

# Error-Recovery Language for Internet Systems

Archana Ganapathi

([archanag@cs.berkeley.edu](mailto:archanag@cs.berkeley.edu))

## 1. Internet System Failures

Failures in Internet Systems (IS) reduce availability [ROC, Oppenheimer et al.]. Moreover, the impact of scale on availability is enormous. In the Internet domain, downtimes are intolerable; they bear exceedingly large economic repercussions. Failures as well as errors arise from various causes and root-cause analysis has been a complex predicament.

Given the diverse need to improve availability in Internet Services [Oppenheimer & Patterson], it is valuable to develop a language that helps users specify dependability requirements as well as failure models of system components including hardware, software, networking and human operators. The long-term goal would be to develop compilers that analyze and synthesize this language to generate software tools. Synthesis can be applied by various IS to gauge availability and track degeneration or progress of their component elements.

Recovery code specifies conditions under which error occurrence and recovery occurs; it spans simple programs for basic revival to specialized exception handlers. The most common form of recovery is *ctrl-z* or *undo*. Usually, recovery of the last action is possible if that transaction has not “closed” or “compacted” by the system. Often, recovery code remains partially untested during system execution. Consider the following examples of recovery-code fragments:

```
Function undelete_table                                     --[Lesandrini]
  For Each table_definition In db.TableDefs
    If Left(tdf.Name, 4) = "~tmp" Then
      sTable = tdf.Name
      sSQL = "SELECT [" & sTable & "].* INTO " & sName
      sSQL = sSQL & " FROM [" & sTable & "];"

      db.Execute sSQL

      sMsg = "A deleted table has been restored as " & sName
      MsgBox sMsg, vbOKOnly, "Restored"
      GoTo Exit_Undelete
    End If

*****

Function_recover -- fetch file from ~/backup                --[scssh.net]
(define file-name (absolute-file-name (cadr (command-line))))
(define backup-name
  (cond ((regexp-search (rx (seq "/"home/"(submatch (** 4 8 1-case)))file-
name)
    => (lambda (m)(regexp-substitute #f m 'pre "/"home/" 1 "/"backup"
'post)))
    (else (error "cannot match in " file-name))))
```

```

(if (file-not-exists? backup-name)
  (error (format #f "backup file not found" backup-name file-name)))
(if (file-exists? file-name)
  (begin (format #t "File ~a exists.\n overwrite it? (y or n)" file-name)
    (let ((answer (read)))
      (if (not (eq? answer 'y))
          (error "abort by user choice")))))
(run (cp -pv, backup-name, file-name))

```

\*\*\*\*\*

```

testcaseExit(BOOLEAN bExcept)                                --[Cluey]
{ // error state: bRecoveringFromError() handles special recovery
  REC_ERROR = bExcept;
  if bExcept
  {
    ExceptLog();
    SetupRecoverReporting("testcaseExit", MESSAGE);
  }
  else
  {
    SetupRecoverReporting("testcaseExit", ERROR);
  }
  _RunRecoverBaseState("RecoverTestCaseExit" );
  CheckMemoryLoss();
  if(GetGuiType() != pm)          // if not running under OS/2
  {
    CheckResourceLoss();
  }
  SetupRecoverReporting("OutsideTestcase", RPT_REC_ERROR);
  // Reset REC_ERROR, so that further checks bypass bRecoveringFromError()
  REC_ERROR = False;
}

// if Recover function undefined, general recovery RecoverBaseState invoked
private _RunRecoverBaseState(string funcRBS)
{ // default: RecoverBaseState()
  // overwrite: RecoverScriptEnter(), RecoverTestcaseEnter(),
  RecoverTestcaseExit()

  if CallFunctionIfExists(funcRBS) return ;
  if CallFunctionIfExists("RecoverBaseState") return ;
}

private STRING sRecoverCallingFunction = "";
private REC_MSG_TYPE RecoverReportErrors = ERROR;
SetupRecoverReporting(STRING sFunctionName, REC_MSG_TYPE ReportStyle)
{
  sRecoverCallingFunction = sFunctionName;
  RecoverReportErrors = ReportStyle;
}

string GetRecoverCallingFunction(){return (sRecoverCallingFunction);}

BOOLEAN bRecoveringFromError()    {return (REC_ERROR);}

type REC_MSG_TYPE is enum {ERROR, WARNING, MESSAGE, NONE}
RecoveryMessage(STRING sMessage, LIST list_info optional)
{
  STRING sTempMsg = "Error in {sRecoverCallingFunction}: {sMessage}";
  switch (RecoverReportErrors)
  {

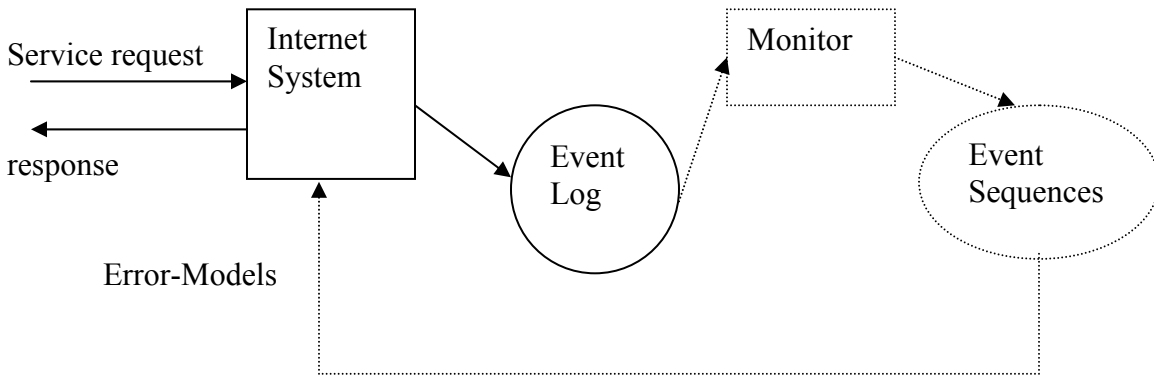
```

```

    case ERROR:      LogError(sTempMsg);
    case WARNING:    LogWarning(sTempMsg);
    case MESSAGE:    Print(sTempMsg);
    case NONE:       return;
    default:         QAPDebugLogWarning("should never be reached");
                    return;
}
if list_info != NULL ListPrint(list_info);
}
}
*****

```

These examples reveal that it is potentially advantageous to consolidate such cumbersome recovery code in a single uniform language, ideally by formal stipulation. Moreover, prescriptive aspects of procedural arrangement can be rephrased as descriptive specifications, removing the implicit ordering of recovery code. Essentially, perusal of recovery code yields error patterns that constitute formal assertions. Recovery code is oblivious of the run-time event sequence that results in errors. Logs are commonly analyzed to identify failures and determine root causes. The result is a list of event sequences that are responsible for system failures. These event sequences form error-model statements.



