Title	Implementing dynamic flowgraph methodology models with logic programs
Author(s)	Karanta, Ilkka
Citation	Journal of Risk and Reliability, Vol. 227 (2013) No: 3, Pages: 302 - 314
Date	2013
URL	http://dx.doi.org/10.1177/1748006X13484425
Rights	SAGE Publications. This post-print version of the article can be downloaded for personal use only.

Implementing dynamic flowgraph methodology models with logic programs

Ilkka Karanta[1]
VTT, Espoo, Finland

Abstract: The dynamic flowgraph methodology (DFM) is a promising way to find the prime implicants of a top event for a dynamic system possibly containing digital subsystems. This paper demonstrates how to express dynamic flowgraph methodology models as logic programs, and top events as queries to those programs, in a natural and comprehensible way. Computation of the logic program lists the prime implicants of a top event in the system. We also present and implement an algorithm for computing the probability of the top event from its prime implicants. Together, computation of prime implicants and calculation of top event probability from these constitute a complete way of finding a system's failure probability. Logic programs, implemented in this paper in the leading logic programming language Prolog, enable rapid prototyping of DFM models. The logic programming framework introduced here could also be utilized in teaching DFM in risk analysis courses.

Keywords: dynamic flowgraph methodology, logic programming, Prolog, dynamic PRA

1. Introduction

The reliability analysis of dynamic systems is becoming more and more important. Still more important is the reliability analysis of systems containing digital (computerized) subsystems; indeed, lack of sufficient evidence of the reliability of e.g. digital automation subsystems in nuclear power plants may in some cases have delayed their eventual commissioning. Appropriate methods for the reliability analysis of dynamic systems possibly containing digital subsystems are therefore of great importance to the industry, and also a field of active study within the reliability research community.

The dynamic flowgraph methodology (DFM) [1] is a reliability modeling framework that can handle both dynamic systems and systems containing digital subsystems. It has been claimed [2], with some justification, that DFM and Markov models are the two main contenders for the reliability analysis of such systems. The main advantage of DFM when compared to such classical methods as fault and event trees is that DFM is "capable of modeling some of the essential logic and dynamic characteristics of hardware/software/firmware/process interactions which are difficult to represent" using the latter [3]. Thus, a viable way to describe and analyze DFM models programmatically should be of great interest to analysts, researchers and teachers in the reliability analysis field.

A drawback in the practical implementation and teaching of DFM is that relatively few software alternatives are available to the aspiring applier of DFM. The present author knows of only two DFM implementations. DYMONDA [4] is a full-fledged commercial implementation of DFM; however, its price may hinder its adoption to educational and research purposes, and the unavailability of its source code makes it difficult to experiment with DFM extensions etc. A DFM implementation developed at VTT (the workplace of the present author), YADRAT [5], is not available outside of VTT.

It seems therefore that a need exists for a programmatic tool for the definition and solution of DFM models. Since no open-source implementation of DFM is currently publicly available, a viable alternative is to find a method to implement DFM models programmatically from scratch. It is natural to set some requirements for such a method as follows:

- the description of DFM models should be simple and natural from the DFM point of view
- the method should include means of finding the solution of DFM models
- the language in which DFM models are described should be readily available, preferably with open-source implementations

[1] VTT, Box 1000 (VM3), FI-02044 VTT, Finland. e-mail: ilkka.karanta@vtt.fi

This paper describes a DFM model implementation method that satisfies these requirements, based on logic programming and the most commonly used logic programming language, Prolog.

The paper is organized as follows. In section 2 we take an elementary look at DFM; the basic concepts of DFM modeling and analysis are explained. In section 3 we introduce the basic concepts of logic programming, with a view of reliability modeling for industrial systems. Section 4 presents a method of implementing DFM models with Prolog. Section 5 considers the problem of calculating the probability of the top event, given the corresponding set of prime implicants; an algorithm is given for the purpose. Section 6 describes a small example, the reliability analysis of a simple pump, that illustrates the concepts outlined in the previous sections. Section 7 puts the results in context, showing how they might be used in practice and describing the relative merits of using Prolog. Finally, section 8 concludes.

2. The dynamic flowgraph methodology

The dynamic flowgraph methodology (DFM) is an approach to modeling and analyzing the behavior of dynamic systems for reliability/safety assessment and verification [1]. DFM models express the logic of the system in terms of causal relationships between physical variables and states of the control systems; the time aspects of the system (execution of control commands, dynamics of the process) are represented as a series of discrete state transitions. DFM can be used for identifying how certain postulated events may occur in a system; the result is a set of prime implicants (multi-state analogue of minimal cut sets) that can be used to identify system faults resulting e.g. from unanticipated combinations of software logic errors, hardware failures, human errors and adverse environmental conditions. DFM has been used to assess the reliability of nuclear power plant control systems [6], but also of space rockets [7] and chemical batch processes [8].

DFM models are directed graphs, analyzed at discrete time instances. They consist of variable and condition nodes; causality and condition edges; and transfer and transition boxes. Each node represents a variable that may be in one of two or more given states at each time instance. The state of a given node at a given time instance depends on the states of its input nodes; this dependence is given for each variable as a decision table. The state of a variable can change at discrete time instances; it is determined by the states of its input variables. Each node can have several inputs but only one output — the state of its variable at the time instance considered. The state of the variable can act as an input to possibly several other variables. The state of a node at time t is determined by

- the states of its input nodes at a single instance of time (say, $t - n$)
- the lag n, an integer that tells how many time instances it takes for an input to cause the state of the present node.

The state of a variable — as a function of the states of its input variables — is determined by a decision table. A decision table is an extension of the truth table where each variable can obtain one of any finite number of states. The decision table contains a row for each possible combination of input variable states. Each row tells what state the variable takes, given that its inputs take the given values. The maximum possible number of rows in the decision table is the product of the numbers of states of the input nodes.

There might be variables in the model that don't have input variables at all; the state of such a variable at any time instance is not determined by anything (except perhaps boundary conditions), and thus can be assigned to any of its possible values. In the following, we call such variables independent variables, and variables that have input variables (and whose state is thus determined by a decision table) dependent variables. Independent variables often represent chance events, and in particular, component failures.

We use a simplified notation that differs somewhat from the notation introduced in [1]; this simplification doesn't affect the semantics of the models. The distinction between causality and condition edges, and the distinction between condition and variable nodes, are for documenting purposes only, and they are therefore ignored in this paper. However, we distinguish the difference between independent and dependent variables: the former are marked by a square, and the latter by a circle. Boxes exist for representing the decision tables; however, decision tables may just as well be assigned to nodes directly, and therefore boxes can be dropped from the diagrams. The time lags associated with transition boxes are, in the simplified notation, attached to the edges.

