Language Guide

Domain Specific Language for Tree-Based Specification Applications

Version 1.0

Denise Case
Kansas State University
Spring 2012
CIS 705 Final Project

Table of Contents

1 Introduction

This document describes a domain specific language written primarily to allow a single developer to prototype efforts quickly in order to evaluate options quickly and to provide some initial feedback before beginning a more painstaking graphical modeling endeavor.

The language was originally developed to assist with rapid specification of goal trees for multi agent systems. K-State's MACR lab has advanced graphical tools for specifying goal trees, but they currently require a somewhat painstaking installation process in Eclipse, which is a powerful but lumbering integrated development environment. I still use it for developing documentation, but much prefer the agility of a little language with a very small footprint.

When specifying organizational goals so that they can be distributed among various distributed agents, we break the high-level goals, starting with a single top-level goal, into sub-goals, joined either conjunctively or disjunctively depending on the relationship between the children. If all child goals are required, it is said to be a conjunctive join. If any one of the child goals is required, it is said to be a disjunctive goal. There are some additional relationships, for example, a goal may "precede" another goal, in that it must be satisfied first before progress towards the second goal can begin. Successful achievement of a goal may also "trigger" the activation of another goal. For example, if an intruder is detected, a location and identification process may be initiated.

I found I could test-drive even simple real-time goal instances from the tree specified quite quickly and was able to bypass a full java installation as well as our required MatLab integration. By simulating results quickly, I can select appropriate simulation goals in the production version.

Because of the success with the goal specification, I could see many parallels with another painstaking application that used frequently during my engineering consulting work. This effort uses a similar tree structure to describe the way equipment relates to overall system availability.

While the languages would eventually separate if they underwent additional development, the overlap has allowed me to test-drive very helpful applications in both areas. If I had to pick one area for continued development, I would definitely pursue the reliability specification.

1.1 Problem Domain

When modeling complex adaptive systems, software engineers talk about the system organization – the set of distributed intelligent software agents and the supporting architecture that makes up the overall solution. This organization includes the agents – as well as supporting entities necessary to shape and guide the desired behavior. Software engineers describe the overall objective of the entire system as its top level "goal" – which is

further broken down into a series of progressively more specific goals that can be achieved by portions of the system. Software components are called “agents”, and each of these is said to “possess” a specific set of “capabilities” – or sets of behaviors the agent can perform. To assist with managing the complexities of the system, the software agents are assigned to specific “roles” depending on the conditions of the system, the environment, and the active goals. Roles are designed specifically to “achieve” particular goals and each role “requires” specific capabilities.

Systems built only in software are modeled much the same as systems that employ specific physical agents, and it may be helpful to think of agents as independently functioning, potentially-breakable or -compromisable entities. Agents are designed to play roles that will achieve the system goals – and each agent class may be capable of playing different roles at different times, depending on the active goals. As in the real world, there are degrees of effectiveness as well. Primary roles can achieve a goal to full completion, but it may also be possible to assign goals to secondary roles (as when primary agents are incapacitated) and these secondary roles can partially –but not entirely – achieve a goal. In addition, not all agents may possess capabilities to the same degree. This is perhaps more obvious in the case of physical agents but it also reflects cases where specialized agents must be kept small to optimize memory space. The organization and the rules and guiding policies are specified at the onset, and in operation, the organization dynamically reconfigures itself in response to events and the environment. In complex adaptive systems, optimum design of the organization can be a significant software engineering effort.

Entities include goals, agents, roles, and capabilities. Operations include specifying the facts of the design models and initializing instances of the model. Events include the triggering of a goal, the successful achievement of a goal, the failure of an agent so that a goal cannot be achieved and the assignment of an agent to a role in order to achieve a specific goal. Features include the ability to include parameters when triggering goals, to be able to specify an indication of the degree to which a role can fulfill a goal, to be able to specify the level at which an agent possesses a specific capability, and the ability to create visual models of static specifications as well as generate simulations of actual instances of agents and goals that illustrate the dynamic adaptive organizational process that may result from the static specification.