## 2. Event Logs – Intermediate Representation

Service originates with user requests. Each service request “compiles” into primitive events that are intermediate level operations processed at Internet service collocation facilities. The event-set repertoire comprises of a list of primitive events such as:

```

ping computer_Rout
send message header to computer_X
wait for message_complete from computer_X
notify user of service_completed
client_send_req_to_primary
primary_broadcast_reqs_to_all_backups
replicas_commit_reply
client_wait_for_identical_replies_from_replicas
return_service_result

```

These intermediate-level operations resemble primitive instruction-set calls; for example, consider message attributes tagged by Pinpoint [Kiciman]. When an instruction does not

complete execution, usually by time-out, a failure occurs. A failure is detected by a raised exception or invocation of an exception handler.

Bug triggers arise from timing, boundary conditions, as well as fault injections and recovery code. Prominent data-corrupting bug triggers arise from boundary conditions, bad recovery code, interaction with bug patches (predominantly wide-impact) and timing (predominantly induce-reboot). Prominent data-corrupting bug triggers are boundary conditions, interaction with bug-patch (predominantly wide-impact), bad recovery code, timing (predominantly induce-reboot) as well as *bohrbugs*.

Interleaving of events lead to a large number of execution traces; it is almost impossible to test these traces. Efficient analysis is essential to identify the “cone of logic” leading to failure.

### 3. Dependence Analysis

Dependence analysis on the event-graph produces root-cause failure chains. Initially we concentrate on errors (system states that lead to failure) ignoring failures (deviation of a system from its specification). Detection (discovering the presence of a fault) reduces to identification of failure nodes in the event log. From the event log, graphs are constructed for each service request. The nodes in the graph represent primitive events. Edges in this graph represent dependence or interference between events. Starting with exceptions, which are failure nodes in event graph, we perform an edge traversal that traces a path to the root fault (deemed cause of error and failure). The cone of logic for failure confinement is then analyzed to produce sequences of events that potentially induce failures. A root to “leaf” (failure node) path identifies an event sequence, a temporal pattern. Such sequences are noted as potential causes for Internet system failures. Temporal patterns have associated action code forming error-model statements. Error-model statements can now substitute recovery code. These sequences formulate assertions that are input to the IS to improve dependability as well as availability.

### 4. Root-cause failure chains: Error-model description

An error state is modeled to check all system invariants. Error-model creation is essentially a two-step process representing recovery code with corresponding formal assertion:

- assertions: data-mined list of variables and functions relevant to properties to be checked
- error-recovery rewrite actions: code that checks correctness or abstracts error-model details

Root-cause failure analysis produces error-model statements. An error-model description consists of a series of rules; rule bodies are executed when the rule precondition is true

rule:  $pre\_condition \implies rule\_body$

Pre\_conditions are temporal sequences. A rule\_body comprises action handlers that are invoked upon error occurrence. Essentially, the procedural nature of the original recovery code is now replaced with a descriptive pattern that is suitable for pattern matching. For example,

```
!S.a && S.b ==> { assert !S.c; assert S.d; S.a = true; function_x(a, b);}
-- match pattern, S.a && S.b, and use RHS as action for output
```

Error\_model statement:

```
event_a followed_by event_b && !event_c ==> exception_handler_x(S.a, S.b)
-- match pattern, event_a followed by event_b, as enabling conditions for
error handling
-- RHS (exception_handler_x) as system action for error-recovery.
```

*exception\_handler\_x* specifies error handlers for exceptions such as

- Receipt of an unexpected (error) message
- Message loss
- Timeout of event
- Detection of link failure
- Invalid response
- Node reboot

## 5. Formalization of Error-Models

Improving reliability, formal methods offer tools to specify the behavioral and structural characteristics of a system [Clarke & Wing]. The idea is to formally prove agreement between system and requirement specifications. Although such techniques cannot guarantee properties of a real system including hardware and all software layers, they verify the correctness of the specification. Using model-checking or theorem-proving techniques, they eliminate ambiguity wherever possible. They can prove the presence of bugs but not their absence [Dahl et al.] and thus they do not eliminate testing. Detected bugs include uninitialized variables to subtle logic errors. For example, the Java Byte-code Verifier uses theorem proving at run-time to establish type preservation, access restrictions and legitimate pointers. Similarly, in a Proof-Carrying Code (PCC) system, the producer creates a safety proof for the code receiver to validate, thus, enabling safe execution of otherwise untrusted machine-code [Necula & Lee].

Program analysis can verify a program's expected properties. Data-flow analyses and domain-specific rule checks verify correctness of code at compile time. By tracking states of variables it is possible to identify bugs such as accessing free or uninitialized memory and out of bound arrays [Purify, Engler et al.]. Data-flow analyses can transitively track data sources to plug security holes during attacks, preventing user from accessing forbidden information. When compared to formal methods, program analysis need not understand or execute program code; it can analyze all execution paths at compile time.

However, checks are “local” and usually incomplete. It attempts to predict and detect problems that result in failures, often yielding false positives especially when analysis is aggressive.

The behavior of a system can be specified in a formal language. Formal descriptions specify functional and temporal behaviors as well as performance expectations of software. Specifications involve states, legal operations and state transitions similar to function signatures in a programming language. Operation description is implicit in the case of axiomatic specifications that define pre and post properties for each operation, invariants appearing when these conditions are alike. Model-based specifications represent the system abstractly; they involve set and function-theoretic approaches as well as logic [Clarke et al.]. Model checkers are completely automatic, systematically enumerating the possible states of a system to exhaustively test code for all inputs, e.g., Murphi [Dill et al.]. Unlike static analysis, they verify high-level and global properties, e.g., data invariants. They include temporal model-checks or behavioral conformance confirmation to verify properties under consideration as well as provide state-machine execution for violation of user-specified invariants. Systematic and exhaustive exploration of the system “state-space” increases computational complexity, as the state-graph is exponential in the size of the program. This process is exhaustive but guaranteed to terminate as the model is finite.

To deal with unbounded number of states, *theorem provers* demonstrate correctness by generating proofs through structural induction and fancy pattern matches. They attempt to prove formal properties that are specified through logic formulae, axioms and inference rules appearing as user suggested lemmas; when unsuccessful, they expose problems in requirement specifications for the modeled system. Axioms or hypotheses are *propositions* and *consequences* derived from them are *theorems*. *Resolution* is a rule of inference that specifies how one proposition follows from others. Using this resolution principle, theorem proving is automated, e.g., a Prolog system is a linear (*Horn* clauses) resolution theorem prover [Clocksin & Mellish].