After construction, the DFM model can be analyzed in two different modes, deductive and inductive [8]. In inductive analysis, event sequences are traced forward in time, from causes to effects. This corresponds to simulation of the model, or the generation of an event tree. In deductive analysis, event sequences are traced backward in time, from effects to causes; this corresponds to the generation of a fault tree. In this paper, we will consider only deductive analysis, or the task of finding the possible reason(s) of a system's failure. However, inductive analysis can be implemented in a logic program, too.

A deductive analysis starts with the identification of a particular system condition of interest (a top event); usually this condition corresponds to a failure. To find the root causes of the top event, the model is backtracked for a predefined number of time steps through the network of nodes, edges, transfer and transition boxes. This means that the model is worked backward in the cause-and-effect flow (that is, backward in time) to find what states of variables - and at what time instances - are needed to produce the top event. A useful analogy is that deductive analysis corresponds to minimal cut set search of a fault tree. The result of a deductive analysis is a set of prime implicants, $PI=\{PI_1,PI_2,\ldots,PI_n\}$.

A prime implicant PI_i is a set of triplets (V, S, T), also known as literals [9]; a literal is a claim that variable V is in a state S at time T. When all the literals in the prime implicant are true, the top event results; furthermore, dropping any literal out from the prime implicant would produce a set of literals that would not cause the top event. Prime implicants are similar to minimal cut sets of fault tree analysis, except that prime implicants are timed and they deal with multi-valued variables (fault trees deal with Boolean variables).

The prime implicants are minimal descriptions of conditions that lead to the top event. Furthermore, the value of a dependent variable depends deterministically on the values of its input variables, that is, a given combination of input variable values for a dependent variable always leads to the value for the variable given by the decision table. Therefore only values of the independent variables and the initial values of the dependent variables leading to the top event are listed in the prime implicant.

The following two definitions will be referred to in sections 5 and 6. A literal $L_1 \equiv (V_1,S_1,T_1)$ is said to be consistent with literal $L_2 \equiv (V_2,S_2,T_2)$ if the following condition holds: $V_1 = V_2 \wedge T_1 = T_2 \Rightarrow S_1 = S_2$. This means that all other pairs of literals are consistent except those where the same variable V_1 obtains a different state at the same time instance.

A prime implicant PI_1 is said to be mutually consistent with another prime implicant PI_2, if all of the literals of PI_1 are consistent with all the literals of PI_2. This means that the two prime implicants can hold simultaneously. If two prime implicants are not mutually consistent, they are said to be mutually inconsistent. For example, let $PI_1 = \{(temperature, hot, -2), (cooling, low, -1)\}$ and $PI_1 = \{(temperature, medium, -2), (cooling, off, -2)\}$. These two prime implicants are mutually inconsistent, because the variable *temperature* cannot be both *hot* and *medium* at time -2.

After *PI* has been determined, the probability of the top event can be computed. We will come back to this in section 5.

2.1. A small example

Consider the following system. A pump pumps water into a vessel from an infinite source, and the water flows out from the vessel. The flow through the pump is controlled by a water level measurement so that if there is too much (or an adequate amount of) water, the pump stops, and if there is too little water, the pump starts. To keep the system as simple as possible, we don't consider the water level measurement directly but assume that it always measures the true water level in the vessel, and transfers this information to the pump without delay. The goal of the system is to keep the water level appropriate.

Assume that there is only one way that the components of the system can fail: the pump can stop working (i.e. the flow is zero no matter what is should be). This failure mode is represented by the independent variable *pumpFailure*.

We now represent the system in DFM. We will use a simplified notation where boxes are not represented, and there are no variable or condition nodes, or causality or condition edges; instead, the notation distinguishes independent nodes from dependent nodes by denoting the former with a square and the latter with a circle (Figure 1). The time delays are marked in the edges; when the delay is 0, the delay is not marked.

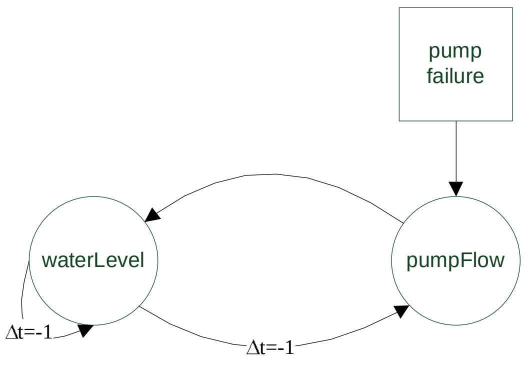


Figure 1. A DFM model of a simple vessel-and-pump system

The variable *waterLevel* may be in one of three possible states: *low*, *normal* or *high*. The variable *pumpFlow* has only two distinct possible states in this model: *zero* and *full*. *PumpFailure* may be in one of two distinct states: *no* and *yes*. The decision table of the variable *waterLevel* is as follows:

Table 1. The decision table of the variable waterLevel.

waterLevel(-1)	pumpFlow	waterLevel
low	zero	low
normal	zero	low
high	zero	normal
low	full	normal
normal	full	high
high	full	high

The decision table of the variable *pumpFlow* is as follows:

Table 2. The decision table of the variable pumpFlow.

waterLevel(-1)	pumpFailure	pumpFlow
low	no	full
normal	no	full
high	no	zero
low	yes	zero
normal	yes	zero
high	yes	zero

Some possible literals of the pump system are, for example, (*pumpFailure*,*no*,-1), and (*waterLevel*,*normal*,-2). The previous literal expresses the statement that at time -1 (one time step before the origin), the value of *pumpFailure* is “no”; in other words, the pump has not failed at that time.

This example will be further developed in section 6.

3. Logic programming and Prolog

In logic programming [15], a program consists of a description of the logical relations present in the problem to be solved. The program doesn’t contain an explicit description of a solution method. The logic programming system tries to solve the problem using deduction. In the following, we will use the nomenclature of [16].

The basic structure of a logic program is called a term. A term may be a constant, a variable or a compound term. A compound term consists of a functor and a sequence of one or more arguments. For example, the compound term `status(valve1, broken)` consists of the functor `status` and the arguments `valve1` and `broken`. This compound term might be used to indicate the condition that the status of the valve `valve1` is broken.

A logic program consists of statements called clauses (also known as rules). Each clause is a universally quantified logical sentence of the form

$$A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_k \tag{1}$$

where *A*, $B_1, \ldots, B_k$ are terms. The left-hand side of the rule, *A*, is called the head, and the conjunction on the right-hand-side is known as the body. The clause is read declaratively: "*A* is implied by the conjunction of $B_1, \ldots, B_k$ ". For example, we might want to state that a certain pump works if it is not broken and if it gets electricity. We could express this as the clause `works(pump1)`←`status(pump1,not_broken)`∧`gets_electricity(pump1)`.

If the right-hand side of (1) is empty (i.e. no terms are needed to imply *A*), the sentence is called a fact, and consists of the term *A* alone. For example, we might want to state the fact that the pipe `pipe1` feeds the vessel `vessel1` by stating the fact `feeds(pipe1,vessel1)`.

