one minus sign, or no digits.

Figure 5.9 shows an implementation of the *get_number* **proc.** *get_number* invokes the *get_field* operation to retrieve the next field in the input stream *s* and invokes the *s2i* operation to convert the field to an integer. Since each arm of an input statement may

```
proc get_number (s,eof,unint,bfmt) returns x
    var field : string
    be { op get_unint () {send}
             op get_bfmt () {send} }
             op get_invalid_char (c : char)  {send} }
        be { op get_eof () {send} }
                     field := get_field (s,get_eof)
        handlers
                     in get_eof () → send eof (); abort
                     ni
        eb
        x := s2i (field,get_invalid_char,get_bfmt,get_unint)
    handlers
        in get_unint () → send unint (field); abort
         [] get_bfmt () → send bfmt (field); abort
         [] get_invalid_char (c) → send bfmt (field); abort
        ni
    eb
end
```

Figure 5.9 — *get_number* **proc.**

only service a single operation, two different handlers—*get_bfmt* and *get_invalid_char*—are required even though their actions are the same.

The *get_field* operation is declared as

op *get_field* (*s* : **stream**; *end_of_file* : **cap** () {**send**}) **returns** *f* : **string**

If *get_field* is invoked and no fields remain in the input stream *s*, it signals an *end_of_file* exception.

Figure 5.10 shows a solution for the *get_field* **proc**, which is similar to that of the CLU solution. *get_field* invokes a *getc* operation for the next character from the input stream *s*. If the input stream is empty, *getc* signals an *end_of_file* exception. The declaration of *getc* is

```
proc get_field (s,eof) returns field
  be { op get_field_eof () {send} }
    field := ""
    var c : char := getc (s,get_field_eof)
    do c = ' ' cor c = '\n' -> c := getc (s,get_field_eof) od
    be { op get_field_eof () {send} }
      do c ~= ' ' cand c ~= '\n' ->
         f := append (field,c)
         c := getc (s,get_field_eof)
      od
    handlers
      in get_field_eof () → replace  ni
    eb
  handlers
    in get_field_eof () → send eof (); abort  ni
  eb
end
```

Figure 5.10 — *get_field* **proc.**

op *getc* (*s* : **stream**; *end_of_file* : **cap** () {**send**}) **returns** *c* : **char**

In this solution, each frame has a local exception *get_field_eof.* An invocation of *getc* is given with the smallest enclosing frame's exception capability. If end of file is reached within an invocation of the innermost **do**, a *get_field_eof* exception is signaled to the inner be statement. Otherwise, the exception is intended to be handled at the outer be statement.

Figure 5.11 gives an alternative solution for the *get_field* **proc.** In this solution, the be statement declares two different exceptions—*valid_eof* and *invalid_eof*—that have handlers corresponding to the two handlers as shown in figure 5.10. Each invocation of *getc* is given with an appropriate exception signaling capability. Thus when the *getc* signals an end of file exception, a correct handling routine is activated. This solution also

```
proc get_field (s,eof) returns field
   be { op valid_eof () {send}
          op invalid_eof () {send} }
       field := ""
       var c : char := getc (s,valid_eof)
       do c = ' ' cor c = '\n' → c := getc (s,valid_eof)  od
       do c ~= ' ' cand c ~= '\n' →
                 f := append (field,c)
                 c := getc (s,invalid_eof)
       od
   handlers
       in valid_eof () → replace
         [] invalid_eof () → send eof (); abort
       ni
   eb
end
```

Figure 5.11 — *get_field* **proc.**

shows how to improve the readability of a program, that is, by grouping all handler routines in one frame and by providing different exception handling operations with invocation statements at different contexts.

The declaration of *s2i* is

```
op s2i ( s : string;
              invalid_char : cap (c : char) {send};
              bad_format : cap () {send};
          unrepresentable_integer : cap () {send} )
             returns x : int
```

s2i converts the input string *s* to an integer, which it then returns. It signals an *invalid_char* exception if the string contains other than a digit or a minus; a *bad_format* exception if the string contains a minus after a digit, more than one minus, or no digits; and an *unrepresentable_integer* exception if the string represents a number that is outside the implemented range of integers. The implementations of *getc* and *s2i* are straightforward and therefore not shown.

5.4. Handling Interface Malfunctions

This section shows an application of the timer clause facility of SR_e. In this example, a *transmitter* process is responsible for sending individual characters that it receives from a transmission request to a serial interface for transmission. After sending a character, it waits for an interrupt from the serial interface. If this interrupt does not arrive during a specified amount of time, it assumes that a hardware malfunction has occurred and the transmission request fails.

A solution in Ada of this example is given in [Ausn85]. Figure 5.12 shows a similar solution in SR_e. For simplicity, the code related to the serial interface operation is omitted. The *transmitter_package* exports two operations: *transmit_character* and *serial_interface_interrupt.* The *transmit_character* operation receives two arguments—a character *c* and a *serial_interface_malfunction* exception capability—from its invoker. After a character is sent to the *hardware_output* **proc**, the execution of the *transmit_character* is blocked pending the arrival of the *serial_interface_interrupt* message. If the message does not arrive during the *interval_between_chars*, the frame containing the **receive** statement is suspended with reason **timeout.** Its handler then signals the *serial_interface_malfunction* asynchronously—**send**—and terminates the *transmit_character* invocation via **abort.**

```
resource transmitter_package
   op transmit_character (c : char; serial_interface_malfunction : cap ())
   op serial_interface_malfunction ()
body transmitter_package ()
   const interval_between_chars := 3
   op hardware_output (byte : char)

   process transmitter
      do true →
         in transmit_character (c,serial_interface_malfunction) →
            op serial_interface_interrupt ()
            hardware_output (c)
            be { op cleanup (r : reasons) } timer interval_between_chars
               receive serial_interface_interrupt ()
            handlers
               in cleanup (r) and r = timeout →
                  send serial_interface_malfunction ()
                  abort
               ni
            eb
         ni
      od
   end

   # the serial interface operation
   proc hardware_output (byte)
      # body of hardware_output
   end
end
```

Figure 5.12 — *transmitter_package* resource.

