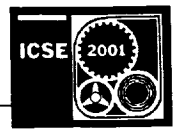
546927

Submitted to the 23rd International Conference on Software Engineering (ICSE 2001), Toronto, Canada
http://www.csr.uvic.ca/icse2001
WP ref: 01/icse/fastre (August 24, 2000).



Fast Formal Analysis of Requirements via "Topoi Diagrams"

Tim Menzies
Dept. Electrical & Computer Eng.
University of British Columbia,
2356 Main Mall, Vancouver, B.C.
Canada, V6T 1Z4
tim@menzies.com

John Powell
Averstar Inc.
NASA IV&V Facility
100 University Drive,
Fairmont WV, USA, 26554
john.powell@ivv.nasa.gov

Michael E. Houle
Basser Dept. of Computer Science
Madsen Building F09
The University of Sydney
Sydney, NSW, Australia, 2006
meh@cs.usyd.edu.au

ABSTRACT
Early testing of requirements can decrease the cost of removing errors in software projects. However, unless done carefully, that testing process can significantly add to the cost of requirements analysis. We show here that requirements expressed as *topoi diagrams* can be built and tested cheaply – using our SP2 algorithm, the formal temporal properties of a large class of topoi can be proven very quickly, in time nearly linear in the number of nodes and edges in the diagram. There are two limitations to our approach. Firstly, topoi diagrams cannot express certain complex concepts such as iteration and sub-routine calls. Hence, our approach is more useful for requirements engineering than for traditional model checking domains. Secondly, our approach is better for exploring the temporal *occurrence* of properties than the temporal *ordering* of properties. Within these restrictions, we can express a useful range of concepts currently seen in requirements engineering, and a wide range of interesting temporal properties.

Keywords
Formal methods, requirements engineering, model checking, SP2.

1 INTRODUCTION

The case for more formality in requirements engineering is overwhelming. Many errors in software can be traced back to errors in the requirements [32]. Often, the conception of a system is improved as a direct result of the discovery of inadequacies in the current conception. The earlier such inadequacies are found, the better, since the cost of removing errors at the requirements stage can be orders of magnitude cheaper than the cost of removing errors in the final system [33].

The benefit of formally checking a system is that formal proofs can find more errors than standard testing. A single formal first-order query is equivalent to many white-box or black-box test inputs [19].

The cost of rigorous requirements engineering may be impractically high. These costs include:

The modeling cost: Analysts must create a *systems model* and a *properties model*. Both models are in some machine-readable form. The properties model is often much smaller than the systems model and contains a formal temporal logic[1] description of the invariants that must be proved in the systems model.

The execution cost: A rigorous analysis of formal properties implies a full-scale search through the systems model. For example, if a given systems model has n variables each of which may take on a finite number of unique values m, then the size of the state space associated with that model is m^n. This space can be too large to explore, even on today's fast machines. Despite extensive research into speeding up this search (see our *Related Work* section), analysts often have to painstakingly rework the systems and properties models into more abstract and succinct forms that are small enough to permit formal analysis.

The personnel cost: Analysts skilled in formal methods must be recruited or trained. Such analysts are generally hard to find and retain.

The development brake: The above costs can be so high that the requirements must be frozen for some time while we perform the formal analysis. Hence, one of the costs of formal analysis is that it can slow the requirements process. Slowing down the requirements process is unacceptable for fast moving software companies, such as the start-up dot.coms.

Ideally, a method for reducing the cost of testing requirements would eliminate the execution cost and reduce the cost and skill involved in building the properties and systems models. If achievable, such a method would also reduce the personnel cost, since it would not require such highly-skilled analysts. Having reduced the personnel, modeling, and execution costs, this hypothetical method would inevitably decrease the development brake.

Some progress has already been made in reducing the cost of properties modeling using *temporal logic patterns*. Dwyer et.al. [9,10] have identified patterns within the temporal logic

[1] Temporal logic is classical logic augmented with some temporal operators such as $\Box X$ (always X is true), $\Diamond X$ (eventually X is true), $\bigcirc X$ (X is true at the next time point), $X \bigcup Y$: (X is true until Y is true),

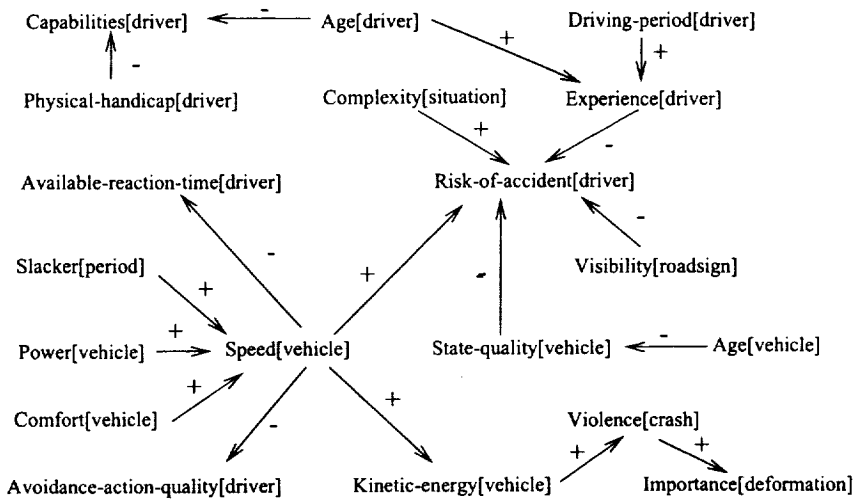


Figure 1: An example topoi from [7]. The formal semantics for topoi is described below. Informally, we say that + approximates "encourages" while − approximates "discourages".

formulae seen in many real-world properties models. For each pattern, they have defined an expansion from the intuitive pseudo-English form of the pattern to a formal temporal logic formulae. In this way, analysts are shielded from the complexity of formal logics. For example, the simple pseudo-English statement

$$always(brake = on)\ between(danger = seen)\ and(car = stop)$$

can be automatically expanded into the more arcane formal statement:

$$\Box((danger = seen \wedge !(car = stop)\ \wedge \Diamond\ (car = stop)) \rightarrow (brake = on\ \cup\ (car = stop)))$$

One drawback with temporal logic patterns is that while complex temporal formula can be automatically generated from intuitive pseudo-English, the execution cost remains. That is, even though we can quickly build the properties model, we may not be able to execute all of that properties model.