Once we have a logic program, we want to know if a certain state of affairs is implied by the program. This state of affairs is represented by a query. A query is a logical conjunction of terms. For example, we might want to know if both pump1 and pump2 work; a query for finding this out could be `works(pump1)`∧ `works(pump2)`.

The computation of a logic program is a constructive proof of a goal statement (query) from the program. The logic programming system produces this computation automatically once the user has typed in the query. In practice, the system starts with the top level goal (query) and expands it to lowel level goals via clauses until it has satisfied all the compound terms in the query. If the query contains free variables, the answer that the system outputs is a combination of bindings for the variables that satisfies the query; the system outputs as many (even all) answers to the query as the user wishes. If the query doesn't contain free variables, the system outputs yes, if the goal is satisfiable, or no if it isn't.

The Prolog programming language is an implementation of some basic logic programming constructs, including the ones introduced above. Additionally, Prolog has some extra-logical constructs to facilitate the programmer's work. Several Prolog implementations exist and some of those are freely available, such as SWI-Prolog [17] and Ciao Prolog [18].

There are some aspects to Prolog's syntax that are worth mentioning here. First, variables are written starting with a capital letter and constants starting with a lowercase letter; thus, in `feeds(Pipe,vessel)` `Pipe` is a variable but `vessel` is a constant. Second, the logical conjunction is denoted by a comma (,).

4. Representation of DFM models in Prolog

A DFM model can be represented as a logic program in a relatively compact way. The basic ideas are that a DFM literal is represented by a Prolog term; that a single row in a decision table is represented by a Prolog clause; and that these representations carry also the necessary information about maximum number of time steps and the prime implicant under construction.

4.1. Representation of decision tables

Each row in a decision table corresponds to a clause in Prolog. Let the row considered be

x1	Δt1	x2	Δt2	…	xn	Δtn	y
value1	lag1	value2	lag2	…	valuen	lagn	valuey

If only the information given in the decision table needs to be represented, this row can be expressed in Prolog as

```
literal(y, valuey,T)  :- T1 is T-lag1, literal(x1, value1,T1),
                         T2 is T-lag2, literal(x2, value2,T1),…,
                         Tn is T-lagn, literal(xn, valuen,Tn).
```

However, we need to express also some logical and computational information concerning each row in the decision table:

- we need to construct a prime implicant. To do that, we need a place to store the literals that have so far been found to belong to the prime implicant. A natural way of doing this is to represent the prime implicant as a list of literals, stored in a variable. The prime implicant is then constructed by accumulating the relevant literals to the variable. The value of this variable can then be given as an answer to the top level query (a top event).
- we need to stop the computation when the maximum number of time steps, defined by the user, has been taken. To do that, we need to compare the current number of time steps to the maximal number.
- we need a way to guarantee that the result is consistent (see section 2). In practice, since state transitions in DFM are deterministic, we need to consider only the initial values and values of independent variables (although it might be more efficient to check consistency when a single literal is generated).

The accumulation of a solution is best done by inserting two variables in a literal: one that contains the solution so far, and one that contains the insertions resulting from the application of the current rule. Taking care that the number of time steps taken does not exceed the maximum, a simple test is sufficient.

```
literal(y,val,T,Tmax,L1,Ln) :- T                >                 Tmax,
                               T1 is T-lag1, literal(x1, val1,T1,L1,L2),
                               T2 is T-lag2, literal(x2, val2,T1,L2,L3),
                               …,
                               Tn is T-lagn, literal(xn, valn,Tn,Lm,Ln).
```

The list L1 is the collection of literals collected so far (when this row in the decision table is called). Ln is the collection of literals that results from the application of the literals in this row of the decision table. Each application of a literal takes the result of the application of the previous literal and produces a new list that contains the previous literals and whatever new literals the application of the literal created.

Consistency could be checked for each rule. However, this can be inefficient. A more efficient method is to check the consistency only when needed, i.e. when generating an initial value of a dependent variable or a value of an independent variable. We'll consider consistency in section 4.2.

4.2. Representation of initial and boundary values

If the maximum number of time steps has been taken in the analysis, the value of the present variable is taken as such; it is an initial value that tells what value the variable must take initially in order to produce the prime implicant being constructed.

Representing initial values is straightforward: just verify that the initial value is consistent with the initial values collected so far, and if it is, insert it to the list of literals collected so far. The initial value is consistent with the literals collected so far if the literals do not contain one with the same variable and time instance but a different value. The representation of an initial value of a dependent variable is as follows:

```
literal(y,val,T,Tmax,L1,[i(y,val,T)|L1]) :- T         ≤          Tmax,
                                            consistent(i(y,val,T),L1).
```

In practical implementation, it is advantageous to avoid inserting the same literal many times in a prime implicant; to avoid this, a separate clause is needed that does not insert anything into the prospective prime implicant but does not interfere with the computation, either:

```
literal(y,val,T,Tmax,L,L) :- T ≤ Tmax, find(i(y,val,T),L).
```

A boundary value is a value that the variable necessarily takes at a given time instance. If the variable would take that value at every time instance, its representation would be easy: just drop all lines not containing the value from the decision table. However, because we want the variable to obtain the value only at the given time instance, a different strategy is needed.

Boundary values tell two things. First, that a variable has a certain value at a certain time instance. And second, that it cannot have any other values at that moment. From the logic programming point of view, this means that attempts to try any other value for the variable at the given moment should fail.

Thus, boundary value is represented in three parts. In the first part, the corresponding row of the decision table is given.

```
literal(y,val,t,Tmax,L1,Ln) :- T                 >                 Tmax,
                               T1 is T-lag1, literal(x1, val1,T1,L1,L2),
                               T2 is T-lag2, literal(x2, val2,T1,L2,L3),
                               …,
                               Tn is T-lagn, literal(xn, valn,Tn,Lm,Ln),
                               !.
```

Note that the time instance is represented by a small `t`; here `t` is not a variable but the constant specific time instance of the boundary value.

In the second part, the initial value of y is represented as above (if t ≤ Tmax).

In the third part, we ensure that an attempt to try any other values for y at the time instance t leads to failure. This is done by using the common idiom of negation in Prolog: if the condition that some other value for the variable `y` at the time instance `t` is tried, no other alternatives are tried and the present try is a failure:

```
literal(y,Val,t,_,_,_) :- Val ≠ val1,!,fail.
```

It is easy to extend this representation to cases where the variable at a certain time instance must obtain a value from a predefined set of values (e.g. the value of valve1position should be closed or semi-closed at time -1). The idea is that the predicate `literal(valve1position,-1,X)` should fail if `X` isn't one of `closed`, `semi-closed`.