CHAPTER 6

Simulation and Implementation

This chapter shows how some features of the Mechanism can be simulated in standard SR (section 6.1) and discusses issues in an implementation of the Mechanism (section 6.2).

6.1. Simulation

Parts of the Mechanism can be simulated using standard SR primitives. To get some feel for the complexity of the Mechanism, this section describes how the explicit exception signaling and handling features of the Mechanism can be simulated in standard SR.

6.1.1. Assumptions

Since standard SR does not support failure detection and handling, it is assumed that no actual failure may occur in SR_e programs. Furthermore, to simplify the transformation process, all variables are declared global.

6.1.2. Transforming Operations

To support non-blocking of invocations, all values being returned from a server must be sent back via messages. Thus, only **val** parameters are allowed, and all **var** and **res** parameters and **returns** variables must be defined as arguments of a capability in the operation declaration. Each server contains a *reply_void* boolean variable to record whether a **reply** or **void** associated with the server has been executed.

Each input statement in a *block* or a *handler_block* **proc** is extended with one arm to accept the termination message—*hdl_end* or *blk_end*—from the be statement monitor (see 6.1.4.1). Each **do**, **if**, and **fa** statements within a frame also checks for these terminating messages by adding condition checking such as ?*blk_end* = 0 or ?*hdl_end* = 0. Similarly, a conditional statement can also be inserted in a sequence of statements to check for these terminating messages.

6.1.3. Transforming Synchronous Statements

Each **call** in a frame is transformed to an asynchronous invocation—**send**—followed by an **in** statement to receive results from its server and a terminating message. For example, a **call** *operation_denotation* is transformed to

send *operation_denotation*
in *result* (*result_parameters*) -> # copy *result parameters* to *result variables*
 [] # arm to accept *hdl_end* or *blk_end*
ni

6.1.4. Simulating the be Statement

A be statement is decomposed into a frame monitor, a *block* **proc**, and a *handler_block* **proc.** The be statement monitor or simply a monitor is needed since the block and the handler block components are simulated as **proc**s and can terminate the frame independently and concurrently. The monitor of a frame activates and controls the execution of the *block* and the *handler_block* **proc**s of the frame. *block* simulates the execution of the block component whereas *handler_block* simulates the execution of the handler block component.

6.1.4.1. The Frame Monitor

Figure 6.1 shows the structure of a frame monitor. A monitor consists of a simple loop that activates and controls the execution of the block and the handler block components of a frame. SR communication primitives provide a dynamic binding between a monitor and its *block* and *handler_block* **proc**s. Thus the statements that can be executed within a frame such as **leave**, **restart**, **abort**, **reply**, **void**, and **return** are converted to *be_leave*, *be_restart*, *be_abort*, *be_reply*, *be_void*, and *be_return* messages, respectively. *block* and *handler_block* are activated with proper capabilities so that they can request their monitor to execute these statements on their behalf. The *blk_end* and a *hdl_end* variables associate an instance of the block and handler block **proc**s with its monitor.

After activating the block and handler block **proc**s, the monitor waits for a request. When it receives a *be_leave* message, it terminates *block* and *handler_block* and then exits the loop. When it receives a *be_restart* request, it also terminates *block* and *handler_block* but then re-executes the loop. When it receives a *be_abort* message, it notifies the terminator and terminates *block*, *handler_block*, and the smallest enclosing server. When it receives a *be_void* message, it just notifies the terminator server and sets the *reply_void* variable to true. When it receives a *be_reply* message, it sends the results to the invoker of the smallest enclosing server and sets the *reply_void* variable to true. When it receives a *be_return* request, it sends the results to the invoker of the smallest enclosing server and terminates *block*, *handler_block*, and the server. The *reply_void* boolean variable indicates whether a result has been sent back to the invoker of the smallest enclosing server of the frame.

```
do true →
  op be_void ()             # void
  op be_abort ()            # abort
  op be_reply ()            # reply
  op be_return_replace ()   # return and replace
  op be_restart ()          # restart
  op be_leave ()            # leave
  var hdl_end : cap ()      # to terminate handler_block
  var blk_end : cap ()      # to terminate block
  # each exception declaration in be frame needs a cap
  var exception_cap : cap (arguments)
  # create handler block component
  handler_block ( be_return_replace,
                  be_abort,
                  be_leave,
                  be_restart,
                  exception_capabilities,
                  hdl_end )
  # create block component
  block ( be_reply,
          be_return_replace,
          be_void,
          be_leave,
          exception_cap,
          blk_end )
  # monitoring execution of block and handler block
  in be_leave () → hdl_end (); blk_end (); exit
   [] be_void () → if ˜reply_void → reply_void := true
                                                        # notify terminator
                   fi
   [] be_abort () → if ˜reply_void → # notify the terminator
                                      hdl_end ()
                                      blk_end ()
                                      return
                    fi
   [] be_reply () → if ˜reply_void → send result (result_list)
                                     reply_void := true
                    fi
   [] be_return_replace () → if ˜reply_void → send result (result_list)
                                              hdl_end ()
                                              blk_end ()
                                              return
                             fi
   [] be_restart () → hdl_end (); blk_end (); next
  ni
od
```

Figure 6.1 — Structure of a frame monitor.