In this article, we argue that we can greatly reduce the execution cost for a class of systems models seen in the requirements stage, and for a large class of temporal logic properties. In our approach, we use temporal logic patterns to reduce the cost of properties modeling, and optimization to reduce the execution cost. The key to this reduction is SP2, a new algorithm for testing temporal properties of *topoi*, which are statements of gradual influences between variables. Topoi can represented graphically by *topoi diagrams*, an example of which is shown in Figure 1. Topoi are quick to sketch, and so (for requirements that are topoi-compatible) our approach also reduces the systems modeling cost.

These cost-reduction benefits can only be realized if we accept certain restrictions:

- Our approach limits the kinds of properties that can be tested.
- The systems model must be expressed as topoi diagrams. Topoi are not very expressive and excludes statement such as first-order assertions, iteration, subroutine calls, and assignment.
- Due to these language limitations, our approach is not suitable to domains that need the excluded statements; e.g. complex protocols seen in concurrent processes.

These restrictions are not fatal to the modeling process, at least at the requirements stage:

- We will describe how to quickly recognize inadmissible properties statements. Further, we will use the Dwyer et.al. survey to show that within the limits to the properties language, we can represent a wide range of useful temporal logic properties.
- We will show that topoi diagrams are sufficient to represent diagrams seen in certain approaches to requirements engineering and recording design rationales. Hence, when we say that this approach is practical and useful, we really mean *practical and useful for early life cycle requirements discussions only*.

This worked is based on Feldman & Compton's study of the validation of topoi [11, 12] (which they called qualitative compartmental models). Menzies tried to optimize that validation process and offered an implementation that was orders of magnitude faster than the validation engine built by Feldman & Compton. However, he could not reduce the exponential upper-bound on the runtimes [21–23]. Assuming a certain restriction on topoi edge types, Cohen, Menzies, Waugh and Goss showed that the cost of checking temporal properties of topoi-based simulation is a function of the

number of time-ticks in the query [24, 25]. This paper improves significantly on the Menzies et.al. result. We assume the same restriction as Menzies et.al. and introduce SP2, a nearly linear-time algorithm for checking a large class of interesting temporal properties (for space reasons, we describe the full details of that algorithm elsewhere [27]). Also, we describe an implementation of SP2 which, in at least one domain, out-performs a state-of-the-art temporal logic model checker (SPIN [15]).

2 About Topoi

Our approach assumes that requirements systems models are expressed in the form of *topoi*; i.e. statements of gradual statements such as (i) the more X, the more Y; (ii) the less X the less Y; (iii) the more X, the less Y; or (iv) the less X the less Y. Dieng et.al. name such statements "topoi" and give numerous examples from their records of interviews with experts [7]. For example:

> *The more there is water infiltration in the roadway body, the worse the foundation risks to be.*
>
> *The higher the speed of the vehicles, the more important the measure of importance relative to the roadway comfort.*
>
> *When the geometry increases, the mass increases and the frequency decreases.*
>
> *If there is a punctual undressing and if the roadway is between five and fifteen years old, then the causes "too old coating" is all the more certain since the roadway is older.*

Our experience has always been that the systems modeling cost with topoi is very low. Topoi graphs can be quickly generated in the requirements stage. Two feuding stakeholders with two marker pens and one whiteboard can generate many, many topoi in just a few hours.

Topoi graphs can be found in many domains. Figure 1 showed a topoi from an insurance domain using the graphical notation of Dieng's 3DKAT tool. Figure 2 show some Mylopoulos-style *soft-goal* graphs [28,29]. Soft-goal graphs represent gradual knowledge about non-functional requirements. In Figure 2, an expert describes how to increase business flexibility. Figure 3 shows a "questions-options-criteria" (QOC) graph from the design rationale community [34]. In such QOC graphs, questions suggest options and deciding on a certain option can raise other questions. Options shown in a box denote selected options. Options are assessed by criteria and criteria are gradual knowledge; i.e. they *tend to support* or *tend to reject* options. QOCs can succinctly summarize lengthy debates; e.g. 480 sentences uttered in a debate between two analysts on interface options can be displayed in a QOC graph on a single page [20]. Figure 5 shows topoi generated from the requirements of a rule-based legal system, shown in Figure 4. This translation

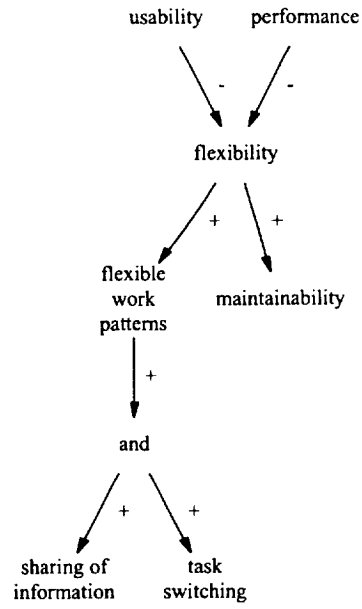


Figure 2: A soft-goal graph: the *and* node denotes that both *sharing of information* and *task switching* are enabled by *flexible work patterns*.

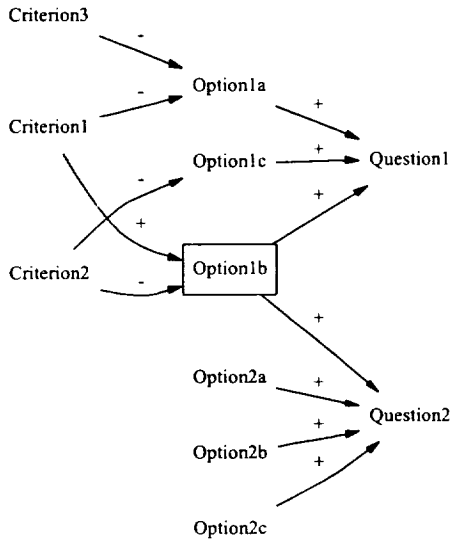


Figure 3: A questions-options-criteria graph from [34]

assumes that propositions in the rule base are modeled as a belief/strength pair where the strength is some continuous number.

When collected from multiple stakeholders, gradual statements can be quite complex, quite large, and contain feedback loops. Smythe extracted a list of gradual influences

```
if infant or moron                                then not legally_responsible.
if guilty                                         then jail.
if age < 7                                        then infant.
if legally_responsible and guilty                 then jail.
if motive and means and opportunity and witnesses then guilty.
if guilty and not legally_responsible             then not jail.
```

Figure 4: Rule-based requirements from a legal system.