4.3. Representation of independent variables

Independent variables don't have decision tables, and therefore don't need multiple clauses for representation. If there is no boundary condition for the independent variable at a time instance, the facts represent the conditions that the variable obtains any value from its value range at a given time instance. Thus they are of the form presented earlier for initial values, with the exception that here there is no need to check whether the number of time steps has been exceeded. Thus, independent variables are represented as follows:

```
literal(y,val,T,Tmax,L1,[i(y,val,T)|L1]) :- consistent(i(y,val,T),L1).
```

4.4. Representation of a top event and finding all its prime implicants

Representation of top events in DFM is straightforward in Prolog: in DFM, top events are conjunctions of DFM literals, so they can be represented in Prolog with a conjunctive query (which is a conjunction of clauses). The query is of the form

```
literal(y1,val1,t1,tmax,[],L1), literal(y2,val2,t2,L1,L2),…,
literal(yn,valn,tn,tmax,Ln,Result).
```

Note that the lists representing the prime implicant constructed so far are chained also in the query: the result of the first literal, L2, is handed over to the second literal and so forth. We start with no literals in the prime implicant; this is expressed in Prolog as the empty list ([]).

Querying Prolog with the top event itself is logically correct, but results in getting the prime implicants one at a time (the user has to return a semicolon after each prime implicant to get more). If the intent is to find all the prime implicants with a single query, one can do as follows. The query is first expressed as a rule

```
top_event(tmax,Result)  :-  literal(y1,val1,t1,tmax,[],L1),
                            literal(y2,val2,t2,L1,L2),
                            …,
                            literal(yn,valn,tn,tmax,Ln,Result).
```

Different names for the head of the rule can of course be used for different top events.

Using the rule above, all the prime implicants of the top event can be found using the Prolog standard predicate `findall` to query as follows:

```
findall(I,top_event(tmax,I),L).
```

Now the Prolog system returns all the prime implicants as lists in the list L.

5. Calculating top event probability

We consider the calculation of the top event probability, given the probabilities of literals occurring in the top event's prime implicants. The top event occurs if and only if at least one of its prime implicants occurs. Thus, if the set of the top event's n prime implicants is $PI=\{ PI_1, PI_2, \ldots, PI_n \}$, we want to find the probability $P(Top) = P(PI_1 \vee PI_2 \vee \ldots \vee PI_n)$.

It seems that this problem has not received much attention in the DFM context. [6], [10], [11] and [12] refer to a method based on converting PI into a set of mutually exclusive implicants (MEI). Due to the exclusivity, the probability of the top event is the sum of the probabilities of the individual MEI's (the method of conversion is not described in these papers). A different approach – based on the inclusion-exclusion development – is adapted here.

There are several methods for finding the probability of the top event, given the set of minimal cutsets [13]. The approach based on inclusion-exclusion development is the simplest of these, and it allows a computationally efficient implementation.

The inclusion-exclusion development utilizes the familiar way of computing the probability of a disjunction by the following expansion:

$$P\left(PI_1 \vee PI_2 \vee \ldots \vee PI_n\right) = \sum_{i=1}^{n} P\left(PI_i\right) - \sum_{\substack{i,j \\ i<j}} P\left(PI_i \wedge PI_j\right) + \ldots + \left(-1\right)^{n-1} P\left(PI_1 \wedge \ldots \wedge PI_n\right) \tag{2}$$

The proposed algorithm for computing this probability is based on the observation that, essentially, each conjunction on the right hand side of (2) is a conjunction of a subset of *PI*. Altogether, all subsets of *PI* are included in the sum in this way. The sought probability is an alternating sum of the probabilities of these conjunctions. The probability that constitutes a summand of the sum is the probability that all the prime implicants in the present subset hold. The sign of a summand is determined by the number of prime implicants in it.

The algorithm consists of systematically constructing the subsets of PI, finding their probabilities, and summing these probabilities. However, before we proceed, we must consider two issues. The first concerns the meaning of the event $PI_i \wedge PI_j$, or indeed the meaning of any conjunction of prime implicants; it is obvious from (2) that we need to be able to represent these events in terms of literals in order to calculate *P(Top)*. The second concerns the consequences of having a pair of mutually inconsistent prime implicants in *PI*. After these issues have been dealt with, we proceed to present an algorithm for the computation of the top event probability.

5.1. The meaning of conjunction of prime implicants

Each prime implicant is a set of literals. Therefore, the event $PI_i \wedge PI_j$ happens when all the literals of PI_i and all the literals of PI_j hold. This means that the set of literals that must hold for $PI_i \wedge PI_j$ to be true is the set-theoretical union $PI_i \cup PI_j$.

For example, let $PI_1 = \{(temperature, hot, -2), (cooling, low, -1)\}$ and $PI_2 = \{(temperature, hot, -2), (alarm, off, -2)\}$. Then the event $PI_1 \wedge PI_2$ holds if and only if all the literals in the set $\{(temperature, hot, -2), (cooling, low, -1), (alarm, off, -2)\}$ hold.

Therefore, when we are finding out the probability of any conjunction of prime implicants, it suffices to consider the probability that all the literals in the corresponding union of the individual prime implicants hold.

This property can be put into good use when calculating the sum (2). We do not have to keep track of the prime implicants involved in a conjunction; it suffices that we maintain a set containing the literals in them.

5.2. Handling of mutually inconsistent prime implicants

In some cases, two prime implicants cannot hold simultaneously. We consider how to take this into account in the calculation of the top event probability.

There are several ways that two or more prime implicants cannot hold simultaneously: this might be the case for logical, physical or other reasons. For example, in some contexts, it might be the case that a component stays in failed state once it fails; then, a prime implicant PI_1 containing a literal that states failure of the component at a time instance (*component,failed,*t_1), and another prime implicant PI_2 containing a literal (*component,working,*t_2) cannot hold simultaneously if $t_1 < t_2$. In this paper, we consider only cases where two prime implicants are inconsistent - that is, cannot hold simultaneously for logical reasons. However, other kinds of cases (two prime implicants cannot hold simultaneously for physical reasons etc.) can be handled similarly.

If the prime implicants PI_i and PI_j are mutually inconsistent, the conjunction $PI_i \wedge PI_j$ (or indeed any conjunction containing both PI_i and PI_j) is impossible in the sense that it cannot ever hold. Therefore its probability $P\left(PI_i \wedge PI_j\right) = 0$.

This fact can be put to good use when calculating the top event probability. Namely, when we are constructing subsets of *PI* for the calculation of (2), we can omit all subsets that contain any pair of mutually inconsistent prime implicants. This can dramatically reduce the number of subsets of *PI* that have to be considered. For example, in a space-based reactor control system presented in [14], the number of prime implicants is 8, and therefore the number of non-empty subsets of *PI* to be considered in (2) is 2^8-1=255. However, some of the prime implicants are mutually inconsistent (due to, e.g., one prime implicant having the literal (*TS,low,-1*), and another implicant having (*TS,null,-1*)), and the number of nonempty subsets of *PI* that actually have to be considered is 18.