Encoding domain-specific knowledge into formal specifications is cumbersome, making complete specification of complex systems infeasible. Modern software API definitions contain English prose, e.g., Request For Comments (RFC) for Internet protocols. Several technical challenges occur when attempting to translate English prose into machine-processable specifications. [Crow & Di Vito] describe Functional Subsystem Software Requirements (FSSR) for space-shuttle software. Requirement specifications reveal strong implementation bias, making formalization difficult. When translating “English” requirements to PVS/Murphi specifications, they encountered several obstacles:

- state variable inference from algorithms and clues
- variables absent in requirement specification introduced as pseudo-inputs/outputs
- requirements written in procedural style allowed for variables to be in undefined states
- formalization for machine verification needed new mathematical formalisms, such as finiteness, injectivity, surjectivity, and left/right inverses.

- When domain knowledge for formal specification was absent help from domain experts solicited
- Input-variable proliferation and complexity required artificial abstraction and simplification
- physical domain size and complexity complicated modeling, initially, too large for Murphi to compile

Recovery code is indispensable to improve reliability. Authoring and testing such code is complex. Usually, when such code is invoked, the system transitions to an abnormal state. Testing such atypical states is tedious as they are numerous, far exceeding normal states. Formal methods carry significant potential to improve code reliability perhaps reducing recovery code. In addition, by “pre-compiling” erroneous patterns, “unusual” states can be avoided, effectively trading “state-space” for pattern matches.

We develop formal models for errors in ISs to enable automatic configuration-verification. This scheme can potentially uncover pitfalls in configuration APIs, identify differences in semantics of states among various collocation facilities and also detect inconsistencies in distributed-system configuration specifications. Upon changes to configurations, IS administrators invoke LISA verification modules to authenticate changes; consequently, each node is notified of invariance violations as well as assertion failures.

## **6. LISA Specification**

LISA is an {interoperable} language to improve availability in Internet Service Architectures (ISA). It is a declarative language to accurately and concisely specify the IS properties to be tested, verified and measured to sustain a high degree of availability. Moreover, it facilitates seamless integration of IS applications as well as distribution of IS Intellectual Property (IP). This language enables an assertion-based methodology for improving checkable specifications and IS configuration properties. Assertions describe undesirable behaviors or expected behaviors that are required to maintain availability. These assertions are checked in either dynamic Internet service or formal verification. Moreover, a library of reusable IP modules, which are assertions grouped together to make a generic checker, supports this methodology. They accelerate the development of a complete IS environment, improving overall efficiency and productivity for applications to scale with improving availability. We introduce high-level declarative constructs that are intended to easily create and maintain concise dependence-analysis code, pinpoint configuration entities that need to be fixed, and measure system quality. It is designed for checking compliance with widely adopted and emerging configuration standards. Systematic checking ensures that IS availability remains at a very high level. Configuration designers create assertions using LISA mechanisms to verify compliance with overall system architecture. This aspect includes configuration completeness, i.e., coverage of functional IS specifications. Coverage indirectly affects availability.

Configurations can no longer be sufficiently verified by ad-hoc testing and monitoring methodologies. This process of working through configuration bugs can cause defects in the configurations themselves. Such difficulties arise because there exists no effective means of specification for exercise and verification against intended IS functionality. The central focus of configuration checks is assertions, which detect bugs, guide configuration authors to remove faults and direct fault-injection modules to produce stimuli for system verification. Assertion-based techniques support two methods: dynamic verification using fault-injection, and formal verification using model checking and proof techniques. LISA provides language constructs to build a hierarchy of concise temporal expressions representing events, transactions and sequences of these entities. Internet systems administrators write assertions to ascertain that illegal behavior is trapped and notified as an error. System testing is essential to uncover bugs resulting from component interactions. During this process, it is beneficial to monitor interesting activities that provide guidance into potential faults or simply poor configurations that reduce availability. These activities, temporal in nature, are sequences of events or data values that occur during a specific period. Often, related statistics gathered as counts or frequency of occurrence, can provide additional insights.

In Internet services, temporal and order related functional problems are difficult to detect and carry the most expensive time to repair (TTR), causing unpredictable delays and enormous cost overruns. For example, it is insufficient to know that an event that arrived at the input was received at the output within the constraints set forth by the Internet service. This transit is associated with several stages of transformation and state changes before service results at the output port. Functional coverage provides more detailed knowledge of combinations of stages that are exercised to assure dependability. Consequently, there arises a necessity to accurately and concisely describe behaviors that span over infinite clock cycle transitions. “State” activities such as input/output behavior, inter-component interactions as well as perturbations in system state arise from complex relationships in the architecture. These characteristics necessitate formal expressions for sequencing, invariants and state machine operations. The following summary of language features considers commonly applicable situations:

- IS events and transactions (events accompanied by state changes)
- sequencing to specify the order of events and transactions
- temporal sequences with references to past and future
- logic connectives such as and, or, not operators
- repetition of sequences, concatenation and overlap of sequences
- implication, conditional sequences with temporal con-sequences

Another important feature of LISA is its ability to facilitate re-usable specifications as facilitated by parameterizing assertions via template classes (as in C++) and building a library of re-usable descriptions. These IP can be used multiple times in different configurations. They guarantee the quality of pre-built components. Thus, quality concerns regarding availability can largely be pacified if not eliminated. Dependence-analysis software is built into the Configuration Checker to provide instant reporting of violations when any assertion is found to conflict with the expected behavior. Complex

algorithms to monitor assertions work in conjunction with dependence analysis to monitor events, as they occur to determine assertion results.

This methodology supports both “black box” (oblivious of the implementation) and “white box” testing by expressing both types as assertions in LISA. In “white box” testing, knowledge of the system architecture is used to perform assertion-based verification. Fault injection provides stimuli at the inputs and assertions check their response at the outputs. The relationships between inputs and outputs of the system are expressed as LISA events. LISA specification remains independent of the internals of the system and is associated with the system without any modification although the system itself may scale dynamically. Internet System administrators check for the correct system behavior, and write assertions (checks) to detect system-level faults. In particular, checking the adherence of interface components to interface standards is important as these seams often hide complicated errors.

The coverage capability monitors the success or failure of each assertion; simply knowing whether an assertion was exercised is insufficient. The assertion may succeed in multiple ways by taking certain branches within the assertion. Each path covers a particular execution scenario. Functional coverage enumerates these paths and monitors them. At the end of the property check, functional feedback provides the taint of enumerated paths to reveal the extent of assertion exercise. In principle, integrity constraints are policed by a “policy compiler” to yield triggers and adaptation-code templates that form the basis of “reaction algorithms”. LISA capability becomes “reactive” when it modifies *transaction control* that is already modeled by the associated assertions. In this case, its re-actions must be integrated with checkpoint management.

## 6.1 Language Syntax & Semantics

LISA specifications permit automatic checks for IS dependability. Specifications are provided as part of system configuration files and automatic checks are invoked upon component deployment in an ISA. This language is evolving, operators for specification have been borrowed from C++ [Stoustrup], Verilog [Moorby], Vera [Synopsys], Specman [Verisity], CadenceTest [Cadence] and PSL[Accelera].