6.1.4.2. The Block Component

block is created by a monitor to execute actions in the block component of a frame. The termination of *block* requires an *end_block* message. Figure 6.2 shows a structure of a *block* **proc.** When *block* is activated, it is given all exception capabilities of the frame and of the enclosing frames so that it can raise an exception to *handler_block* directly. *block* returns an *end_block* capability to the monitor as the terminating message capability. The rest of *block*'s body is the body of the block component with all necessary transformation.

```
op block ( be_reply : cap ();
           be_reply_return : cap ();
           be_void : cap ();
           be_leave : cap ();
           exception_capability_list;
           res blk_end : cap () )

proc block ( be_reply,
             be_return,
             be_void,
             be_leave,
             exceptions,
             end_blk )
   op end_block ()
   end_blk := end_block
   reply
   # body of block component
end
```

Figure 6.2 — Structure of a *block* **proc.**

```
op handler_block ( be_replace : cap ();
                                be_abort : cap ();
                                be_leave : cap ();
                                be_restart : cap ();
                                res exception_capability_list;
                                res hdl_end : cap () )

proc handler_block ( be_replace,
                                be_abort,
                                be_leave,
                                be_restart,
                                exceptions,
                                end_hdl )
   op end_handler_block ()
   # exception declarations
   exception_op_list
   # assign exception_op_list to exceptions
   end_hdl := end_handler_block
   reply
   do true →
     # body of handler block component
   od
end
```

Figure 6.3 — Structure of a *handler block* **proc.**

6.1.4.3. The Handler Block Component

Similar to *block*, *handler_block* is also created by a monitor to execute actions in the handler block component of a frame. The termination of *handler_block* requires an *end_handler_block* message. Figure 6.3 shows a structure of a *handler_block* **proc.**

All exceptions that are local to the frame are declared in *handler_block.* After *handler_block* is activated, it returns all exception handling capabilities to the monitor so that they can be passed to appropriate components. Then it loops, repeatedly servicing exceptions. Each iteration represents one activation of the handler block component. The body of the loop is the body of the handler block component with necessary transformation. A **deactivate** is transformed to reiterate the outermost **do** in figure 6.3.

6.1.5. Example

To illustrate how to transform SR_e programs into standard SR, the strict-resumption model solution of the convert operation example from section 5.2 is shown in standard SR in figures 6.4 – 6.6.

The *main* process is transformed to a **proc** with a *result* parameter and is created by a **send** with a **noop** capability. The be statement in the original solution is transformed to

```
resource test ()
  op main ( result : cap () )
  op convert ( code[1:10] : int;
               badcode : cap ( i : int; result : cap (x : int) );
                   convert_reply : cap (str[1:10] : char) )
  op block ( be_reply : cap ();
                   be_return : cap ();
                   be_void : cap ();
                   be_leave : cap ();
              invalid : cap ( i : int; res result : cap (n : int) )
                   res blk_end : cap () )
  op handler_block ( be_replace : cap ();
                         be_abort : cap ();
                         be_leave : cap ();
                         be_restart : cap ();
                   res exception : cap ( i : int; result : cap (n : int) );
                         res hdl_end : cap () )
  var array[1:10] : int
  var s[1:10] : char
  initial
    send main (noop)
  end
```

Figure 6.4 — Simulation of *convert* in SR—Part 1.

a **do** statement and two **proc**s—*block* and *handler_block*— as described in section 6.1.2. The *convert* operation declaration declares a *convert_reply* capability to send the return value to its invoker. The synchronous invocation *convert* is transformed into an asynchronous invocation immediately followed by an input statement as suggested in section 6.1.4.

The resulting program can be simplified since the *convert* example follows the strict-resumption model and is elementary. For example, the declarations and implementations of *be_reply*, *be_return*, *be_void*, and *be_leave* can all be eliminated because the handler block component must resume the signaler and the program always terminates normally.

6.2. Implementation Outline

The example in section 6.1.5 also illustrates in part the complexity in an actual implementation of the Mechanism. This section outlines a simplified version of an implementation of SR_e, which is based on the implementation of SR. However, the actual code has yet to be developed and is expected to be more complicated than what is described in this section.

An implementation of SR_e is expensive for two reasons. First, the exception signaling and handling is based on SR communication primitives, which are inherently expensive. Second, the block and the handler block components have the potential to be executed concurrently; therefore, they are in essence to be implemented as **proc**s, which are also not ordinary procedures in SR.

```
proc main (result)
   var reply_void := false
   writes ("input 10 integers: ")
   fa i := 1 to 10 → read (array[i])  af
   # simulate the be statement
   do true →
      op be_abort ()
      op be_void ()
      op be_reply ()
      op be_return_replace ()
      op be_restart ()
      op be_leave ()
      var hdl_end, blk_end : cap ()
      var invalid : cap ( i : int; res result : cap n : int) )
      handler_block ( be_return_replace, be_abort,
                                     be_leave, be_restart,
                                     invalid, hdl_end )
      block ( be_reply, be_return_replace,
                    be_void, be_leave,
                    invalid, blk_end )
      in be_leave () → hdl_end (); blk_end (); exit
      [] be_void () → if ˜reply_void → reply_void := true
                                                  # notify terminator
                           fi
      [] be_abort () → if ˜reply_void → reply_void := true
                                                  # notify terminator
                                                  hdl_end ()
                                                  blk_end ()
                                                  return
                           fi
      [] be_reply () → if ˜reply_void → send result ()
                                                  reply_void := true
                           fi
      [] be_return_replace () → if ˜reply_void → send result ()
                                                              hdl_end ()
                                                              blk_end ()
                                                              return
                                      fi
      [] be_restart () → hdl_end (); blk_end (); next
      ni
   od
   write (s)
end
```