5.3. An algorithm for computing the probability of top event by inclusion-exclusion development

The algorithm is based on a simple way of listing all the subsets of a given set: for the current element, find all the subsets containing it, and all the subsets not containing it. This basic scheme has been modified to take into account that subsets containing any pair of mutually inconsistent prime implicants need not be listed. The algorithm also maintains a list of literals rather than a list of prime implicants, and the probability of a conjunction (corresponding to a subset of prime implicants) is calculated for this list of literals. The algorithm is presented in Figure 2.

```
 1 function probability_by_inclusion_exclusion(RemainingImplicants,
 2                                             SubsetSoFar,
 3                                             Sign) returns P

 4   if RemainingImplicants == null then
 5      P := Sign * probability_of_intersection(SubsetSoFar)
 6      return P
 7   end if

 8   CurrentImplicant := pick_one_of(RemainingImplicants)
 9   RestImplicants   := RemainingImplicants except CurrentImplicant

10   if consistent(CurrentImplicant,SubsetSoFar) then
11      NewSubset  := union(CurrentImplicant,SubsetSoFar)
12      Sign1      := (-1) * Sign
13      P1         := probability_by_inclusion_exclusion(RestImplicants, NewSubset, Sign1)
14   else
15      P1 := 0
16   end if

17   P2 := probability_by_inclusion_exclusion(RestImplicants, SubsetSoFar,Sign)

18   P := P1 + P2
19   return P
20 end function

21 function probability_of_top_event(PrimeImplicants)
22   return probability_by_inclusion_exclusion(PrimeImplicants, null, -1)
23 end function
```

Figure 2. An algorithm for computing the probability of a top event from its prime implicants by the inclusion-exclusion development

Let us first consider the function `probability_of_top_event` (lines 21-23). It receives a list of prime implicants as its inputs and returns the probability of the corresponding top event. However, it is just an interface to the recursive function `probability_by_inclusion_exclusion`.

The function `probability_by_inclusion_exclusion` is the heart of the algorithm. Its inputs are

- `RemainingImplicants`, a set of prime implicants that have not been processed yet,
- `SubsetSoFar`, the subset of prime implicants constructed so far (expressed as a set of literals),
- `Sign`, the sign of the current summand (+1 or -1).

It returns a partial sum of (2) that consists of the sum of those conjunctions (or subsets of *PI*) that include the prime implicants in `SubsetSoFar`.

In lines 4-7, we consider the case that we have actually processed all the prime implicants, either by taking them into the current subset or not. In this case, the set of remaining implicants is empty (in other words, null), and we can calculate the probability that all the prime implicants in the current subset hold. After calculating this probability, control is returned to the calling function.

Let the current subset be $\left\{PI_{i1}, PI_{i2}, \ldots, PI_{ik}\right\}$ (and thus the size of the current subset be *k*). The function `probability_of_intersection` calculates the probability that all these prime implicants hold. This probability is, of course, the probability that all the literals in the set $L_i = \left\{L \middle| \exists PI_j \in \left\{PI_{i1}, PI_{i2}, \ldots, PI_{ik}\right\} \text{ such that } L \in PI_j\right\}$ hold. How this probability is calculated depends on the system considered. In the simplest case (implemented in section 6) the probabilities of each literal holding are independent, and the sought probability is the product of the individual probabilities of literals. If, e.g., common cause faults need to be taken into account, the calculation of the probability will reflect this.

Consider the example in section 5.1. Let the probabilities of the literals be $P\left((temperature, hot, -2)\right) = 0.2$ (that is, the probability that the temperature is hot at time instance -2 equals 0.2), $P\left((cooling, low, -1)\right) = 0.3$, and $P\left((alarm, off, -2)\right) = 0.7$. The probability $P\left(PI_1 \wedge PI_2\right) = P\left((temperature, hot, -2) \wedge (cooling, low, -1) \wedge (alarm, off, -2)\right)$. Assuming independence, this probability is $P\left(PI_1 \wedge PI_2\right) = P\left((temperature, hot, -2)\right) P\left((cooling, low, -1)\right) P\left((alarm, off, -2)\right) = 0.042$.

In lines 8-9 we choose one of the remaining prime implicants (called `CurrentImplicant`). The set of prime implicants still to be considered will then be `RemainingImplicants` with `CurrentImplicant` removed from it; call this new set `RestImplicants`.

In lines 10-16 we take into account the special features of DFM explained in sections 5.1 and 5.2. If `CurrentImplicant` is consistent with `SubsetSoFar`, we find the probability of the subsets containing both `SubsetSoFar` and `CurrentImplicant`. The sign is reverted on line 12, because the sign of the summands in the sum (2) alternates with the number of prime implicants included. If `CurrentImplicant` is not consistent with `SubsetSoFar`, (that is, at least one of the literals of `CurrentImplicant` is inconsistent with some literal in `SubsetSoFar`), the probability of a subset constructed from `SubsetSoFar` and `CurrentImplicant` equals 0, and no subsets containing both `CurrentImplicant` and `SubsetSoFar` need to be generated.

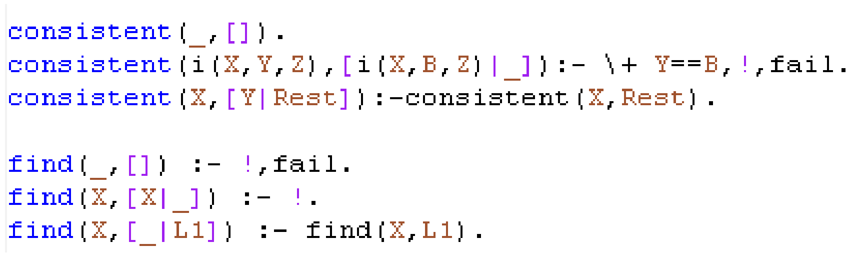
In line 17 we simply find the probability of `SubsetSoFar` appended with, in turn, all the subsets of `RestImplicants`. The crucial point is that `CurrentImplicant` is not included (because it was included in lines 10-16).

Finally, in line 18, we sum the two mutually exclusive probabilities (they are mutually exclusive because one is for sets containing `CurrentImplicant` and the other for sets not containing it). This sum is the probability that the current call to `probability_by_inclusion_exclusion` returns.

A Prolog implementation of the algorithm is presented in section 6.

6. An application example

To illustrate the model representation formalism introduced above, let us consider the simple pump-and-vessel system introduced in section 2.1. The model was implemented in SWI-Prolog. First, we need two auxiliary Prolog predicates: one to guarantee the consistency of the prime implicants generated, and another to make the generated prime implicant non-redundant by detecting whether a literal has already been inserted to it. These auxiliary predicates are as follows.



```
consistent(_,[]).
consistent(i(X,Y,Z),[i(X,B,Z)|_]):- \+ Y==B,!,fail.
consistent(X,[Y|Rest]):-consistent(X,Rest).

find(_,[]) :- !,fail.
find(X,[X|_]) :- !.
find(X,[_|L1]) :- find(X,L1).
```

Figure 3. Auxiliary predicates for computing the prime implicants of a DFM model. `consistent` checks the consistency of a literal with a list of literals; `find` checks whether an element is in the given list.