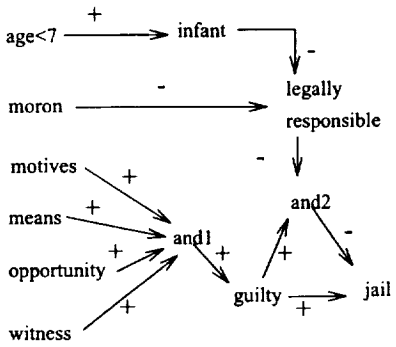


Figure 5: Topoi from Figure 4.

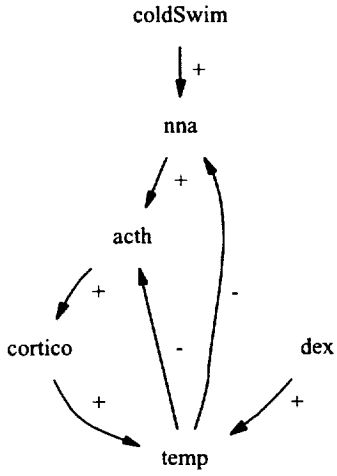


Figure 6: The Smythe '87 theory. From [35]. The diagram shows statements of gradual knowledge relating to laboratory experiments on mammals.

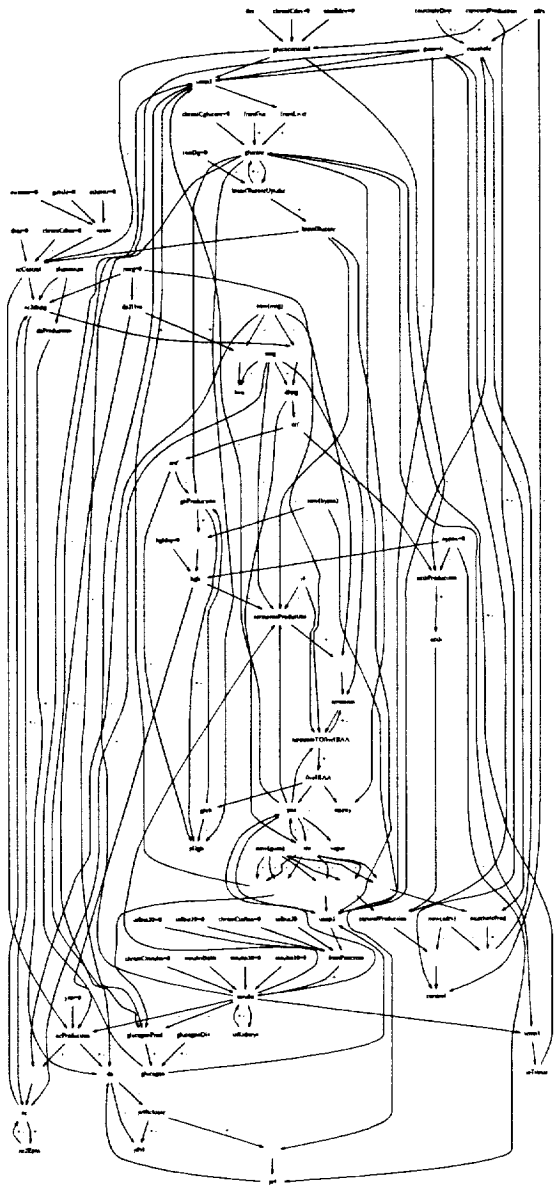


Figure 7: A large topoi with many loops.

from a set of articles from different authors relating to human internal physiology. The resulting network contains loops; see Figure 6. The experiments described later in the paper are based on the large topoi of Figure 7.

A pre-experimental concern is that informal topoi are so under-defined that we could use them to infer any properties at all. This turns out not to always be the case. Recall Figure 2 and the fragment:

$$usability \bar{\rightarrow} flexibility \bar{\leftarrow} performance$$

Note that there is no way to explain the output of $\{flexibility \uparrow\}$ from the input of $\{usability \uparrow, performance \uparrow\}$. That is, while topoi are over-generalized, they may still be restrictive enough to demonstrate what cannot be proved. We describe below experiments which show that large real-world topoi can be restrictive enough to block an interesting number of temporal properties.

Topoi: Formal Semantics

Formally, we say that a topoi is a directed, possibly cyclic graph G containing vertices and edges $< V, E >$. E are the connectors between variables and are one of a set of pre-

defined types; e.g. $\stackrel{+}{\rightarrow}$ or $\stackrel{-}{\rightarrow}$. That is:

$$\begin{aligned} E_i &= V_i \stackrel{+}{\rightarrow} V_j \text{ or } V_i \stackrel{-}{\rightarrow} V_j \\ G &= < V, E > \end{aligned}$$

The vertices of a topoi can be assigned a finite number of values; e.g. *up*, *down* or *steady*. These values model the sign of the first derivative of these variables (i.e. the rate of change in each value). $X \stackrel{+}{\rightarrow} Y$ denotes that Y being *up* or *down* could be explained by X being *up* or *down* respectively. That is:

$$V_i \stackrel{+}{\rightarrow} V_j \equiv \begin{cases} V_i \uparrow & implies \quad V_j \uparrow \\ V_i \downarrow & implies \quad V_j \downarrow \end{cases} \tag{1}$$

(where $\uparrow$ and $\downarrow$ denote up and down respectively.)

$X \stackrel{-}{\rightarrow} Y$ denotes that Y being *up* or *down* could be explained by X being *down* or *up* respectively. That is:

$$V_i \stackrel{-}{\rightarrow} V_j \equiv \begin{cases} V_i \uparrow & implies \quad V_j \downarrow \\ V_i \downarrow & implies \quad V_j \uparrow \end{cases} \tag{2}$$

Tacit in our topoi diagrams are conjunctions of influences. We can view topoi as influences splashing around pipes that connect tubs. Pairs of competing influences can cancel out. That is, we can explain the level of water in a tub remaining steady via conjunction of competing upstream influences; e.g.

$$\begin{pmatrix} (V_i \uparrow \ implies \ V_j \uparrow) \\ and \\ (V_k \downarrow \ implies \ V_j \downarrow) \end{pmatrix} implies \begin{pmatrix} (V_i \uparrow \wedge V_k \downarrow) \\ implies \\ (V_j = steady) \end{pmatrix} \tag{3}$$

This formal semantics is sufficient to guide the translation of topoi for a formal model checker such as SPIN. Figure 8 shows the results of such a translation of Figure 6. In this figure, all the nodes have the values *up*, *down*, *steady* and *unknown* (which is a placeholder for the initial conditions). Also, for convenience, all systems model inputs X are declared to be X_{chg} variables with values $arrived, left$ denoting the differences between these variables in different experiments. For example, if we increase the injections of *dex*, then we also say that $dex_{chg} = arrived$.