Figure 6.5 — Simulation of *convert* in SR—Part 2.

```
proc block (be_reply, be_return, be_void, be_leave, badcode, end_blk)
    op end_block ()
    end_blk := end_block
    reply
    # call convert
    op convert_reply (str[1:10] : char)
    send convert (array,badcode,convert_reply)
    in convert_reply (str) → s := str
     [] end_block () → return
    ni
    # terminate frame
    send be_leave ()
    receive end_block ()
end

proc handler_block ( be_replace, be_abort, be_leave,
                                    be_restart, exception, end_hdl )
     op end_handler_block ()
     op invalid ( i : int; res hdl_reply : cap (n : int) )
     exception := invalid
     end_hdl := end_handler_block
     reply
     do true → in invalid (i,hdl_reply) and ?end_handler_block = 0 →
                    var n : int
                      writes (i, " is invalid, enter new number: "); read (n)
                              send hdl_reply (n)
                       [] end_handler_block () → exit
                      ni
     od
end

proc convert (code,badcode,convert_reply)
    op hdl_reply (x : int)
    var str[1:10] : char
    fa i := 1 to ub(code) →
       do code[i] < int(low(char)) | code[i] > int(high(char)) →
             send badcode (code[i],hdl_reply)
             in hdl_reply (x) → str[i] := char(x)  ni
       do
       str[i] := char(code[i])
    af
    send convert_reply (str)
end
end
```

Figure 6.6 — Simulation of *convert* in SR—Part 3.

6.2.1. Compiler Modifications

The modification of SR's lexical analyzer and parser is straightforward. All the new tokens and production rules have to be recognized. The parser also has to ensure that all declared exceptions are implemented in the handler block component and not in the block component of the frame; that **void**, **replace**, **abort**, **leave**, **restart**, and **deactivate** are only executed in a proper construct; and that an invoker always provides proper exception capabilities with an invocation.

The major modification in the compiler is in the code generator. In this outline, the service provided by a frame is implemented in the language's extended run-time support (RTS_e). The role of the code generator is to generate code for the new constructs that invokes appropriate routines in RTS_e. The concurrent access of variables shared by the the block and the handler block components of a frame can be supported by using indirect variable referencing.

For each instance of a frame, the frame is registered with RTS_e. When the frame terminates, its record in RTS_e has to be deleted. Similarly, each invocation's activation record and its associated frame has to be registered with the terminator running in RTS_e. The generated code has to distinguish an invocation from an exception signaling and to forward exception signaling operations to RTS_e. In order to distinguish between a "true" operation and an exception, one bit is added to each operation's data structure.

After the expression in a timer clause is evaluated, the resulting deadline value with its associated frame is forwarded to RTS_e. The generated code also has to invoke RTS_e to perform side effects that are related to the frames when one of the following statements are executed—**restart**, **void**, **leave**, **replace**, **abort**, **deactivate**, **reply**, **return**, **next**, and

exit.

6.2.2. Extended Run-Time Support—RTS_e

RTS_e enforces the semantics of the Mechanism. It monitors the deadline of each executing frame. An efficient timeout detection data structure is a deadline queue in which the relative timeout value—the difference with respect to that of the last frame in the queue—is stored. At some predetermined interval, the deadline queue is visited and all executing frames that have their deadline values expired are deleted from the queue and suspended with the reason **timeout.**

When an exception is signaled, RTS_e searches for the frame that is responsible to handle the exception, activates the handler block if necessary, and delivers the exception to the handler. When a frame terminates, it notifies RTS_e to ensure that both the block and the handler block components are terminated and are removed from its record. RTS_e is also responsible to deliver the result of an invocation to its invoker and to keep track the status of all active invocations and frames.

The terminator is a special process in RTS_e and is run periodically. At certain intervals, the terminator broadcasts its status to other resources in the system. Conversely, if the terminator fails to receive the status messages from other resources, it assumes that these resources have failed and terminates all pending invocations to and from these resources.

CHAPTER 7

Discussion

This chapter evaluates the Mechanism. Section 7.1 compares the exception handling features of the Mechanism with other similar facilities surveyed in Chapter 2. Section 7.2 assesses the SR_e language. In particular, it considers the advantages and disadvantages of using SR's communication primitives as the basis of exception handling.

7.1. Comparison

Figure 7.1 compares the exception handling facility of the Mechanism with the proposals of Goodenough, Cocco and Dulli, and Yemini and Berry, and with similar facilities in Ada, CLU, LISP, and Mesa. The comparison is based on the following categories:

- Model—classifies the exception handling according to the four models described in Chapter 1.
- New primitives—indicates the relative number of language primitives needed to support the exception handling features.
- Compile-time checking—indicates whether unhandled exceptions can be detected at compile-time.
- Granularity—indicates the smallest unit with which a handler can be associated.
- Parameterization—shows whether a signaler can pass arguments with an exception.

	Model	New primitives	Compile-time checking	Granularity	Parameterized
Goodenough	combination	many	yes	expression	no
Cocco & Dulli	combination	few	yes	unit	yes
Yemini & Berry	resumption	few	yes	expression	yes
Ada	termination	few	no	frame	no
CLU	termination	few	yes	statement	yes
LISP	termination	few	N.A.	function call	no
Mesa	combination	many	no	statement	yes
SR_e	combination	many	yes	frame	yes

Figure 7.1 — Comparison of Various Exception Handling Mechanisms.

7.1.1. Goodenough's Proposal

Goodenough's proposal provides numerous primitives to support the combination model. Exceptions and handlers can be associated with expressions and statements. The requirements of explicit exception declarations, static association of handlers with exceptions, and explicit propagation permit a compiler to detect unhandled exceptions. However, since exceptions are parameterless, a signaler cannot provide additional information with an exception.

7.1.2. Cocco and Dulli's Proposal

Similar to the Goodenough's proposal, Cocco and Dulli's exception handling mechanism supports the combination model. Exceptions are associated with units, which provide modularity of programs, and can be parameterized. Only a few primitives are required to support the proposal. The requirement of full declaration and explicit signaling also permits a compiler to detect unhandled exceptions.