Now we are in a position to express the DFM model itself.

```
% the initial values of waterLevel
literal(waterLevel,Y,T,Tmax,L,L) :- T=<Tmax, find(i(waterLevel,Y,T),L),!.
literal(waterLevel,Y,T,Tmax,L,[i(waterLevel,Y,T)|L]) :-
        T=<Tmax, consistent(i(waterLevel,Y,T),L).
% the decision table of waterLevel
literal(waterLevel,low,T,Tmax,L,Res) :- T>Tmax, T1 is T-1,
                                        literal(waterLevel,low,T1,Tmax,L,L1),
                                        literal(pumpFlow,zero,T,Tmax,L1,Res).
literal(waterLevel,low,T,Tmax,L,Res) :- T>Tmax, T1 is T-1,
                                        literal(waterLevel,normal,T1,Tmax,L,L1),
                                        literal(pumpFlow,zero,T,Tmax,L1,Res).
literal(waterLevel,normal,T,Tmax,L,Res) :- T>Tmax, T1 is T-1,
                                        literal(waterLevel,high,T1,Tmax,L,L1),
                                        literal(pumpFlow,zero,T,Tmax,L1,Res).
literal(waterLevel,normal,T,Tmax,L,Res) :- T>Tmax, T1 is T-1,
                                        literal(waterLevel,low,T1,Tmax,L,L1),
                                        literal(pumpFlow,full,T,Tmax,L1,Res).
literal(waterLevel,high,T,Tmax,L,Res) :- T>Tmax, T1 is T-1,
                                        literal(waterLevel,normal,T1,Tmax,L,L1),
                                        literal(pumpFlow,full,T,Tmax,L1,Res).
literal(waterLevel,high,T,Tmax,L,Res) :- T>Tmax, T1 is T-1,
                                        literal(waterLevel,high,T1,Tmax,L,L1),
                                        literal(pumpFlow,full,T,Tmax,L1,Res).
% initial values of pumpFlow
literal(pumpFlow,Y,T,Tmax,L,L) :-     T=<Tmax, find(i(pumpFlow,Y,T),L),!.
literal(pumpFlow,Y,T,Tmax,L,[i(pumpFlow,Y,T)|L]) :-
        T=<Tmax, consistent(i(pumpFlow,Y,T),L).
% the decision table of pumpFlow
literal(pumpFlow,full,T,Tmax,L,Res) :- T>Tmax, T1 is T-1,
                                        literal(waterLevel,low,T1,Tmax,L,L1),
                                        literal(pumpFailure,no,T,Tmax,L1,Res).
literal(pumpFlow,full,T,Tmax,L,Res) :- T>Tmax, T1 is T-1,
                                        literal(waterLevel,normal,T1,Tmax,L,L1),
                                        literal(pumpFailure,no,T,Tmax,L1,Res).
literal(pumpFlow,zero,T,Tmax,L,Res) :- T>Tmax, T1 is T-1,
                                        literal(waterLevel,high,T1,Tmax,L,L1),
                                        literal(pumpFailure,no,T,Tmax,L1,Res).
literal(pumpFlow,zero,T,Tmax,L,Res) :- T>Tmax, T1 is T-1,
                                        literal(waterLevel,low,T1,Tmax,L,L1),
                                        literal(pumpFailure,yes,T,Tmax,L1,Res).
literal(pumpFlow,zero,T,Tmax,L,Res) :- T>Tmax, T1 is T-1,
                                        literal(waterLevel,normal,T1,Tmax,L,L1),
                                        literal(pumpFailure,yes,T,Tmax,L1,Res).
literal(pumpFlow,zero,T,Tmax,L,Res) :- T>Tmax, T1 is T-1,
                                        literal(waterLevel,high,T1,Tmax,L,L1),
                                        literal(pumpFailure,yes,T,Tmax,L1,Res).
% the independent variable pumpFailure
literal(pumpFailure,Y,T,_,L,L) :- find(i(pumpFailure,Y,T),L),!.
literal(pumpFailure,Y,T,_,L,[i(pumpFailure,Y,T)|L]) :-
         consistent(i(pumpFailure,Y,T),L).
```

Figure 4. The simple vessel-and-pump system represented in Prolog

Only the code contained in figures 3 and 4 is needed for the prime implicant computation.

To find the prime implicants of a top event, the top event is represented as a query. Consider the the top event that the water level is too low for two consecutive time instances. This can be represented as

```
literal(waterLevel,low,-1,Tmax,[],L), literal(waterLevel,low,0,Tmax,L,I).
```

Figure 5. Representation of a top event in Prolog

The number of time steps has to be chosen. In this case we may simply consider two time steps, and set Tmax thus to -2. In running the query, a Prolog system proceeds by trying to find bindings for the variables `L` and `I` (see Figure 5) that satisfy the facts and rules in the Prolog program (presented in figures 3-4). The system first takes the goal `literal(waterlevel,low,-1,-2,[],L)`, and tries to find a binding for the variable L. It goes through the facts and rules of the program until it finds a fact or a rule that satisfies this; in this case, it is the third rule in Figure 4 (also the fourth rule satisfies the goal, and this possibly provides an alternative solution). Using the third rule, the system finds that it has to satisfy the two goals `literal(waterlevel,low,-2,-2,[],L1)` and `literal(pumpFlow,zero,-1,-2,L1,L)`. Proceeding in this way, the Prolog system finds bindings for the variables and returns them as results (for a more complete explanation of how Prolog systems work, the reader is referred to [16]). The results of running the query of Figure 5 are shown in figure 6.

```
I = [i(pumpFailure, yes, 0), i(pumpFailure, yes, -1), i(waterLevel, low, -2)] ;
I = [i(pumpFailure, yes, 0), i(pumpFailure, yes, -1), i(waterLevel, normal, -2)]
```

Figure 6. Results of querying the top event in SWI-Prolog

As can be seen from the results, the top event is implied by two prime implicants. In both, there is pump failure on both time instances (that is, time instances 0 and -1). The difference between these prime implicants is that in the former, the initial value of waterLevel (that is, its value in time instance -2) is low, whereas in the latter it is normal.