Proving Formal Properties in Topoi

We can test topoi using libraries of expected or desired behavior. Such libraries can be quickly built via interviews with users. We have found it useful to structure these interviews in an OO framework. After generating use cases and particular scenarios [18], we ask our users to clarify exactly what are the expected inputs and required outputs for each scenario. This generates two artifacts. Firstly, it leads to topoi graph describing how they think influences should propagate around a systems model. Secondly, it leads to the formulation of properties models of the form:

When I do this, I expect to see that.

```
#define DOWN  0
#define STEADY  1
#define UP 2
#define UNDEF 3
#define ARRIVED 0
#define LEFT  1

byte chg_cold_swim = UNDEF /* chg_cold_swim = {ARRIVED,LEFT} */
byte chg_dex = UNDEF       /* chg_dex = {ARRIVED,SWIM}      */
byte cold_swim = UNDEF     /* cold_swim = {DOWN,STEADY,UP}  */
byte dex = UNDEF           /* dex = {DOWN,STEADY,UP}        */
byte temp = UNDEF          /* temp = {DOWN,STEADY,UP}       */
byte nna = UNDEF           /* nna = {DOWN,STEADY,UP}        */
byte acth = UNDEF          /* acth = {DOWN,STEADY,UP}       */
byte cortico = UNDEF       /* cortico = {DOWN,STEADY,UP}    */

active proctype smythe() {
      if
        ::dex == UNDEF  -> dex = DOWN
        ::dex == UNDEF -> dex = STEADY
        ::dex == UNDEF -> dex = UP
      fi;
      if
        ::cold_swim == UNDEF -> cold_swim = DOWN
        ::cold_swim == UNDEF -> cold_swim = STEADY
        ::cold_swim == UNDEF -> cold_swim = UP
      fi;
      if
        ::chg_dex == UNDEF -> chg_dex = ARRIVED
        ::chg_dex == UNDEF -> chg_dex = LEFT
      fi;
      if
        ::chg_cold_swim == UNDEF -> chg_cold_swim = ARRIVED
        ::chg_cold_swim == UNDEF -> chg_cold_swim = LEFT
      fi;
      if
        ::chg_dex == ARRIVED -> temp = UP
        ::chg_dex == LEFT -> temp = DOWN
      fi;
      if
        ::chg_cold_swim == ARRIVED -> nna = UP
        ::chg_cold_swim == LEFT -> nna = DOWN
      fi;
      do
        ::(chg_cold_swim == ARRIVED && temp == UP) -> nna = STEADY
        ::(chg_cold_swim == LEFT && temp == DOWN) -> nna = STEADY
        ::temp == DOWN -> nna = UP
        ::temp == UP -> nna = DOWN
        ::temp == DOWN -> acth = UP
        ::temp == UP -> acth = DOWN
        ::nna == UP -> acth = UP
        ::nna == DOWN -> acth = DOWN
        ::acth == UP -> cortico = UP
        ::acth == DOWN -> cortico = DOWN
        ::cortico == UP -> temp = UP
        ::(cortico == UP && chg_dex == LEFT) -> temp = STEADY
        ::cortico == DOWN -> temp = DOWN
        ::(cortico == DOWN && chg_dex == ARRIVED) -> temp = STEADY
        ::(temp == UP && nna == UP) -> acth = STEADY
        ::(temp == DOWN && nna == DOWN) -> acth = STEADY
      od;
}
```

Figure 8: Figure 6 expressed in the PROMELA language used in SPIN model checker [15].

or, in the language of temporal logic used in (e.g.) SPIN:

$$\Box (Inputs \rightarrow \Diamond \ Outputs) \tag{4}$$

i.e. always the *inputs* lead, eventually, to the *outputs*.

We encounter problems if we use Equation 4 to check large topoi using standard model checkers. While SPIN checks Equation 4 against Figure 8 in less than a second, it can fail to terminate for larger systems models. In one study, we offered 40 properties of the form of Equation 4 to SPIN along with Figure 7 expressed in the same format as Figure 8. Given 100MB of maximum RAM, SPIN ran out of memory for most of the properties. We suspected that the search space was too big. Figure 7 contains 80 variables, each of which has at least the values $up, down, steady, undef$; i.e. total space of options at least of size ($4^{80} \approx 10^{48}$). In a second study, we reduced the size of the system by removing the *steady* values. This shrank the options to ($3^{80} \approx 10^{38}$). However, even in this reduced system, SPIN ran out of memory and failed to prove anything for 29 of the 40 proper-

ties [31].

In summary, while theoretically we can assess topoi using standard model checkers, in practice, this may not be feasible.

3 SP2: A Model Checker for Topoi

While general topoi defeat general-purpose model checkers, specialized model checkers can quickly check the temporal properties of a restricted class of topoi. Consider a topoi containing two-valued nodes connected by the "+" and "−" edges defined in Equation 1 and Equation 2. Such a topoi has *symmetric edges*; i.e. each edge comments on a connection of every upstream node's value to every downstream node's value. Menzies et.al. showed that when every edge of a symmetric topoi comments on all the values of its downstream vertices, then the state space rapidly *saturates* [24, 25]. That is, the granularity of the time axis reduces to the number of variables in that theory. For example, in a systems model where every variable has only two values, everything that is reachable can be reached in two time ticks.

Using the result of Cohen et.al we have defined SP2, a specialized model checker for symmetric topoi [27, 31]. SP2 is a variant of Dijkstra's shortest path algorithm [6, 8]. The algorithm inputs a symmetric topoi with edge set E, node set V, and an initial set $S \subset V$. S contains some value assignments to some nodes and represents the initial conditions of the system. The algorithm outputs a set of edges Z with the following properties:

- Z is a collection of trees spanning all the nodes reachable from the inputs.
- For any reachable node z, Z contains the shortest topoi path from the inputs to z.
- The nodes of V spanned by Z are partitioned into two sets S' and T', where:
  - No edge of Z passes from T' to S'.
  - Each set is consistent; that is, will not contain both $x\uparrow$ and $x\downarrow$.