7.1.3. Yemini and Berry's Proposal

The replacement model, a version of the resumption model, proposed by Yemini and Berry is well integrated into the base language. Signaling and handling exceptions are just procedure invocation and servicing. Thus, minimum language support is needed. Furthermore, the full declarations of exceptions permit unhandled exceptions to be detected at compile-time. The model can be embedded in any expression-oriented language and can also be adapted to other languages.

7.1.4. Ada

Ada's exception handling mechanism follows the termination model and is applied to **begin** − **end** blocks. The mechanism only requires a few support primitives. Since exceptions are parameterless, are automatically propagated along the invocation chain, and are not declared in the external interface of the signaler, a compiler cannot assist in detecting unhandled exceptions.

7.1.5. CLU

CLU's exception handling mechanism conforms well with one of the language's important design goals—simplicity. The mechanism supports the termination model and is strongly typed. Exceptions must be declared at the external interface specification of a signaler and propagated explicitly, thus a compiler can identify unhandled exceptions. Exceptions can be parameterized and handlers are only associated with statements.

7.1.6. LISP

LISP's exception handling mechanism only supports multilevel return of function calls. It follows the termination model and is parameterless. Only a few primitives are required to support the mechanism. Since, in general, LISP is an interpreted language, compile-time checking for unhandled exception is not applicable.

7.1.7. Mesa

Mesa's exception-handling mechanism provides parameterized exceptions and supports the combination model. Mesa's exceptions are automatically propagated along the invocation chain, thus compile-time checking is impossible. Contrary to Ada and CLU, Mesa requires many support primitives. Handlers are associated with invocation statements and closed constructs.

7.1.8. SR_e

The Mechanism supports the combination model. The requirement of explicit declaration of all possible exceptions and explicit signaling allows compile-time checking for unhandled exceptions possible. Even though the Mechanism requires numerous support primitives, the exception handling facility is relatively simple and integrated well with the base language. Exceptions are parameterized and a frame is the unit of exception handling.

7.2. Assessment

The concept of frames augmented with timeout and SR communication primitives introduces a simple failure detection and handling mechanism for SR, yet its impact on the base language is minor (see section 4.2). This section describes other advantages and limitations of the Mechanism.

7.2.1. Failure Detection

A timed frame is a low-cost construct to provide failure detection service and can be applied uniformly to invokers and servers. By encapsulating an invocation statement or a server in a timed frame, programmers can control the maximum duration to execute an invocation. One drawback, however, is that it is very difficult to estimate deadlines. Furthermore, since the deadline value has no effect on the handler block component, the Mechanism cannot guarantee that a server will be aborted when the deadline has expired. Therefore, an orphan may interfere with other servers' execution . A solution would be to require each server to monitor subsequent invocations.

The terminator is executed by the language's run-time support. The heartbeat approach allows the system to detect failures in the background.

7.2.2. Failure Recovery

The Mechanism does not introduce new primitives to support automatic crash recovery in order to preserve SR's design goals—simplicity, expressiveness, and efficiency.

However, this approach violates the principles of user friendliness. Were the Mechanism to support automatic recovery, the performance of the language would be reduced drastically since the language's run-time environment would need to be saved periodically in stable storage, which is an expensive operation. A stable storage primitive similar to that in Schlichting's proposal [Schl87] is an attractive compromise solution since it allows users to specify the objects that are needed to be saved permanently while it shields users from the implementation details.

7.2.3. Failure Handling

Using SR communication primitives as the basis for exception signaling and handling provides a powerful and flexible failure handling facility. It also raises some critical issues that would not exist if a completely new set of primitives were introduced. Figure 7.2 summarizes the advantages and disadvantages of basing exceptions on SR's communication primitives.

Synchronous invocation is well suited for the strict-resumption model while the asynchronous invocation augmented with **void** can be used to simulate the termination model. The ability to terminate a synchronous signaling statement via **void** allows SR_e to provide service for the resumption model. Exception capabilities introduce a simple solution for multilevel exception signaling, which permits an exception to be raised directly to its handler; thus it is particularly useful in the resumption model. By requiring an invoker to provide all exception capabilities to its server, a single-level signaling discipline can also be achieved. Furthermore, SR's input statement augmented with **reply** provides a conversation facility between a signaler and its handler.

	Description	Source
Advantages	support combination model	flexibility of SR operations
	asynchronous signaling	send statement
	multilevel signaling	op capability
	compile-time checkable	full declaration of op
	easy to learn	based on operation primitives
	conversation between signaler and handler	block construct
Disadvantages	propagation of exception needs different exception name	scope rule of op
	concurrent execution of block and handler block	send and reply statements
	unexpected exception signaling	capability passing
	expensive	operation invocation
	requires many new primitives	handler as input statement
	invoker may disable signaler's ability to raise exceptions	noop and null capabilities

Figure 7.2 — Consequence of Basing Exceptions on SR Communication Primitives.

The idea of defining only one language-defined exception is convenient in the Mechanism. Since each arm of an input statement may only service one operation, all exceptions declared in a frame must be handled in different arms—handlers—even though the body of each handler is the same. By using a cleanup exception with suspending reasons, one can use the synchronization expression in an input statement to associate different reasons with one handler or to provide one handler for each reason. The example in figure 5.11 also shows how to associate the same exception with different handlers under different contexts in order to improve the readability of the program.

'xcv'

One of the major contributions of this research is that it introduces the notion of asynchronous exception signaling and handling. An application of asynchronous signaling is that in a long computation, a server can send intermediate results or status back to its invoker, while the invoker is still blocked for the final result and the server does not require immediate attention from the handlers. This interaction may assist in the recovery process if the invocation is subsequently terminated abnormally.