For computing the probability of the top event, we need a Prolog implementation of the algorithm presented in section 5.3. This implementation is given in Figure 7.

```
probability_by_inclusion_exclusion([],Impls,Sign,P) :-
        probability_of_intersection(Impls,P1),
        P is Sign * P1.
probability_by_inclusion_exclusion([CurrentPI|RestPIs],SubsetSoFar,Sign,P) :-
        probability_with_this_included(CurrentPI,RestPIs,SubsetSoFar,Sign,P1),
        probability_by_inclusion_exclusion(RestPIs,SubsetSoFar,Sign,P2),
        P is P1 + P2.

probability_with_this_included(CurrentPI,RestPIs,SubsetSoFar,Sign,0) :-
        \+ consistent_implicants(CurrentPI,SubsetSoFar), !.

probability_with_this_included(CurrentPI,RestPIs,SubsetSoFar,Sign,P) :-
%       consistent_implicants(CurrentPI,SubsetSoFar),
        Sign1 is (-1)*Sign,
        union_of_literals(CurrentPI,SubsetSoFar,NewSubset),
        probability_by_inclusion_exclusion(RestPIs,NewSubset,Sign1,P).

probability_of(PrimeImplicants,P) :-
        probability_by_inclusion_exclusion(PrimeImplicants,[],-1,P).
```

Figure 7. Prolog implementation of the top event probability calculation algorithm presented in section 5.3

An implementation of the calculation of a single conjunction of prime implicants (represented as literals) is given in Figure 8. The probabilities are assumed to be independent, so the sought probability is the product of the probabilities of individual literals.

```
probability_of_intersection([],0).
probability_of_intersection(Impls,P) :-
        probability_of_intersection_aux(Impls,1,P).

probability_of_intersection_aux([],P,P).
probability_of_intersection_aux([Lit|Rest],P0,P):-
        p(Lit,Plit),
        P1 is Plit * P0,
        probability_of_intersection_aux(Rest,P1,P).
```

Figure 8. Prolog implementation of the probability of conjunction calculation

Finally, the implementation of the union of two sets of literals, and the consistency check for two prime implicants (expressed as two sets of literals) are given in Figure 9.

```
union_of_literals([],X,X).
union_of_literals([X|Rest],Y,Merged) :-
        member(X,Y), !, union_of_literals(Rest,Y,Merged).
union_of_literals([X|Rest],Y,[X|Merged]) :-
        union_of_literals(Rest,Y,Merged).

consistent_implicants([],_).
consistent_implicants([X|Y],Z) :-
        consistent(X,Z),
        consistent_implicants(Y,Z).
```

Figure 9. Auxiliary predicates needed by the top event probability calculation

Finally, we can package computation of the prime implicants and calculation of the top event probability into a compact interface of Figure 10. `probability_of_top_event` returns the probability of this particular top event.

```
find_all_prime_implicants(Nsteps,L) :-
        findall(I,top_event(Nsteps,I),L).

top_event(Nsteps,I):-
        Tmax is -Nsteps,
        literal(waterLevel,low,-1,Tmax,[],L),
        literal(waterLevel,low,0,Tmax,L,I).

probability_of_top_event(Nsteps,P) :-
        find_all_prime_implicants(Nsteps,L),
        probability_of(L,P).
```

Figure 10. Query interface for prime implicant computation and probability calculation.

Let the probabilities be as in Figure 11 (the underscore in the first clause is a don't-care variable binding – in practice it means that the probability of pump failure is 0.1 for all time instances).

```
p(i(pumpFailure,yes,_),0.1).
p(i(waterLevel,low,-2),0.2).
p(i(waterLevel,normal,-2),0.3).
```

Figure 11. Probabilities of literals for the pump-and-vessel system example

From Figure 6, we notice that the two prime implicants of the top event are mutually inconsistent because waterLevel cannot be both low and normal at the same time. Therefore the probability of the top event is, in this case, simply the sum of the probabilities of the individual prime implicants. It is easy to see that this probability is 0.005, and this is what the Prolog system gives.

7. Discussion

In section 6, we have seen how the formulation of a DFM model given in section 4 can be used in implementing the toy example of section 2.1. Larger DFM models can be implemented in the same way. The auxiliary predicates of Figure 3, the top event probability computation predicates of figures 7-9, and the query interface predicates of Figure 10 (which are not needed, but make using the program simpler) can be used as such. It is handy to collect these predicates in a single file, and represent the DFM model in a separate file; in this way, the model can easily be changed. In a single Prolog session the change proceeds by first by retracting the `literal` predicate (if the new model contains same variable as the previous model) and loading the file containing the new model, and between sessions simply by loading the file containing the desired DFM model.

On a risk analysis course covering DFM modeling and analysis, computer-based DFM exercises can now be incorporated with no cost and a small amount of effort on the lecturer's side. The auxiliary and top event computation predicates can be handed to the students in a file, and an example can be shown on how to implement a DFM model. Then the students can be given the task of modeling a system in DFM, implementing it in Prolog, and analyzing the solution. It is of course helpful if the students have been exposed to Prolog in, e.g. a course on artificial intelligence or logic programming; however, this is not necessary, because they need only to provide their own set of `literal` clauses, and in this they can follow the pattern and example provided in sections 4 and 6.

Rapid prototyping of DFM models proceeds in a similar manner. All that is needed is a representation of the DFM model as Prolog facts and rules containing `literal` predicates (as exemplified in Figure 4). This provides a simple and cost-free implementation of DFM models, available to anyone with the necessary modeling skills and a computer.

The implementation in section 6 demonstrates some advantages Prolog has, in the present context, over commonly used programming languages such as Java, C, C++ or Visual Basic. First of all, Prolog allows an easy and compact representation of a DFM model within Prolog itself. Expressing a DFM model in any of these languages would be cumbersome at best, and a better solution would be to express the model in a separate description language (which would bring the overhead of implementing a parser for that language). Second, Prolog offers in itself computation facilities that would have to be implemented if any other language were used. A case in point is the backtracking needed in deductive analysis of DFM models (see section 2): in Prolog, we have no need to implement this, but in another language it would have to be implemented programmatically. Third, Prolog allows for a clear and compact implementation of algorithms and computation: the implementation of the top event probability computation algorithm of Figure 2 takes 32 lines of code (see figures 7-9), not considerably more than the expression of the algorithm in pseudocode (23 lines of code). It is quite likely that the implementation of the algorithm in an ordinary programming language would take tens, if not hundreds of lines of code, including the implementation of list handling, set handling (calculation of the union of two sets), etc.

We haven't touched on the question of DFM modeling in general. It seems that there are no treatises concerning how to construct a DFM model, but the reader gain insight by studying the systems and their respective DFM models presented in, e.g., [1], [6], [7], [8], [10], [11] and [12].

8. Conclusion

We have presented a relatively straightforward method of representing DFM models in the logic programming framework, and a simple method for calculating the probability of a top event. In the representation, a DFM literal (consisting of a variable and its value at a given time instance) is denoted by a Prolog term. Each line of a variable's decision table is represented by a Prolog clause. Independent variables – i.e. variables that don't depend on other variables and thus don't have a decision table – and initial values of dependent variables are handled separately. A prime implicant is expressed as a list of DFM literals of independent variables and initial values of dependent variables. A top event is expressed as a conjunction of DFM literals.