*ISA\_events* are externally observable system events that are specified by Internet-System service management in configuration specifications. An event is an atomic operand that participates in LISA operations.

*ISA\_expressions* are Boolean compositions involving *ISA\_events*; they contain *ISA\_events* and *ISA\_logical\_operators* only. Expressions evaluate to True or False.

*ISA\_transactions* are sequences of *ISA\_expressions*; they are enclosed in *{}*; they contain *ISA\_expressions* and *ISA\_sequential\_operators*, including extended regular expressions. Event expressions describe temporal sequences of events.

*ISA\_clock\_cycle* is the time period of the “system clock”, a time unit specified by Internet-System service management in configuration files. This time cycle can represent days, hours, minutes or seconds.

*ISA\_Properties* describe the behavior of relationships such as *ISA\_events* that relate to system processes or data, both transient as well as persistent. These associations specify logical and temporal relationships among Boolean or sequential expressions and also other properties that cumulatively verify system behavior. Properties can be temporal or non-temporal as they relate to processes, data or IS architecture requirements such as restrictions on the number of nodes that belong to a category, e.g., consider Byzantine architecture with  $n$ -nodes and  $f$ -nodes [Castro & Liskov].

### *ISA\_Operators*

Logical:

&     -- and  
|     -- or  
~     -- not

Sequential:

;     -- concatenation  
:     -- overlap

Implication:

->    -- logical if or sequential implication  
<->   -- logical iff implication  
=>    -- temporal ‘next’ implication

Extended Regular Expressions

\*     -- 0 or more repetition  
+     -- 1 or more repetition  
?     -- optional  
[ ]    -- count qualifier

### *ISA\_Assertions*

Assertions validate IS behavior; they verify architecture and functionality invariance as well as dynamic implications. Examples of transactional assertions in IS functionality include data persistence and event-transaction relations such as “Once Event\_A is asserted, it must remain asserted until the transaction is complete”. Examples of architecture properties include Byzantine fault tolerance, checking if  $n > 3f$  always holds [Castro & Liskov] and in general deducing conformity in system structure and dependencies. Violations of macro invariants include implementation interface bugs or signs of system intrusion, such as “*upper bound on the number of hops made during message delivery on a P2P system*” [Chen et al.]. Similarly, consider the following perfect failure detector protocol for completely synchronous systems [Fetzer]; to verify the status of a system component  $c$ , a configuration process asserts function  $ISA\_f(c) == "up"$ .

```
function ISA_f (component c)
```

```

    {send ping to c; wait on receive pong from c return "up"; after
    2*τ return "crashed";}
always (on receive ping from sender send pong to sender);

```

Similarly, *ISA\_Assertions* discover erroneous configurations or system un-availability such as *component\_out\_of\_message\_bounds*, *component\_uninitialized\_use* and *component\_micro\_reboot*.

Consider the following example specifications and their formal representation in LISA.

- Invariance: *ISA\_Event a or ISA\_Event b must prevail.*  
`assert always {a | b} @ISA_clk;`
  - Sequential invariance: *ISA\_Event a must be immediately followed by ISA\_Event b followed by 1 to 3 occurrences of ISA\_Event c.*  
`assert always {a; b; c[1..3]} @ISA_clk`
  - Implication: *if ISA\_Event a or ISA\_Event b occurs then two occurrences of ISA\_Event c must immediately follow.*  
`assert always {a | b} => {c; c} @ISA_clk`
  - *Whenever ISA\_Event "a" occurs, ISA\_Event "b" must occur within 1-3 ISA\_cycles later, and event "a" should remain active until event "b" occurs.*  
`assert always (a => b[1..3] & a) @ISA_clk`
  - *If a is True intermittently or continuously for 3 ISA\_cycles then after that b must be True within 4 ISA\_cycles, unless c happened in the meantime.*  
`assert always (a[1..3]) => b[1..4] | c @ISA_clk`
  - *Network property to guarantee "free of routing loops": at most one entry in table, count less than number of nodes in network.*  
`assert always {(seqa < seqb) - (seqa = seqb ^ hop_cnta > hop_cntb)}`
  - IP Network Configuration: Inconsistent definitions specified in router configuration language or arising from global definitions (e.g., OSPF default weight/cost), dependence of configuration options on default parameter settings (e.g., operator neglecting to assign OSPF weight), dependence on the order of entries in configuration options, and inconsistent references to remote nodes:  
*permit a.b.c.x.x.x.y followed by deny a.b.c.z.x.x.x.d makes deny ineffective as packets match permit in access list – the operator may have intended deny to precede permit*  
`assert never (permit a.b.c; [*]; deny a.b.c) @ISA_clk`
- dependencies across files: two configuration files referring to the same item must have the same view of the item*  
`assert always(neighbor a.b.c.d remote-as number) @ISA_clk`

- *Network protocol verification, e.g., SONNET verification*
- Programming Language specification rules (e.g., informal rule for the C *stdio* library):  
*A file must be opened before a read/write operation or closed; it cannot be used once closed.*  
`assert always (x = fopen("name", ""); [*]; fclose(x)) @(ISA_clk)`
- Other examples:  
`assert always (a => {b} | {c}) @(ISA_clk)`  
`assert always ((a != x) -> never ({b} => {c})) @(ISA_clk);`  
`assert always (a -> next (b until c)) @(ISA_clk);`  
`assert always (a before b) @(ISA_clk);`

### ***ISA\_IPS***

ISA Assertion verification Intellectual Properties (IPs) are reusable verification and checker modules that monitor the availability of a comprehensive ISA environment.

### **Syntax**

```

LISA_Statement      ::= Assertion Action

Action              ::= ==> { <ok message> , <recovery code> }

Assertion           ::= assert Property @ ISA_clk ;

Property            ::= Sequential_Expression
                    | Logical_Expression
                    | Temporal_Operation

Sequential_Expression ::= Boolean
                    | Sequential_Expression [
Regular_Expression ]
                    | { Sequential_Expression }
                    | Sequential_Expression :
Sequential_Expression
                    | Sequential_Expression ;
Sequential_Expression
                    | Sequential_Expression &
Sequential_Expression
                    | Sequential_Expression |
Sequential_Expression

Logical_Expression  ::= ~ Property
                    | Property & Property
                    | Property | Property
                    | Property -> Property
                    | Property <-> Property
                    | Property ==> Property

Temporal_Operation  ::= always Property
                    | never Property
                    | next Property

```

		Property <b>before</b> Property
		Property <b>until</b> Property
Regular_Expression	::=	* Range
		+
		?
Range	::=	$\epsilon$
		Number
		Number .. Number

## Semantics

The semantics is defined by a model represented by the triple  $\langle A, F, S \rangle$ .