The ability to construct a handler block component with arbitrary code provides a flexible exception handling facility. Figure 5.5 shows how programmers can encapsulate the code to monitor the number of times that a handler has been invoked within a handler block component. This facility is not available in any exception handling mechanism surveyed in this thesis. For example, the **exception** part of Ada's **begin** block can contain only handlers.

Using the SR communication primitives as the basis for exception signaling and handling also creates some unexpected disadvantages. First of all, it requires more keywords and statements than originally expected. SR has about 70 keywords. The Mechanism introduces 10 new keywords, 7 new primitives, 1 predefined enumerated type, and 1 predefined exception. The existence of asynchronous invocation and **reply** also raises the possibility of concurrent execution of the block component, the handler block component, and the signaler. An actual implementation of the Mechanism is therefore expected to be expensive (see section 6.2).

Since a construct can raise an exception as long as it has a capability for an exception of a frame, multiple exception signalings to the frame and unexpected exception signalings can occur. For example, if an invoker provides an exception capability of an enclosing frame to its server *A*, and the capability is further passed to server *B*, then *A* and *B* may signal an exception to the frame concurrently. Now, assume that server *A* terminates the original invocation and that server *B* is still executing. Any exception raised by *B* using the same exception capability is thus unexpected by the frame.

The scope rule of operations prevented a signaler from propagating an exception that is also declared locally. Section 5.3 shows a way to propagate an exception using different names. The following shows how to propagate an exception with the same name that is declared in a nested frame.

Example:

```
be { op overflow () }
   var overflow_alias : cap () := overflow     # to resolve scope problem
   be { op overflow () }
      # do something
   handlers
      # handler for overflow in inner frame
      in overflow () →
            call overflow_alias()     # propagate overflow to outer frame
      ni
   eb
handlers
   # handler for overflow in outer frame
   in overflow () → ...  ni
eb
```

In this example, both the outer and the inner frames declare an *overflow* exception. The capability of the *overflow* exception of the outer frame is stored in the *overflow_alias* variable so that it can be raised in the inner frame. When an *overflow* exception is signaled to the inner frame, the exception is propagated via signaling the *overflow_alias* variable.

SR’s **noop** and **null** capability literals deserve some attention. Recall (section 3.4.2) that execution of an invocation statement causes a run-time error if the denoted operation is a capability with value **null** and that execution of an invocation statement has no effect if the denoted operation is a capability with value **noop.** Since an invoker can provide **noop** or **null** as an exception capability, it can disable a signaler’s ability to raise an exception, which violates the intuitive concept of exception signaling. The dependence of exception handling on the input statement also produces some irregularity. For example, a server can now be terminated by two different statements—**return** and **replace.**

CHAPTER 8

Conclusions and Future Research

This thesis addresses the issues of utilizing the concept of frames augmented with timeout and SR communication primitives to be the basis of a failure detection and handling mechanism for SR. The major concentration of this research is on the exception handling facility of the Mechanism. This chapter summarizes the work of this thesis and discusses future research.

8.1. Summary

A frame is a delimited SR block. Failure detection is supported by a timed frame and the terminator. A timer clause specifies the deadline of a frame that in turn may contain an invocation or an input statement. Invocations that are not started or completed during a specified amount of time are considered to have failed. An input statement that is not invoked or that has not completed within a specified amount of time is aborted. The terminator monitors the program through heartbeat; thus, it provides a default failure detection in the background. It ensures that all orphans are terminated and that invokers are notified if their invocation have failed.

Failure recovery is not addressed in the Mechanism and must be implemented by programmers using standard SR primitives. For example, the input/output facility in conjunction with the file system can simulate the stable storage that is essential in restoring the run-time environment in case of crashes. The resource initialization routine allows programmers to implement crash recovery procedures.

'xcix'

Failure handling is based on exception handling, which follows the combination model. Exceptions are declared, signaled, and handled in ways very similar to those for SR operations. A frame must fully declare all exceptions that the frame can handle. Each operation must declare all possible exceptions that can be raised by its implementations. An invoker must provide capabilities of the handlers that service exceptions raised by its server. The Mechanism only introduces one language-defined exception—*cleanup* with one parameter. The cleanup exception provides a convenient way to associate multiple exceptions with one action while its parameter can be used to distinguish different terminating reasons.

By basing the exception handling facility on SR communication primitives, exceptions are parameterized and compile-time checking for unhandled exceptions is possible. An exception is declared similarly to an operation and can be raised synchronously or asynchronously. Exception capabilities allow an exception to be explicitly propagated along the invocation chain or directly to its handler. A handler may resume or terminate the signaler, propagate an exception to another frame, re-execute the frame with which the handler is associated, and terminate the smallest enclosing server.

On the negative side, the Mechanism has some disadvantages. First, a deadline is difficult to estimate. Second, the orphan elimination algorithm of the terminator has not yet been developed. Third, using SR file manipulation primitives to implement stable storage is inconvenient. Finally, the handler block component requires many primitives to support its flexibility.

’c’

8.2. Future Research

Even though the explicit exception signaling and handling of the Mechanism can be simulated using standard SR, the practicality of the Mechanism can only be studied through an actual implementation of the Mechanism. As illustrated in section 6.2, a prototype of SR_e, especially the exception handling features, can be developed easily by an experienced developer. However, the cost of the Mechanism’s extra overhead is high. In a normal execution—one in which exceptions are not raised—RTS_e needs to be notified of the instantiation of and the exit from a frame. The handler block has to be implemented like a **proc** requiring a separate process. Thus, an efficient implementation of SR_e might be difficult to achieve.

More work is needed to perfect the Mechanism. Details of the terminator need to be developed and the definition of the timer clause should be investigated further. [Walk84] outlines an algorithm for orphan elimination. The semantics of a deadline is very general; for example, it is used to monitor both the beginning and the finishing time of an invocation. A refinement of the timer clause should be made when more experience using SR_e is obtained.