This representation allows for a straightforward and relatively compact representation of DFM models. The prime implicants of a top event can easily be generated by expressing the top event as a Prolog query.

The method for the calculation of the probability of a top event, given its prime implicants, is based on the inclusion-exclusion development. It allows for a compact Prolog implementation.

Together, DFM model presentation, prime implicant computation and top event probability calculation form a complete framework for presenting and solving DFM models. This framework has at least two appropriate uses. In the rapid prototyping of small to medium-sized DFM models, it enables quick model construction and agile experimentation with alternatives in modeling a given system. In the teaching of DFM on risk analysis courses, it allows the students to rapidly construct simple DFM models of their own, based on learning from and modifying given example models.

A natural extension of the present framework would be to implement inductive analysis for the models. This might be based on forward-chaining [19]; however, it is likely that some program transformations [16] would have to be applied to the DFM model of interest first. Future directions in the representation of DFM models in logic programming are related to extending the DFM modeling paradigm by e.g. expressing general time constraints with constraint logic programming [20], or incorporating elements of process algebra [21] or stochastic activity networks [22]. The framework can be extended in the future to incorporate such central tasks of probabilistic risk analysis as the calculation of importance measures (see [23], [24]) and uncertainty analysis for DFM models.

Funding

This research was supported by the Fortum Foundation.

References

[1] Garrett, C., Guarro, S., Apostolakis, G. “The Dynamic Flowgraph Methodology for Assessing the Dependability of Embedded Software Systems”, IEEE Trans. on Systems, Man and Cybernetics. 25, 824-840, 1995.

[2] Aldemir T., Miller D.W., Stovsky M.P., Kirschenbaum J., Bucci P., Fentiman A.W., Mangan L.A. Current State of Reliability Modeling Methodologies for Digital Systems and Their Acceptance Criteria for Nuclear Power Plant Assessments. NUREG report NUREG/CR-6901, February 2006.

[3] Aldemir, T., Guarro, S., Mandelli, T., Kirschenbaum, J., Mangan, L.A., Bucci, P., Yau, M., Ekici, E., Miller, D.W., Sun, X., Arndt, S.A. Probabilistic risk assessment modeling of digital instrumentation and control systems using two dynamic methodologies. Reliability Engineering and System Safety. 95, 1011–1039, 2010.

[4] ASCA Incorporated. DYMONDA. URL http://ascainc.com/dymonda/dymonda.html (link referred January 11, 2012)

[5] Björkman, K., Karanta, I. A dynamic flowgraph methodology approach based on binary decision diagrams. Proceedings of 2011 International Topical Meeting on Probabilistic Safety Assessment and Analysis (PSA 2011), Wilmington, NC, 13 - 17 March 2011. American Nuclear Society 2011, 267-278.

[6] Aldemir T., Stovsky M.P., Kirschenbaum J., Mandelli D., Bucci P., Mangan L.A., Miller D.W., Sun X., Ekici E., Guarro S., Yau M., Johnson B., Elks C., and Arnd S.A. Dynamic reliability modeling of digital instrumentation and control systems for nuclear reactor probabilistic risk assessments. NUREG report NUREG/CR-6942, October 2007.

[7] Yau M., Guarro S. and Apostolakis G. Demonstration of the dynamic flowgraph methodology using the Titan II space launch vehicle digital flight control system. Reliability Engineering and System Safety. 49, 335-353, 1995.

[8] Houtermans M., Apostolakis G., Brombacher A. and Karydas D. Programmable electronic system design & verification utilizing DFM. In SAFECOMP 2000 (F. Koornneef and M. van der Meulen, eds.), Lecture Notes in Computer Science Vol. 1943, Springer 2000, 275-285.

[9] Houtermans M. A method for dynamic process hazard analysis and integrated process safety management. Doctoral thesis, Technische Universiteit Eindhoven 2001.

[10] Dixon S., Yau M., and Guarro S. Demonstration of the context-based software risk model method for risk informed assurance and test of software-intensive space systems. Paper 280 in International Probabilistic Safety Assessment and Management Conference (PSAM) 9, 18-23 May, 2008, Hong Kong, China.

[11] Yau M., Dixon S., Motamed M., and Guarro S. Application of the Dynamic Flowgraph Methodology to a digital instrumentation and control system benchmark study. Paper 408 in International Probabilistic Safety Assessment and Management Conference (PSAM) 9, 18-23 May, 2008, Hong Kong, China.

[12] Aldemir T., Guarro S., Mandelli D., Kirschenbaum J., Mangan L.A., Bucci P., Yau M., Ekici E., Miller D.W., Sun X., and Arndt S.A. Probabilistic risk assessment modeling of digital instrumentation and control systems using two dynamic methodologies. Reliability Engineering and System Safety, 95, 1011-1039, 2010.

[13] Limnios N. Fault trees. ISTE 2007.

[14] Garrett C., and Apostolakis G. Automated hazard analysis of digital control systems. Reliability Engineering and System Safety. 77, 1-17, 2002.

[15] Nilsson U. and Maluszynski J. Logic, programming and Prolog, 2nd edition. Wiley 1995.

[16] Sterling L. and Shapiro E. The art of Prolog – advanced programming techniques, 2nd edition. MIT Press 1994.

[17] Wielemaker J., Schrijvers T., Triska M. and Lager T. SWI-Prolog. Theory and Practice of Logic Programming. 12, 67-96, 2012.

[18] Hermenegildo M., Bueno F., Carro M., López-García P, Mera E., Morales J. and Puebla G. An overview of Ciao and its design philosophy. Theory and Practice of Logic Programming. 12, 219-252, 2012.

[19] Covington M., Nute D., and Vellino A. Prolog programming in depth. Prentice-Hall 1997.

[20] Marriott K., Stuckey P.J. and Wallace M. Constraint logic programming. Chapter 12 in F. Rossi, P. van Beek and T. Walsh (eds.), Handbook of Constraint Programming, Elsevier 2006, 409-452.

[21] Schuster, J., Siegle, M. Path-based calculation of MTTFF, MTTFR, and asymptotic unavailability with the stochastic process algebra tool CASPA. Journal of Risk and Reliability. 225, 399-406, 2011.

[22] Maza, S. Dynamic modelling and simulation of fault-tolerant systems based on stochastic activity networks. Journal of Risk and Reliability. 226, 455-463, 2012.

[23] Tyrväinen, T., Björkman, K. Modelling common cause failures and computing risk importance measures in the Dynamic Flowgraph Methodology. 11th International Probabilistic Safety Assessment and Management Conference and The Annual European Safety and Reliability Conference PSAM11/ESREL12, 25-29 June, 2012, Helsinki, Finland.

[24] Karanta, I. Importance measures for the Dynamic Flowgraph Methodology. Research report VTT-R-00525-11, VTT, Espoo, 2011.