Elsewhere, we have proved that SP2 generates S' and T' correctly, and runs in $O(|V| + |E| \log |V|)$ time in the worst case [27]. SP2 is efficient due to its exploitation of saturation. While spreading out over the topoi, it maintains two sets of nodes: the *now* set (S') and the *later* set (T'). If the algorithm reaches a node that contradicts something else in the *now*, it moves the new node into the *later* set. The repeated application of this rule on a 2-spaced symmetric topoi results in a fast division of the nodes reachable from the initial conditions into the two sets S' and T'.

Using SP2, we can very quickly explore temporal properties that can be proved in two time ticks. A large range of interesting queries can be executed in two time ticks (but see below for a discussion on the properties that require more than two time ticks). Once S' and T' are generated, we can convert our temporal properties into set membership tests of these sets. Figure 9 and Figure 10 show conversion rules for common temporal properties.

exp	→ □ exp	//always
	| ◇ exp	//eventualy
	| ○ exp	//next
	| exp *W* exp	// weak until
	| exp ⋃ exp	//until
	| exp ⋀ exp	//conjunction
	| exp ⋁ exp	//disjunction
	| exp → exp	//implication
	| ! exp	//negation
	| x	//proposition
exp *W* exp	→ (exp ⋃ exp) ⋀ □ exp	
□ exp	→ (x ∈ now') ⋀ (x ∈ later')	
◇ exp	→ (x ∈ now') ⋁ (x ∈ later')	
○ exp	→ (x ∈ later')	
$exp_1 \bigcup exp_2$	→ (y ∈ now') ⋁ ((x ∈ now') ⋀ (y ∈ later'))	
$exp_1 \bigwedge exp_2$	→ (x ∈ now') ⋀ (y ∈ now')	
$exp_1 \bigvee exp_2$	→ (x ∈ now') ⋁ (y ∈ now')	
$exp_1 \rightarrow exp_2$	→ (y ∈ now') ⋁ ((x ∉ now') ⋀ (y ∉ now'))	
exp_1 *W* exp_2	→ ($exp_1 \bigcup exp_2$) ⋁ (□ exp_1)	
! exp	→ (x ∉ time)	
x	→ (x ∈ time)	
time	→ now' | later'	
now'	→ s'	
later'	→ t'	

Figure 9: Rewrite rules for converting linear temporal logic expressions into set membership tests of SP2's S', T'.

SP2 offers two major other advantages over standard temporal reasoning:

1. SP2 runs, terminates, returns Z, and then we perform set membership of Z to prove our properties. That is, we do not test for properties till *after* SP2 terminates. Hence, the inference time is not much affected by the complexity of the properties to be tested.

2. SP2 uses a shortest-paths tree to build its proofs. That is, when explaining how properties were reached, SP2 will generate the shortest explanation possible, Hence, a user of SP2 need not wade through mountains of trace files in order to understand how the properties were proved.

Experiments with SP2

Figure 11 shows a comparison of SPIN vs SP2 using properties of the form of Equation 4 and the systems model of Figure 7. Of the 40 properties which were analyzed by both SPIN and SP2, SPIN was able to return a verification result in only 11 out of 40 cases (27.5%) before running out of memory. In every case where SPIN did return a verification result, SP2's result was in agreement.

Regarding computer resources, SP2 used less than 1% of the RAM required by SPIN. Also, in the case of the unprovable properties, SP2 terminated in less than a second CPU time while SPIN took much longer.

Figure 10.A: Absence properties: p is false

Property	LTL	SP2
Globally	$\Box(!p)$	$p \notin S' \bigwedge p \notin T'$
Before R	$\Diamond R \rightarrow (!p \bigcup R)$	$R \in S' \bigvee R \notin T' \bigvee p \notin S'$
After Q	$\Box(Q \rightarrow \Box(!p))$	$(Q \in S' \bigvee (p \notin S' \bigwedge p \notin T')) \bigwedge (Q \in T' \bigvee p \notin T')$
Between Q and R	$\Box((Q \bigwedge !R \bigwedge \Diamond R) \rightarrow (!p \bigcup R))$	$(Q \notin T' \bigvee p \notin T') \bigwedge (Q \notin S' \bigvee R \in S' \bigvee R \notin T' \bigvee p \notin S')$
After Q until R	$\Box(Q \bigwedge !R \rightarrow (!p \mathcal{W} R))$	$(Q \notin T' \bigvee R \in T') \bigwedge (Q \notin S' \bigvee R \in S' \bigvee (p \notin S' \bigwedge R \in T'))$

Figure 10.B: Existence properties: p becomes true

Globally	$\Diamond(p)$	$p \in S' \bigvee p \in T'$
Before R	$!R \; \mathcal{W}(p \bigwedge !R)$	$(p \in S' \bigwedge (R \notin S' \bigvee p \in T')) \bigvee (R \notin S' \bigwedge p \in T' \bigwedge R \in T')$
After Q	$\Box(!Q \bigvee \Diamond(Q \bigwedge \Diamond p))$	$p \in T' \bigvee ((Q \notin S' \bigvee p \in S') \bigwedge Q \notin T')$
Between Q and R	$\Box(Q \bigwedge !R \rightarrow (!R \mathcal{W}(p \bigwedge !R)))$	$Q \in S' \bigwedge Q \in T' \bigwedge p \in T' \bigwedge (R \in S' \bigvee p \in S' \bigvee R \notin T')$
After Q until R	$\Box(Q \bigwedge !R \rightarrow (!R \bigcup (p \bigwedge !R)))$	$Q \in S' \bigwedge Q \in T' \bigwedge (R \in T' \bigvee p \in T') \bigwedge (R \in S' \bigvee p \in S' \bigvee (p \in T' \bigwedge R \notin T'))$

Figure 10.C: Universality: p always true

Globally	$\Box(p)$	$p \in S' \bigwedge p \in T'$
Before R	$\Diamond R \rightarrow (p \bigcup R)$	$R \in S' \bigvee (R \in T' \bigwedge p \in S')$
After Q	$\Box(Q \rightarrow \Box(p))$	$(Q \notin S' \bigvee (p \in S' \bigwedge p \in T')) \bigwedge Q \notin T' \bigvee p \in T')$
Between Q and R	$\Box((Q \bigwedge !R \bigwedge \Diamond R) \rightarrow (p \bigcup R))$	$Q \in S' \bigvee R \in S' \bigvee R \notin T' \bigvee p \in S'$
After Q until R	$\Box(Q \bigwedge !R \rightarrow (p \mathcal{W} R))$	$Q \in S' \bigwedge Q \in T' \bigwedge (R \in S' \bigvee p \in S') \bigwedge (R \in T' \bigvee p \in T')$