Since the Mechanism does not provide stable storage primitives, it is cumbersome to use to support failure recovery. The failure detection and handling mechanism suggested by Schlichting is attractive in the sense that programmers can specify stable objects and have the control to request the language’s run-time support to begin or to end monitoring a given processor. It also provides a uniform primitive that can be applied to both recovery and handling—the **binding** variable. Therefore, one possible refinement of the Mechanism is to adopt some of Schlichting’s approach into the Mechanism such as

stable and **binding** variables. Unfortunately, his approach does not provide a general exception handling facility and is expensive.

The rule to search for a proper handler when RTS_e detects an exceptional condition as described in section 4.1.4.2 is complicated and should be simplified as well. The complication occurs because RTS_e signals exceptions according to the termination model and exceptions can be originated from within a frame (invocation failed), from an enclosing server (orphaned), and/or from an enclosing frame (timeout). Furthermore, nested frames must be terminated from the innermost frame to the outermost one.

Asynchronous exception signaling and handling provides a powerful and flexible exception signaling and handling facility. Its usage should be exploited. An application of these features is that multiple exceptional conditions can be signaled to a handler concurrently and that the handler block component can determine which exceptions are more critical. It also permits conversations between a signaler and its handler and allows a signaler to raise exceptions to different frames.

In addition to the in-depth investigation of the usefulness of the Mechanism, SR_e provides an interesting case study in the development of proof rules for concurrent programming languages. Exception handling is inherently difficult to be verified because exceptions are, by definition, denoted unexpected conditions. One of the problems of unverifiability is that exceptions cannot be fully declared. SR_e takes the approach that all possible exceptions must be fully defined when an operation is declared, that the binding between a handler and signaler is explicit, and that the point at which a signaler continues execution after signaling an exception is well defined. Thus, it is expected that proof rules can be developed for the exception handling features of SR_e.

’cii’

An axiomatic approach to provide proof rules for exception handling features in Ada, Cocco’s proposal, and the replacement model are discussed in [Barr82, Cocc82, Luck80, Yemi82]. A prototype for an axiomatic semantics for the standard SR is being developed [Arch89]. Since the exception handling features of the Mechanism is based on SR operations, the prototype can be extended to provide proof rules for the exception handling features of SR_e.

APPENDIX

A Synopsis of the SR_e Language

The general forms of the main SR_e constructs are summarized below. They are similar to those in standard SR except the exception statements and **void.** *Optional items are enclosed in brackets, and plurals and ellipses are used to indicate zero or more occurrences of items. The syntax supports more flexibility than is shown here. For example, items in components and blocks may appear in any order and may be intermixed.*

Components

global	**global** identifier constants, types, and operation types **end** [identifier]

simple combined resource	**resource** identifier([parameters])
	    declarations
	    procs and processes
	end [identifier]

general combined resource	**resource** identifier
	    import or **extend** component_identifiers
	    constants, types, or operations
	body identifier([parameters])
	    [**import** component_identifiers]
	    declarations
	    [**initial** block **end**]
	    procs and processes
	    [**final** block **end**]
	end [identifier]

abstract spec	**resource** identifier
	    import or **extend** component_identifiers
	    constants, types, or operations
	end [identifier]

separate concrete spec	**resource** identifier
	    import or **extend** component_identifiers
	    constants, types, or operations
	body identifier([parameters]) **separate**

’civ’

separate resource body	**body** identifier
	    [**import** component_identifiers]
	    declarations
	    [**initial** block **end**]
	    procs and processes
	    [**final** block **end**]
	end [identifier]

block	declarations statements

Types and Declarations

basic types	**bool**, **int**, **char**, **string**(integer_expression)
type declaration	**type** identifier = type_definition
type definitions	**enum**(enumeration literals) **rec**(field definitions) **ptr** type **cap** resource or operation or operation type
constant	**const** identifier [subscripts] [: type] := expression
uninitialized variable	**var** identifier[subscripts], ... : type
initialized variable	**var** identifier[subscripts] [: type] := expression
operation type	**optype** identifier [=] ([parameters]) [**returns** result]
operation	**op** identifier[subscripts] ([parameters]) [**returns** result] **op** identifier[subscripts] [:] optype_identifier **sem** identifier[subscripts]

'cv'

Procs and Processes

proc	**proc** identifier([formal_identifiers]) [**returns** result_identifier] block **end** [identifier]
process	**process** identifier[quantifiers] block **end** [identifier]

Sequential Statements

skip	**skip**
stop	**stop** [(expression)]
assignment	variable := expression variable ++ variable −− variable :=: variable
alternation	**if** boolean_expression → block [] ... **fi**
iteration	**do** boolean_expression → block [] ... **od** **fa** quantifiers → block **af** **exit** **next**
quantifier	bound_variable := expr1 direction expr2 [**st** bool_expr]

Exception Statements

frame	**be** [{operations}] [**timers** expression] [block] [**handlers** block] **eb** **leave**
handling statements	**abort** **replace**

’cvi’

restart
deactivate

Interaction Statements

operation invocation	[**call**] operation([actuals])
	send operation([actuals])
	V(semaphore)
operation implementation	**in** [(quantifier)] operation([formal_identifiers])
	[**and** boolean_expression] [**by** expression]
	→ block [] ... **ni**
	receive operation([variables])
	P(semaphore)
	return
	reply
	void
resource control	capability := **create** resource_id([actuals]) [**on** virtual_machine]
	capability := **create** **vm**() [**on** physical_machine]
	destroy capability
concurrent invocation	**co** [(quantifiers)] invocation [→ block] // ... **oc**

Keywords (new SR_e keywords are marked by "* ")

EOF	**fi**	**receive**
***abort**	**file**	***replace**
af	**final**	**reply**
and	***handlers**	**res**
any	**global**	**resource**
***be**	**if**	***restart**
body	**import**	**return**
bool	**in**	**returns**
by	**initial**	**sem**
call	**int**	**send**
cap	***leave**	**separate**
char	**new**	**skip**
co	**next**	**st**
const	**ni**	**stderr**
create	**noop**	**stdin**
***deactivate**	**not**	**stdout**
destroy	**null**	**stop**
do	**oc**	**string**
downto	**od**	*** timer**
***eb**	**on**	**to**
else	**op**	**true**
end	**optype**	**type**
enum	**or**	**V**
exit	**P**	**val**
extend	**proc**	**var**
external	**process**	***void**
fa	**ptr**	**xor**
false	**rec**	

REFERENCES

[Ada83] *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD-1815A-1983.