1.2 Complex Adaptive Systems – Software Support

Testing the design must include the testing of the static specification of the organization, its components and policies – the part that does not change – as well as testing the dynamic behavior of the organization in response to the flow of time and events in the environment. Tools that can assist with the design and implementation of a highly functional organization can be very valuable. K-State has developed a graphical environment to assist with many of these tasks. Static aspects of the organization can be designed using UML-based models. Developing a little language for prototyping would integrate well with our existing framework and allow even quicker testing and evaluation of multiple early designs before the drawing

process begins. As design proceeds, the little language could be used to instantiate simulated trials of possible designs and provide an efficient means to simulate and evaluate the possible performance of the dynamic organization over time. Implementation in a little language would lay the foundation for an extending our current tools into a platform easily adopted by designers of complex, adaptive systems. It may even help provide a framework for testing the new holonic organization we are creating to provide a complex, adaptive control system for the evolving power grid in our Intelligent Power Distribution System project. The little language would offer a complementary alternative to the graphical option, and could be written to allow fluid transition between both views. The little language would facilitate development of the graphical depictions helpful for enhanced communication and assimilation "at-a-glance" while the power and speed of a concise little language implemented in a simple command line interface would allow for rapid prototyping and testing of possible designs.

There are many of the related software engineering tasks required during the design and implementation of an adaptive organization. A sample list of some of these tasks is provided below and these will be referred to in the next section, where we select specific scenarios or use cases in which to test the little language. Some of the tasks which must be performed include:

1. Designing the static goal model in an "and/or" tree structure.
2. Adding partial achievement of goals and triggering relationships between goals to the goal specification.
3. Defining events, such as triggering the top-level goal when the system is initialized.
4. Specifying instance-specific attributes for each goal (for example when several "search area" goals are needed, each searching a specific area which must be provided and maintained by the goal instance.)
5. Being able to instantiate instances of goals in response to specific events and simulate performance of the dynamic system.
6. Designing the agent classes and sub-organizations that will populate the organization including agents, actors, organizations, roles, capabilities, protocols, and services and relationships such as inheritance, possesses, plays, requires, and provides.
7. Designing the role model, representing the roles in the organization, the goals they are capable of achieving and to what degree, as well as the available interaction protocols.
8. Designing the organizational model depicting how the interaction between the organization and external actors.
9. Designing other specifications such as the initial depictions of protocols, plans, actions, and others, but which would go far beyond the scope of this effort.

2 Lexical Structure

2.1 Keywords

Keywords include: 'or','and','at','time','delete', 'trigger', 'triggers', 'achieves','to','requires','possesses','has','assigned'

2.2 Operators and Punctuators

Operators and punctuators include: [';', '=', '(', ')', '+', '&','|', ',', '>','{','}']

2.3 Literals

Literals may be any combination of alpha-numeric characters, no special characters may be used including underscores or dashes.

3 Sample Applications

From the problem domain, there are several scenarios that lend themselves to testing the applicability and utility of the little language.

Scenario 1 – Construction of Hierarchical Goal Tree Specification

This scenario occurs early in the process. The software engineer gets an initial set of requirements for a complex, adaptive system and begins using these requirements to design a goal-based organization. The software engineer wants to create a single top-level goal and then progressively break it into conjunctive and disjunctive goal trees. Conjunctive goal trees require all child goals to be met, and disjunctive goal trees require any one of the child goals to be met. This could require the following language constructions:

- g1 = g2 and g3 [and g4…], where g1, g2, g3, and g4 are abbreviations for identifiers that would actually provide a more descriptive name for the goal.
- g2 = g4 or g5 [or g6…]

Implementation of this scenario would use interpretation of the little language for the following:

- Obtaining the names of each goal.
- Obtaining the hierarchical relationships among the goals.
- Determination of the top-level goal and automatic numbering of all goals reflecting their level in their hierarchy and their arrangement within the set of sibling goals.
- Validation of inputs to indicate logical inconsistencies (optional – to manage scope, may assume only correct trees are entered).