A is a non-empty set of atomic propositions.

S is a finite set of states.

F is a function that maps each state from S to the alphabet  $2^A$ , with a set of valid atomic propositions.

$$F: S \rightarrow 2^A$$

$f \models b$  Boolean expression b holds under truth assignment represented by f

$f \models b$	$\Leftrightarrow$	$b \in f$
$f \models \neg b$	$\Leftrightarrow$	$f \not\models b$
$f \models b_1 \ \& \ b_2$	$\Leftrightarrow$	$f \models b_1$ and $f \models b_2$
$f \models b_1 \   \ b_2$	$\Leftrightarrow$	$f \models b_1$ or $f \models b_2$

$w \models r$  Word w is in the language of the extended regular expression r

$$w = f_0 f_1 \dots f_{n-1} \text{ has length } n, |w| = n$$

$w \models b$	$\Leftrightarrow$	$ w  = 1$ and $f_0 \models b$
$w \models r_1 ; r_2$	$\Leftrightarrow$	$\exists w_1, w_2$ s.t. $w = w_1 w_2, w_1 \models r_1, w_2 \models r_2$ (note: $\exists$ is used as a symbol for "there exists")
$w \models \{r_1\} \ \& \ \{r_2\}$	$\Leftrightarrow$	$w \models r_1$ and $w \models r_2$
$w \models r[*]$	$\Leftrightarrow$	$w = \epsilon$ or, $w = w_1 w_2 \dots w_n$ , foreach $i \in [1..n]$ $w_i \models r$

## 7. Examples

We present examples of converting system requirements from simple English to LISA. LISA specifications can be further compiled into Verilog, allowing us to create tools to actively monitor the system's behavior.

### 7.1 System Ping-Pong

#### *IS-dictation*

Within 1 to 3 ISA\_cycles after ISA\_event *ping* occurs, ISA\_event *pong* must occur

#### *LISA-statement*

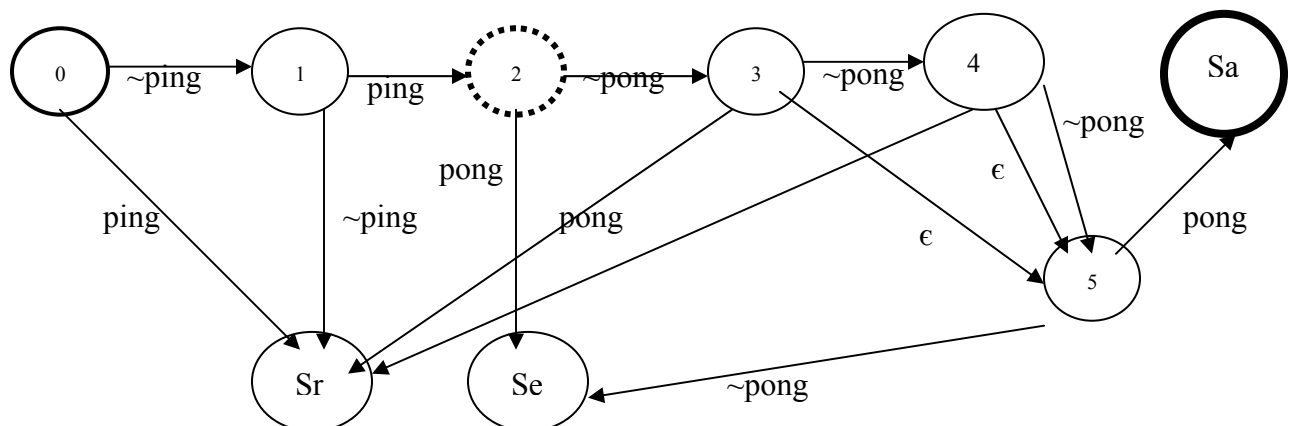
```
assert always {~ping; ping} -> {~pong[1..3]; pong} @(ISA_clk);
```

#### *Verilog program (hand-written; non state-machine model)*

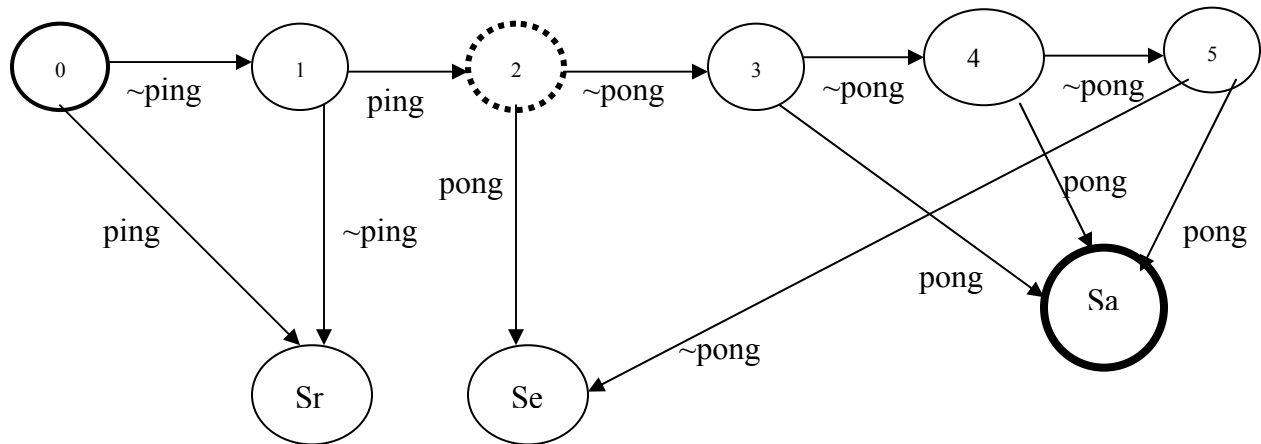
```
always @(ping)
begin
  repeat (1) @(ISA_clk);
  fork: P
  begin
    @(pong);
    $display($time, "Computer up");
    disable P;
  end
  begin
    repeat (2) @(ISA_clk);
    $display($time, "Computer crashed");
    disable P;
  end
end
join
end
```

#### *System state-machine*

Non-deterministic Finite State Automaton



## Deterministic Finite State Automaton



### Verilog program

```

module state_machine(ISA_clk, in, ISA_reset, out);

input ISA_clk, in, ISA_reset;
output [3:0] out;
reg [3:0] out;
//
//          +----- out3
//          |----- out2
//          ||----- out1
// extra bit ---+ |||----- out0
//          | |||
//          V VVVV
`define S0 5'b0_0000 // all bits set to 0
`define S1 5'b0_0001 // extra bit == 0
`define S2 5'b0_0010
`define S3 5'b0_0011
`define S4 5'b0_0100
`define S5 5'b0_0101
`define Sa 5'b1_1000 // accept_state
`define Sr 5'b1_1001 // reject_state
`define Se 5'b1_1010 // error_state

// parameter
zero=0,one=1,two=2,three=3,four=4,five=5,six=6,seven=7,eight=8;

// Declare state flip flops
reg [4:0] state;

// Continuous assignment of state bits to outputs
assign {out3, out2, out1, out0} = state[3:0];