Figure 10.D: Precedence: S precedes p

Globally	$!p \mathcal{W} S$	$S \notin S' \bigvee (p \in S' \bigwedge (p \notin T' \bigvee S \in T'))$
Before R	$\Diamond R \rightarrow (!p \bigcup (S \bigvee R))$	$S \in S' \bigvee R \in S' \bigvee R \notin T' \bigvee p \notin S'$
After Q	$\Box !Q \bigvee \Diamond(Q \bigwedge (!p \mathcal{W} S))$	$(Q \notin S' \bigwedge Q \notin T') \bigvee (Q \in S' \bigwedge (S \in S' \bigvee (p \notin S' \bigwedge (p \notin T' \bigvee S \in T'))) \bigvee (Q \in T' \bigvee p \notin T' \bigvee S \in T'))$
Between Q and R	$\Box((Q \bigwedge !R \bigwedge \Diamond R) \rightarrow (!p \bigcup (S \bigvee R)))$	**Inexpressible**: needs > 2 time ticks
After Q until R	$\Box(Q \bigwedge !R \rightarrow (!p \mathcal{W}(S \bigvee R)))$	$(S \in S' \bigvee R \in S' \bigvee (p \notin S' \bigwedge p \notin T') \bigvee (p \in S' \bigwedge (S \in T' \bigvee R \in T'))) \bigwedge (S \in T' \bigvee R \in T' \bigvee p \notin T')$

Figure 10.E: Response: S responds to p

Globally	$\Box(p \rightarrow \Diamond S)$	$S \in T' \bigvee (p \notin T' \bigwedge (p \notin S' \bigvee S \in S'))$
Before R	$\Diamond R \rightarrow (p \rightarrow (!R \bigcup (S \bigwedge !R))) \bigcup R$	**Inexpressible**: needs > 2 time ticks
After Q	$\Box(Q \rightarrow \Box(p \rightarrow \Diamond S))$	$S \in T' \bigvee ((Q \notin S' \bigvee (p \notin T' \bigwedge (p \notin S' \bigvee S \in S'))) \bigwedge (Q \notin T' \bigvee p \notin T'))$
Between Q and R	$\Box((Q \bigwedge !R \bigwedge \Diamond R) \rightarrow (p \rightarrow (!R \bigcup (S \bigwedge !R))) \bigcup R)$	**Inexpressible**: needs > 2 time ticks
After Q until R	$\Box(Q \bigwedge !R \rightarrow ((p \rightarrow (!R \bigcup (S \bigwedge !R))) \mathcal{W} R))$	$Q \in S' \bigwedge Q \in T' \bigwedge (R \in T' \bigvee S \in T' \bigvee p \notin T')$

Figure 10: Common temporal logic queries converted into set membership tests of SP2's S', T'. This table was generated by applying the re-write rules of Figure 9 to a survey of common temporal logic queries [9, 10]. From [31].

	SPIN	SP2	Number of Cases
	??	proved	21
	??	unproved	8
	proved	proved	11
	unproved	unproved	0
	proved	unproved	0
	unproved	proved	0
RAM used (max)	100MB	< 1MB	

Figure 11: Proving properties of Figure 7 in SPIN and SP2. "??" denotes that SPIN did not terminate in 100MB of RAM.

We mentioned earlier that one pre-experimental concern with informal topoi is that they are so under-defined that we could use them to infer any set of properties at all. Figure 11 shows that this is not always true. In the case of 8 of the 40 properties, SP2 could not prove them across the large under-defined topoi of Figure 7.

Limits to SP2

What are the practical implications of SP2's restrictions? We discuss below two important implications: restrictions of the properties that can be proved and the need for special tools to handle conjunctions.

Inadmissible Properties

Figure 12 shows Dwyer et.al.'s classification of over five hundred linear temporal logic (LTL) properties [9,10]. Those properties divide into eight groups and each group contains the five temporal scopes seen in Figure 10; i.e. globally; before event R; after event Q; events Q and R; and after event Q until event R. $\frac{22}{40}$ of these scopes are expressible in terms of two time ticks [31]. The inexpressible scopes all require proving some ordering of > 2 events. By definition, such an ordering cannot be expressed using merely the two time intervals of S' and T' generated by SP2.

Figure 12 shows us that SP2-style inference on symmetric topoi can say more about the *occurrence* of a given event/state during system execution than about the *order*ing in which multiple events/states occur. It is a simple matter to detect the temporal properties that are inadmissible for SP2. All such properties require more than two time ticks; e.g.

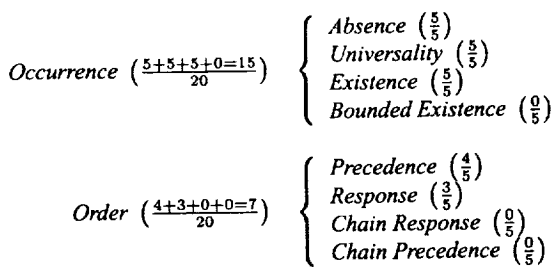


Figure 12: Coverage of the Dwyer corpus of temporal properties by SP2. Each right-hand-side group of properties contains five scopes. Fractions denote how many of those scopes can be handled by SP2, as seen in Figure 10. Adapted from [31].

until operators nested to a depth greater than two such as:.

$$\Big(day = sunday \bigcup \Big(day = monday \bigcup day = tuesday\Big)\Big)$$

Handling Conjunctions

Another problem is that symmetric topoi have no special knowledge of and-nodes. This can lead to some less-than-desirable results. Consider the following topoi:

$$usability \xrightarrow{-} flexibility \xleftarrow{-} performance$$

Equation 3 says that the conjunction of competing upstream influence can result in a steady value in a downstream variable; i.e.

$$\begin{aligned} usability\uparrow &\rightarrow and001 \\ performance\downarrow &\rightarrow and001 \\ and001 &\rightarrow flexibility = steady \end{aligned} \quad (5)$$

where $and001$ is an and-node especially created for this conjunction. A reasonable temporal interpretation of and-nodes is that all pre-conditions must appear before or at the same time as the post-conditions. Suppose we seek to $flexibility = steady \in S'$, but SP2 computes a node partition in which $usability\uparrow \in T'$ and $performance\downarrow \in T'$. We would like to be able to coax these pre-conditions back in time to S' such that they do not occur at a time that is later than $flexibility = steady \in S'$.