[Andr88] Andrews, G.R., Olsson, R.A., Coffin, M., Elshoff, I., Nilsen, K., Purdin, T., and Townsend, G. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10, 1 (Jan 1988), 51-86.

[Andr89] Andrews, G.R. and Olsson, R.A. Report on the SR programming language, version 1.1. CSE-89-11, Division of Computer Science, University of California, Davis, California, May, 1989, (TR 89-6, Dept. of Computer Science, The University of Arizona, Tucson, Arizona).

[Arch89] Archer, M., Fisher, G., Levitt, K., Olsson, R., Alves-Foss, J., Buffenbarger, J., Frincke, D., Huang, D., Windley, P. Specification and verification of secure concurrent and distributed software systems. Internal report in preparation. Division of Computer Science, University of California, Davis, California.

[Ausn85] Ausnit, C., Cohen, N., Goodenough, J., and Eanes, R.S. *Ada in Practice*. Springer-Verlag, 1985.

[Barr82] Barringer, G. and Mearns, I. Axioms and proof rules for Ada tasks. *IEE Proc.*, Vol. 129, Pt. E, No. 2 (March 1982) 38-48.

[Bern87] Bernstein, A. J. Predicate transfer and timeout in message passing systems. *Information Processing Letter*, 24, 1 (Jan. 1987) 43-52.

[Cocc82] Cocco, N. and Dulli, S. A mechanism for exception handling and its verification rules. *Computer Languages*, 7 (1982), 89-102.

[Dijk76] Dijkstra, E. W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J. 1976.

[Good75] Goodenough, J. Exception handling: Issues and a proposed notation. *Communication of the ACM*, 18, 12 (Dec. 1975), 683-606.

[Knut74] Knuth, D. E. Structured programming with goto statements. *ACM Computing Surveys* 6, 4 (Dec. 1974), 261-301.

[Lisk79] Liskov, B. and Snyder, A. Exception Handling in CLU. *IEEE Transactions on Software Engineering*, SE-5, 6 (Nov. 1979), 546-558.

[Lisk83] Liskov, B. and Scheifler, R. Guardians and actions: linguistic support for robust, distributed programs. *ACM Transactions on Programming Language and Systems* 5, 3 (July 1983), 381-404.

[Lisk87] Liskov, B., et al. Orphan detection. *Fault Tolerant Computing Symposium*, (July 1987), 2-7.

[Luck80] Luckham, D.C., and Polak, W. Ada exception handling: an axiomatic approach. *ACM Transactions of Programming Languages and Systems*, 2, 2 (April 1980), 225-233.

[Mesa85] *Mesa Language Manual*. Xerox Corporation, 610E00170, Sept., 1985.

[Panz88] Panzieri, F. and Shrivastava, S. Rajdoot: A remote procedure call mechanism supporting orphan detection and killing. *IEEE Transactions on Software Engineering*, 4, 1 (Jan. 1988), 30-37.

[Schl83] Schlichting, R.D. and Schneider, F.B. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*. 1, 3 (Aug. 1983), 222-238.

[Schl87] Schlichting, R., Cristian, F., and Purdin, T. Mechanisms for failure handling in distributed programming languages. TR 87-13, Dept. of Computer Science, The University of Arizona, June, 1987.

[Walk84] Walker, E. Orphan detection in the Argus System, Master's thesis, MIT/LCS/TR-326. Massachusetts Institute of Technology, June, 1984.

[Wile84] Wilensky, R. *LISPcraft*, Norton, New York, 1984.

[Yemi82] Yemini, S. An axiomatic treatment of exception handling. In *Proceedings 9th Symposium on Principles of Programming Languages* (Jan. 1982).

[Yemi85] Yemini, S. and Berry, D. A modular verifiable exception handling mechanism. *ACM Transactions on Programming Languages and Systems*, 7, 2 (April 1985), 214-243.

[Zahn74] Zahn, C. T., Jr. A control statement for natural top-down structured programming. In *Programming Symposium: Proceedings, Colloque sur la Programmation*, Paris, April 9-11, 1974, B. Robinet, ed. *Lecture Notes in Computer Science*, Vol. 19, G. Goos and J. Hartmanis, eds., Springer-Verlag, New York.

TABLE OF CONTENTS

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I am most indebted to my advisor Dr. Ronald A. Olsson. I thank him for the enormous amount of time and effort in listening to my ideas, making many invaluable suggestions, and promptly reading the drafts of this thesis.

I am also grateful to the other members of my committee. Dr.'s Gene Fisher and Karl Levitt have greatly enriched my technical knowledge and their constructive criticisms have been quite helpful in the final phases of writing.

I am also indebted to my fellow graduate students: Sara Kalvala, Carole McNamee, Debbie Frincke, Jim Alves-Foss, Jim Buffenbarger, Joseph Herman, and Teng-Sheng Moh for their suggestions and friendship while this thesis was underway.

Finally, I thank my parents and sister for their support, understanding, and constant encouragement over the years. Without them, my graduate studies would have been impossible.