```

```

// Define State Transitions
//
always @(ISA_clk or ISA_reset)
begin
    if (ISA_reset)
        state = `S0;
    else
        case (state)
        `S0: if (!ping)
            state = `S1;
            else
                state = `Sr;
        `S1: if (ping)
            state = `S2;
            else
                state = `Sr;
        `S2: // transient state
            if (!pong)
                state = `S3;
            else
                state = `Se;
        `S3: if (!pong)
            state = `S4;
            else
                state = `Sa;
        `S4: if (!pong)
            state = `S5;
            else
                state = `Sa;
        `S5: if (pong)
            state = `Sa;
            else
                state = `Se;
        default:
            state = `S0;
        endcase
end
endmodule

```

### ***ISA deployment Run-time***

Consider  $ISA\_clock = 2 \cdot \tau$

```

τ      ping = 0 pong = 0
3*τ    ping = 1 pong = 0
5*τ    ping = 0 pong = 1
7*τ    ping = 1 pong = 1
9*τ    ping = 0 pong = 0
11*τ   ping = 1 pong = 0
13*τ   ping = 0 pong = 1
15*τ   ping = 1 pong = 1
17*τ   ping = 0 pong = 0
19*τ   ping = 1 pong = 0

```

\*\*\* assertion failure 5\*τ ► 7\*τ

\*\*\* assertion failure 13\*τ ► 15\*τ

## 7.2 Property validity duration

### *IS-dictation*

Check if an ISA\_expression *exp* remains true for at least a minimum number of ISA\_clock\_cycles *min* but no longer than a maximum number of ISA\_clock\_cycles *max*

### *LISA-statement*

```
assert always {~exp; exp} -> {exp[*min]} @(ISA_clk);
assert always {exp[*max]} => {~exp} @(ISA_clk);
```

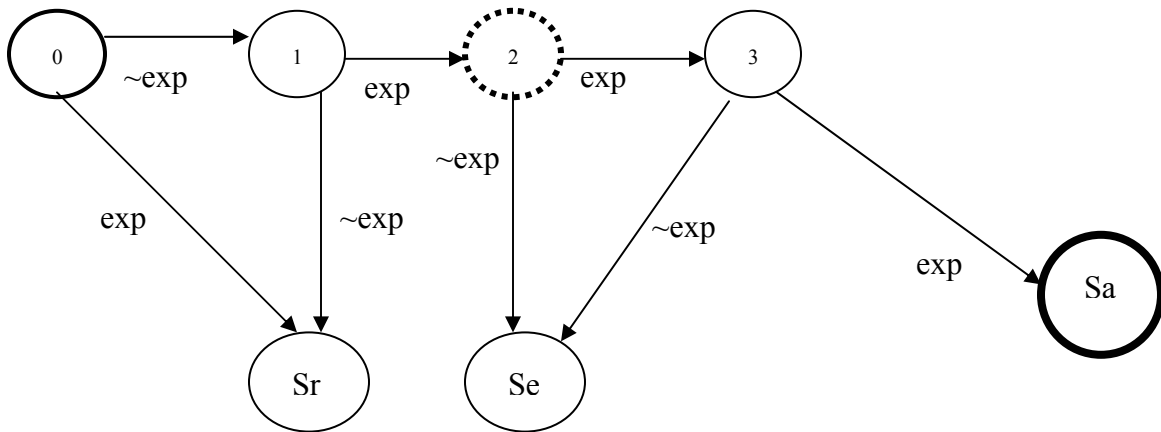
### *Verilog program (hand-written)*

```
always @(ISA_clk) begin
  exp1 <= exp;
  case (state)
    START:
      if(!exp1 && exp) begin
        num_cycles <= 1;
        if(min_cycles > 0) state <= CHECK_MIN;
        else if(max_cycles > 0) state <= CHECK_MAX;
      end
    CHECK_MIN:
      if(~exp) begin
        num_cycles <= num_cycles + 1;
        if(num_cycles >= min_cycles) begin
          if(max_cycles > 0) state <= CHECK_MAX;
          else state <= IDLE;
        end
      end
      else begin
        if(num_cycles < min_cycles) $display($time,,"Expression
Volatile");
        state <= START;
      end
    CHECK_MAX:
      if(exp) begin
        num_cycles <= num_cycles + 1;
        if(num_cycles > max_cycles) $display($time,,"Expression
Volatile");
        state <= IDLE;
      end
      else begin
        if(num_cycles > max_cycles) $display($time,,"Expression
Volatile");
        state <= START;
      end
    IDLE:
      if(!exp) begin
        state <= START;
      end
  endcase
end
```

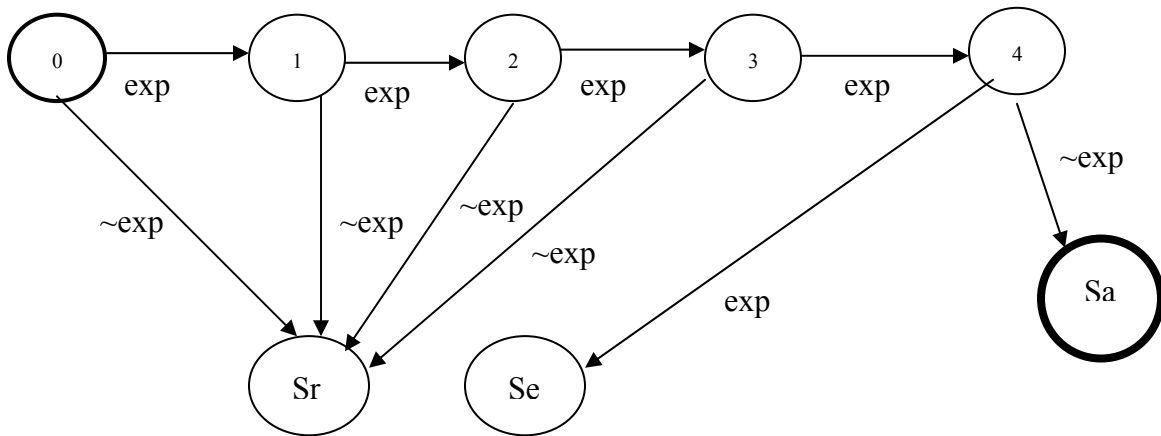
*System state-machine*

Deterministic Finite State Automata

$min = 2$



$max = 4$



## 8. Related Research

Assertion based languages have been used to automate debugging. ANNA is an annotation language for ADA supporting assertions for variable and type declarations [Luckham et al.]. Similarly, follow-on specification languages for ADA, TSL [Luckham\_2 et al, Luckham\_3 et al.] and Rapid [Luckham\_4 et al.] introduce the notion of events to describe task behavior. Assertions, written in ADA, are checked dynamically at run-time, obviating the need for post-mortem analysis. [Rosenblum] describes an annotation language for the programming language C. [Baiardi et al.] introduce events that describe process communication and connection to channels. Behavioral expressions describe sequences of events with run-time assertion evaluation. Assertions in temporal interval logic include assignment to variables, label reachability, interprocess communication and process instantiation/termination [Goldszmidt et al., Goldszmidt\_2 et al].