Another case where we want to coax edge weights is the *bad-and* situation. The rules of symmetric topoi require that if we create the edges shown in Equation 5, then we must also create the following complementary rules:

$$\begin{aligned} usability\downarrow &\rightarrow \overline{and001} \\ performance\uparrow &\rightarrow \overline{and001} \\ \overline{and001} &\rightarrow \overline{flexibility = steady} \end{aligned} \quad (6)$$

where $\overline{X}$ is an invented node representing "all the other values of X". The addition of the nodes $\overline{and001}$ and $\overline{flexibility = steady}$ is required to ensure the symmetry properties upon which SP2 is dependent. However, they are just nonsense symbols that should never appear in any explanation of how certain inputs lead to certain properties. That is, pathways from inputs to properties should never include these nonsense symbols. Hence, if possible, SP2 should be 'coaxed' into producing shortest path trees in which these spurious nodes appear at the leaves.

SP2 contains a mechanism to implement such coaxing: each edge in the topoi is augmented with an edge weight, which SP2 uses to compute shortest paths – the length of a path is simply the sum of the weights of the edges along the path. At the core of SP2 is a priority queue. At runtime, the next edge to be explored is one of the edges with lowest weight within the queue. This means that by adjusting weights and re-running the algorithm, we can choose to explore edges at some earlier time or later time. Hence, to coax $usability\uparrow$ and $performance\downarrow$ into S', we can adjust the weights upstream of those nodes. In coaxing, the weights can be adjusted arbitrarily, provided that the any symmetric pair of edges receives the same weight for both edges. Elsewhere [27] we define a set of minimal edge adjustment heuristics which input SP2's shortest path tree Z, the cut set C containing the edges that connect S' to T' and which outputs changes to the edge weights.

A major pre-experimental concern was that the nearly linear-time processing of SP2 could be followed by an indefinitely long coaxing process. After much experimentation, we can report that we have never seen this worst-case behavior in practice. In those experiments we used SP2 to explore randomly generated properties of the form of Equation 4 over dozens of randomly generated topoi graphs. We varied topoi fanout (2 to 6 edges per node) and the frequency of and-nodes (from 5% to 75%). Each experiment was terminated when the % of provable properties reached some plateau. In all the experiments, the plateau was reached after < 5 iterations of SP2+coaxing. Also, the plateau reached after 10 coaxes barely changed in up to 100 coaxes. Further, SP2 never used more than 1MB of memory or one minute of runtime. Our conclusion from these experiments is that the need for heuristic coaxing does not diminish the time and space efficiency of SP2.

4 Related Work

We are hardly the first to explore formal methods for requirements engineering. For example:

- In the KAOS system [36], analysts generate a properties model by incrementally augmenting object-oriented scenario diagrams with temporal logic statements. Potentially, this research reduces the costs of formal requirements analysis by integrating the generation of the properties model into the rest of the system development. Our reading of the KAOS work is that while the resulting model may be more formal, the level of skill

required to write the temporal logic can significantly increase the personnel cost. Further, the extra time required for the augmentation could increase the effect of the development brake.

- Schneider et.al. [33] explored reducing the manual modeling costs using *lightweight formal methods*. In the lightweight approach, only partial descriptions of the systems and properties models were constructed using the SPIN formal analysis tool [15]. Despite their incomplete nature, Schneider et.al. found that such partial models could still detect significant systems errors. While exciting research, this approach still incurs the personnel cost since scarce expertise is required to drive tools like SPIN.

Nor are we the first to explore optimizing temporal logic model checking. Elaborate tools have been developed to tame the state space explosion problem including:

Abstraction or partial ordering: Only use the part of the space required for a particular proof. Implementations exploiting this technique can restrain how the space is traversed [14, 26], or constructed in the first place [13, 33].

Clustering: Divide the systems model into sub-systems which can be reasoned about separately [2, 4, 30, 37].

Meta-knowledge: Avoid studying the entire space. Instead, only study succinct meta-knowledge of the space. One example used an eigenvector analysis of the long-term properties of the systems model under study [17].

Exploiting symmetry: Prove properties in some part of the systems model, then reuse those proofs if ever those parts are found elsewhere in the systems model [3].

Semantic minimization: Replace the space with some smaller, equivalent space [16] or ordered binary decision diagrams [1]. For example, the BANDERA system [5] reduces both the systems modeling cost and the execution cost via automatically extracting (slicing) the minimum portions of a JAVA program's bytecodes which are relevant to particular properties models.

While the above tools have all proved useful in their test domains, they may not be universally applicable.

- Certain optimizations require expensive pre-processing, such as [17].
- These methods may rely on certain topological features of the system being studied. Exploiting symmetry is only useful if the system under study is highly symmetric. Clustering generally fails for tightly connected models.

Also, for requirements engineering, systems like BANDERA are not suitable. BANDERA only works on implemented systems; that is, not until long after the requirements phase has ended.

Hence, in the general case, only small models can be tested. Further, these models must be precisely specified. In contrast, this work describes methods for quickly proving properties in large models that have been hastily sketched.

5 Conclusion

We need better formal testing for our requirements. Applying formal methods can lead to an unacceptable brake on the development process. Cost-effective formal methods have to reduce the cost and skill involved in modeling systems and their properties. The cost of properties modeling can be reduced via temporal logic patterns. However, the execution cost of the resulting properties model may require expensive rework of the properties generated from the patterns.

In the specific case of requirements that can be mapped into symmetric topoi, we have shown that the systems modeling cost is reduced (since the topoi can be sketched quickly). For such symmetric topoi, we can reduce the execution cost for proving formal properties to time that is nearly linear on the number of edges and nodes in the topoi.

The combination of easy specification of properties and systems models implies that the personnel cost of formal modeling is reduced. This cost-reduction can only be achieved in domains were the systems model can be expressed as topoi and the properties model refers more to temporal occurrence properties than temporal ordering properties. We have argued that requirements engineering is one such domain.