Several researchers have concentrated on data-mining large numbers of bug reports produced by verification tools. They attempt to eliminate false-positives and minor bugs that are inconsequential. The Xgcc system uses statistical heuristics, using clustering, to rank severe, hard-to-find bugs, above others [Hallem et al.]. PREFIX uses ranking and filtering heuristics to reduce “noise” [Bush et al.]. Daikon dynamically detects arithmetic invariants using statistical confidence checks to suppress precarious invariants [Ernst]. Houdini [Flanagan & Leino] guesses invariants and then ESC/Java [Flanagan et al.] prunes those that cannot be verified. [Nimmer & Ernst] use a similar approach that integrates Daikon and ESC/Java relying more on the user to help debug specifications.

Strauss is a specification miner that uses the *sk-string* algorithm [Raman & Patrick] to infer finite automata in specification<sup>1</sup> and partially automate temporal property specifications [Ammons et al.]. [Ammons\_2 et al] describe a general method to debug formal temporal specifications that exploit small program execution traces generated by program verification tools from specification-mined violations. They use concept analysis, traversing small concept lattices, to cluster violation traces. Similarly, ESP is a scalable temporal property verifier that works on specifications with one variable only [Das et al.]. Other temporal-property verifiers include SLAM that verifies properties expressed as state machines in SLIC, a variant of C [Ball & Rajamani] and Berkeley Lazy Abstraction Software Verification Tool (BLAST) similar to SLAM, is a model checker for C based on lazy predicate abstraction; it adopts lazy evaluation to improve performance [Henzinger et al.]. Intrusion detection systems are based on automata that match intrusion attack scenarios, e.g., USTAT [Illgun et al.] and IDIOT [Kumar & Spafford] that implements pattern matching as unification using colored Petri nets [Jensen, Kristensen et al.].

---

<sup>1</sup> Other algorithms are surveyed by [Murphy]

## 9. Future Work & Conclusions

The design of LISA is rapidly evolving. Our next course of action is to write a LISA to Verilog compiler. The following techniques can improve specification as well as implementation to produce improved Verilog output.

- Transformations

<code>a &amp; b</code>	<code>≡</code>	<code>~(~a   ~b)</code>
<code>a -&gt; b</code>	<code>≡</code>	<code>(~ a)   b</code>
<code>a &lt;-&gt; b</code>	<code>≡</code>	<code>(a -&gt; b) &amp; (b -&gt; a)</code>
<code>a before b</code>	<code>≡</code>	<code>always {~b} until a</code>

- Merging DFAs into a common DFA (consider example 2)  
It is advantages to produce a composite DFA produced by merging assertions with shared patterns.
- Diagnosis (identification of failed components, e.g., Pinpoint) was not an objective; perhaps our module can co-operate with Pinpoint [Kiciman].

Based on further analysis of Internet service event logs and execution traces, we would like to extend the LISA vocabulary and functionality. We would like to get feedback from system administrators about LISA's ability to express their configuration intents. We hope to deploy our system and observe its behavior at an active Internet service. Perhaps we can also consider incorporating a dynamic "learning" phase where use statistical learning theoretic techniques to extract event sequences from an active system.

## 10. Bibliography

- [Accelera] PSL Reference Manual, Version 1.01  
[www.accellera.com](http://www.accellera.com)  
April 25, 2003
- [Ammons et al.] Ammons, G., Bodik, R., and Larus, J.  
Mining Specifications with Strauss.  
PLDI 2002.
- [Ammons\_2 et al.] Ammons, G., Bodik, R., Mandelin, D. and Larus, J.  
Debugging Temporal Specifications with Concept Analysis.  
PLDI 2003, June 9-11, 2003, San Diego, California.
- [Baiardi et al.] Baiardi, F., De Francesco, N., and Vaglini, G.  
Development of a Debugger for a Concurrent Language,  
IEEE Transactions on Software Engineering, vol. SE-12, No 4, pp.547-553,  
April 1986.
- [Ball & Rajamani] Ball, T. and Rajamani, S.K..  
Automatically Validating Temporal Safety Properties of Interfaces.  
Software Productivity Tools,  
Microsoft Research.
- [Bush et al.] Bush, W.R., Pincus, J. D., and Sielaff, D.J.  
A static analyzer for finding dynamic programming errors.  
*Software - Practice and Experience*, 30:775-802, 2000.
- [Cadence] Cadence Testbench Tools  
[www.cadence.com](http://www.cadence.com)
- [Castro & Liskov] Castro, M. and Liskov, B.  
Practical Byzantine Fault Tolerance  
Proceedings of the third OSDI, 1999.
- [Chen et al.] Chen, M., Kiciman, E, Accardi, A., Fox, A. and Brewer, E.  
[Using runtime paths for macro analysis](#),  
*Proc. 9th Workshop on Hot Topics in Operating Systems*, May 2003
- [Clarke & Wing] Edmund M. Clarke and Jeannette M. Wing.  
Formal methods: State of the art and future directions.  
*ACM Computing Surveys*, 28(4), December 1996.
- [Clarke et al.] E.M. Clarke, O. Grumberg, and D. Peled.  
Model Checking. MIT Press, 1999.
- [Clocksin & Mellish] Clocksin, W.F. and Mellish, S.M.  
*Programming in Prolog*  
3<sup>rd</sup> ed. New York, Springer-Verlag. 1987.
- [Cluey] [http://www.sqa-test.com/mr\\_cluey/recover.inc.txt](http://www.sqa-test.com/mr_cluey/recover.inc.txt)
- [Crow & Di Vito] Judith Crow and Ben DiVito.  
Formalizing space shuttle software requirements: Four case studies.  
*ACM Transactions on Software Engineering and Methodology*, 7(3), July 1998.