Having built the SP2 engine, our next goal is the construction of a shell that exploits this engine. Our current research goal is the construction of the RAPTURE shell. RAPTURE exploits SP2 to enable the fast formal analysis of topoi-compliant descriptions of software systems.

ACKNOWLEDGEMENTS

This work was partially supported NASA through cooperative agreement #NCC 2-979.

REFERENCES

[1] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3), September 1992.

[2] D. Clancy and B. Kuipers. Model decomposition and simulation: A component based qualitative simulation algorithm. In *AAAI-97*, 1997.

[3] E. Clark and T. Filkorn. Exploiting symmetry in temporal logic model checking. In *Fifth International Conference on Computer Aided Verification*. Springer-Verlag, 1993.

[4] P. Clark and T. Ng. The cn2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[5] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasarenu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proceedings ICSE2000, Limerick, Ireland*, pages 439–448, 2000.

[6] T. Cormen, C. E. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990. ISBN: 0262031418.

[7] R. Dieng, O. Corby, and S. Lapalut. Acquisition and exploitation of gradual knowledge. *International Journal of Human-Computer Studies*, 42:465–499, 1995.

[8] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[9] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *ICSE98: Proceedings of the 21st International Conference on Software Engineering*, May 1998.

[10] M. B. Dwyer, G. S. Avrunin, and J. Corbett. A system specification of patterns. `http://www.cis.ksu.edu/santos/spec-patterns/`, 1997.

[11] B. Feldman, P. Compton, and G. Smythe. Hypothesis Testing: an Appropriate Task for Knowledge-Based Systems. In *4th AAAI-Sponsored Knowledge Acquisition for Knowledge-based Systems Workshop Banff, Canada*, 1989.

[12] B. Feldman, P. Compton, and G. Smythe. Towards Hypothesis Testing: JUSTIN, Prototype System Using Justification in Context. In *Proceedings of the Joint Australian Conference on Artificial Intelligence, AI '89*, pages 319–331, 1989.

[13] M. Fujita. Model checking: Its basics and reality. In *Asia and South Pacific - Design Automation Conference*, 1998.

[14] P. Godefroid. On the costs and benefits of using partial-order methods for the verificiation of concurrent systems (invited papers). In *The 1996 DIMACS workshop on Partial Order Methods in Verificaition, July 24-26, 1996*, pages 289–303, 1997.

[15] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[16] G. Holzmann and A. Puri. A minimized automaton representation of reachable states, 1999.

[17] Y. Ishida. Using global properties for qualitative reasoning: A qualitative system theory. In *Proceedings of IJCAI '89*, pages 1174–1179., 1989.

[18] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[19] M. Lowrey, M. Boyd, and D. Kulkarni. Towards a theory for integration of mathematical verification and empirical testing. In *Proceedings, ASE'98: Automated Software Engineering*, pages 322–331, 1998.

[20] A. MacLean, R. Young, V. Bellotti, and T. Moran. Questions, options and criteria: Elements of design space analysis. In T. Moran and J. Carroll, editors, *Design Rationale: Concepts, Techniques, and Use*, pages 53–106. Lawerence Erlbaum Associates, 1996.

[21] T. Menzies. *Principles for Generalised Testing of Knowledge Bases*. PhD thesis, University of New South Wales, 1995. Available from `http://tim.menzies.com/pdf/95thesis.pdf`.

[22] T. Menzies. Applications of abduction: Knowledge level modeling. *International Journal of Human Computer Studies*, 45:305–355, 1996. Available from `http://tm.menzies.com/pdf/96abkl.pdf`.

[23] T. Menzies. On the practicality of abductive validation. In *ECAI '96*, 1996. Available from `http:/tim.menzies.com/pdf/96ok.pdf`.

[24] T. Menzies and R. Cohen. A graph-theoretic optimisation of temporal abductive validation. In *European Symposium on the Validation and Verification of Knowledge Based Systems, Leuven, Belgium*, 1997. Available from `http://tim.menzies.com/pdf/97eurvav.pdf`.

[25] T. Menzies, R. Cohen, S. Waugh, and S. Goss. Applications of abduction: Testing very long qualitative simulations. *IEEE Transactions of Data and Knowledge Engineering (to appear). Available from* `http://tim.menzies.com/pdf/97iedge.pdf`, 2001.

[26] T. Menzies and P. Compton. Applications of abduction: Hypothesis testing of neuroendocrinological qualitative compartmental models. *Artificial Intelligence in Medicine*, 10:145–175, 1997. Available from `http://tim.menzies.com/pdf/96aim.pdf`.

[27] T. Menzies, M. Houle, and J. Powell. Rapture/sp2: Efficient testing of temporal properties without search space explosion, 1999. NASA IV&V Facility Technical Report, `http://research.ivv.nasa.gov/docs/techreports.html`.

[28] J. Mylopoulos, L. Cheng, and E. Yu. From object-oriented to goal-oriented requirements analysis. *Communications of the ACM*, 42(1):31–37, January 1999.

[29] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions of Software Engineering*, 18(6):483–497, June 1992.

[30] K. Olender and L. Osterweil. Interprocedural static analysis of sequencing constraints. *TOSEM*, 1(2):21–52, 1992.

[31] J. Powell. The rapture/sp2 approach to model checking: An explanation and viability experimentation, 1999.

[32] D. Reifer. Software failure modes and effects analysis. *IEEE Transactions on Reliability*, pages 247–249, 1979.

[33] F. Schneider, S. Easterbrook, J. Callahan, G. Holzmann, W. Reinholtz, A. Ko, and M. Shahabuddin. Validating requirements for fault tolerant systems using model checking. In *3rd IEEE International Conference On Requirements Engineering*, 1998.

[34] S. B. Shum and N. Hammond. Argumentation-based design rationale: What use at what cost? *International Journal of Human-Computer Studies*, 40(4):603–652, 1994.

[35] G. Smythe. Hypothalamic noradrenergic activation of stress-induced adrenocorticotropin (ACTH) release: Effects of acute and chronic dexamethasone pre-treatment in the rat. *Exp. Clin. Endocrinol. (Life Sci. Adv.)*, pages 141–144, 6 1987.

[36] A. van Lamsweerde and L. Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE Transactions on Software Engineering, Special Issue on Scenario Management*, November 1998.

[37] Z. Zhang. An approach to hierarchy model checking via evaluating ctl hierarchically. In *Fourth Asian Test Symposium*, 1995.