- [Dahl et al.] Dahl O.J, Dijkstra E.W., and Hoare C.A.R.  
*Structured Programming*. Academic Press, 1972.
- [Das et al.] Das, M., Lerner, S. and Seigle, M..  
ESP: Path-Sensitive Program Verification in Polynomial Time.  
PLDI, Berlin, 2002.
- [Dill et al.] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang.  
Protocol verification as a hardware design aid.  
In IEEE International Conference on Computer Design: VLSI in Computers and Processors, pp. 522-525, 1992.
- [Engler et al.] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem.  
Checking system rules using system-specific, programmer-written compiler extensions.  
In *Proc. 4th USENIX Symp. on Operating Systems Design and Implementation*, San Diego, CA, Oct 2000.
- [Ernst] Ernst, Michael D.  
*Dynamically Discovering Likely Program Invariants*.  
PhD thesis, University of Washington, August 2000.
- [Fetzer] Fetzer, C.  
[Perfect Failure Detection in Timed Asynchronous Systems](#),  
*IEEE Transactions on Computers*, February 2003.
- [Flanagan & Leino] Flanagan, C. and Leino, K.R.M.  
Houdini, an annotation assistant for ESC/Java.  
In *International Symposium on FME 2001: Formal Methods for Increasing Software Productivity*, LNCS, volume 1, 2001.
- [Flanagan et al.] Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.  
Extended static checking for Java.  
In *Proceedings of the ACM SIGPLAN 2002 PDLI*, pages 234-245, June 2002.
- [Goldszmidt et al] Goldszmidt, G., Katz, S., Yemini, S.  
Interactive Blackbox Debugging for Concurrent Languages  
SIGPLAN Notices vol.24, No 1, pp. 271-282, 1989.
- [Goldszmidt\_2 et al] Goldszmit, G., Yemini, S., and Katz, S.  
High-Level Language Debugging for Concurrent programs  
ACM Transactions on Computer Systems, vol.8, No.4, November 1990, pp.311-336.
- [Hallem et al.] Hallem, S., Chelf, B., Xie, Y., and Engler, D.  
A system and language for building system-specific, static analyses.  
In *Proceedings of the ACM SIGPLAN 2002 PLDI*, pages 69-82, May 2002.
- [Henzinger et al.] Henzinger, T.A., Jhala, R., Majumdar, R., and Sutre, G.  
Lazy Abstraction  
In ACM SIGPLAN-SIGACT Conf. on Principles of Prog. Languages, Portland, Oregon, 2002.
- [Illgun et al.] Ilgun, K., Kemmerer, R., and Porras, P.

- State Transition Analysis: A Rule-Based Intrusion Detection Approach.  
IEEE Transactions on Software Engineering, 1995.
- [Jensen] Jensen, Kurt.  
An Introduction to the Practical Use of Colored Petri Nets.  
Department of Computer Science, University of Aarhus, Denmark, 1996.
- [Kiciman] Kiciman, E.  
Progress on Pinpoint, And Integrating It with JAGR/ROC-2  
ROC Retreat, June 4-6, Chaminade at Santa Cruz.
- [Kristensen et al.] Kristensen, L.M., Christensen, S., and Jensen, K  
The Practitioner's Guide to Coloured Petri Nets.  
International Journal on Software Tools for Technology Transfer, 2 (1998),  
Springer Verlag, 98-132.
- [Kumar & Spafford] Kumar, S, and Spafford, E.  
A Pattern-Matching Model for Misuse Intrusion Detection.  
Proceedings of the 17th National Computer Security Conference, 1994.
- [Lesandrini] Lesandrini, D.  
How to Recover a Table Deleted from an Access Database  
<http://www.databasejournal.com/features/msaccess/article.php/1494111>  
Nov 20, 2000.
- [Luckham et al] Luckham, D.C., Sankar, S., and Shuzo Takahashi, S.  
Two-Dimensional Pinpointing: Debugging with Formal Specifications.  
IEEE Software, January 1991, pp.74-84.
- [Luckham\_2 et al] Luckham, D.C., Bryan, D., Mann, W., Meldal, S., and Helmbold, D.P.,  
An Introduction to Task Sequencing Language, TSL version 1.5 (Preliminary  
version),  
Stanford University, February 1, 1990, pp. 1-68.
- [Luckham\_3 et al.] Luckham, D.C., Vera, J., Bryan, D., Augustin, L., and Belz, F.,  
Partial Ordering of Event sets and Their Application to Prototyping Concurrent,  
Timed Systems  
Journal of Systems and Software, vol 21, 1993, pp.253-265.
- [Luckham\_4 et al.] Luckham, D.C., Kenney, J.J., Augustin, L.M., Vera, J., Bryan, D., Mann, W.,  
Specification and Analysis of System Architecture Using Rapid,  
IEEE Transactions on Software Engineering, vol.21, No.4, April 1995, pp.336-  
355.
- [Moorby] Moorby, P  
Verilog
- [Murphy] Murphy, Kevin, P.  
Passively learning finite automata.  
Technical Report 96-04-017, Santa Fe Institute, 1996.
- [Musuvathi et al] Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, and  
David L. Dill.  
Cmc: A pragmatic approach to model checking real code.  
In *Proc. 5th USENIX Symp. on Operating Systems Design and Implementation*,  
Boston, MA, 2002.

- [Necula & Lee] Necula, George C. and Lee, Peter  
"[Research on Proof-Carrying Code for Untrusted-Code Security](#)"  
In Proceedings of the 1997 IEEE Symposium on Security and Privacy, Oakland, 1997.
- [Nimmer & Ernst] Nimmer, J.W. and Ernst, M.D.  
Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java.  
In *Proceedings of RV'01, First Workshop on Runtime Verification*, Paris, France, July 2001.
- [Oppenheimer & Patterson] Oppenheimer, D. and Patterson, D.A.  
Architecture, operation, and dependability of large-scale Internet services: three case studies.  
*IEEE Internet Computing* special issue on Global Deployment of Data Centers, September/October 2002.
- [Oppenheimer et al.] Oppenheimer, D., Ganapathi A. & Patterson, D.  
*Why do Internet services fail, and what can be done about it?*  
4<sup>th</sup> USENIX Symposium on Internet Technologies & Systems, Seattle, Washington, March 2003.
- [Purify] <http://www-306.ibm.com/software/awdtools/purifyplus/>
- [Raman & Patrick] Raman, A.V. and Patrick, J.D.  
The sk-strings method for inferring PFSA.  
In *Proceedings of the workshop on automata induction, grammatical inference and language acquisition at the 14th international conference on machine learning (ICML97)*, 1997.
- [ROC] <http://roc.cs.berkeley.edu>
- [Rosenblum] Rosenblum, D.  
A Practical Approach to Programming with Assertions,  
*IEEE Transactions on Software Engineering*, vol.21, #1, 1994, pp.1253-1260
- [scsh.net] <http://www.scsh.net/cgi-bin/wiki.cgi?ReCover>
- [Stoustrup] Stoustrup, B  
C++ Programming Language
- [Synopsys] Vera  
[www.synopsys.com](http://www.synopsys.com)
- [Verisity] Specman  
[www.verisity.com](http://www.verisity.com)