Scenario 2 – Addition of Triggering Relationships and Parameters to the Goal Specification

The engineer may need to indicate when the successful completion of a terminal or leaf goal will trigger the activation of another goal, which would require a language construction similar to the following:

- g4 triggers g5 [, g6, g7…]

Scenario 3 – Instantiation of the Goal tree

The little language could be used to generate test instances of the goal specification in order to test the dynamic behavior of the specification. All leaf goals would be given attributes indicating specific goal conditions.

The static specification defines the structure and policies for the dynamic organization. Nothing can be simulated or instantiated until events are implemented in the little language. Events allow changes in the environment to provoke system behavior. The most important event is the initial activation of the top-level goal. This could be represented in the little language as follows:

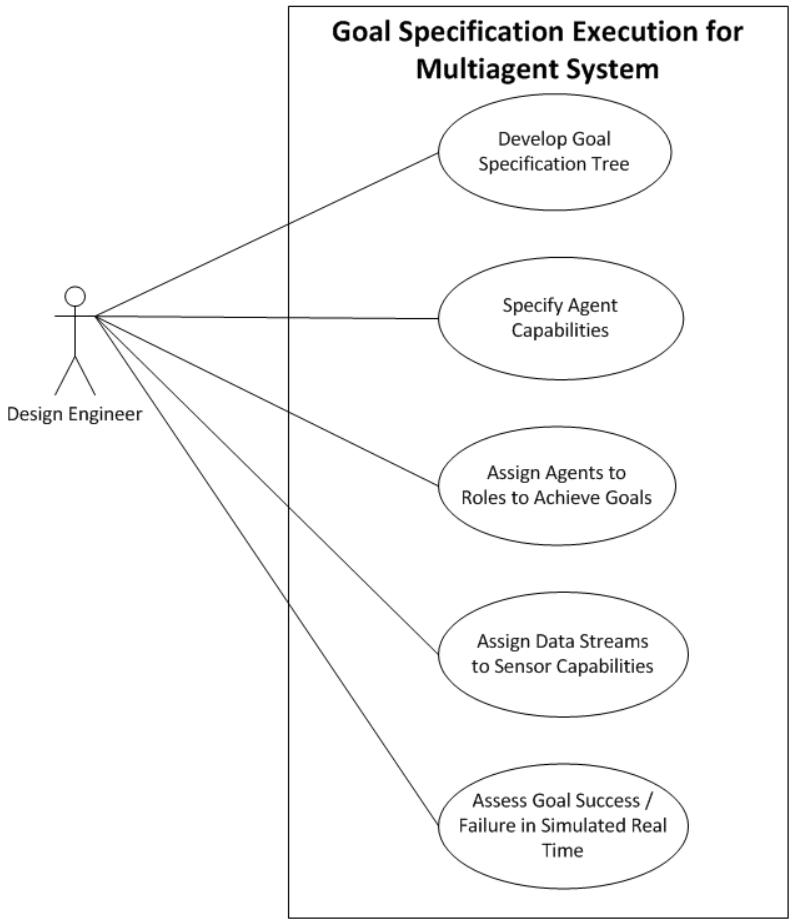
- trigger g1 [at [time] 1] where if no time is provided, the event is assumed to be triggered at the first possible time. This in effect acts as a "run" command to kick off the simulation.

Scenario 4 – Design of Additional Organizational Entities

Because of the power of the little language, it would be possible to use a similar approach when beginning the design of many of the other organizational entities. These specifications could be represented in the little language by statements such as the following:

- role1 achieves goal4 to 88% where the percent sign is optional – but may be helpful for easy reading.
- role1 requires capability1 [, capability 2, capability 3…]
- agentClass1 possesses capability1 [at 100%, capability 2 at 75%, …]

The examples below describe sample applications for this domain specific language.



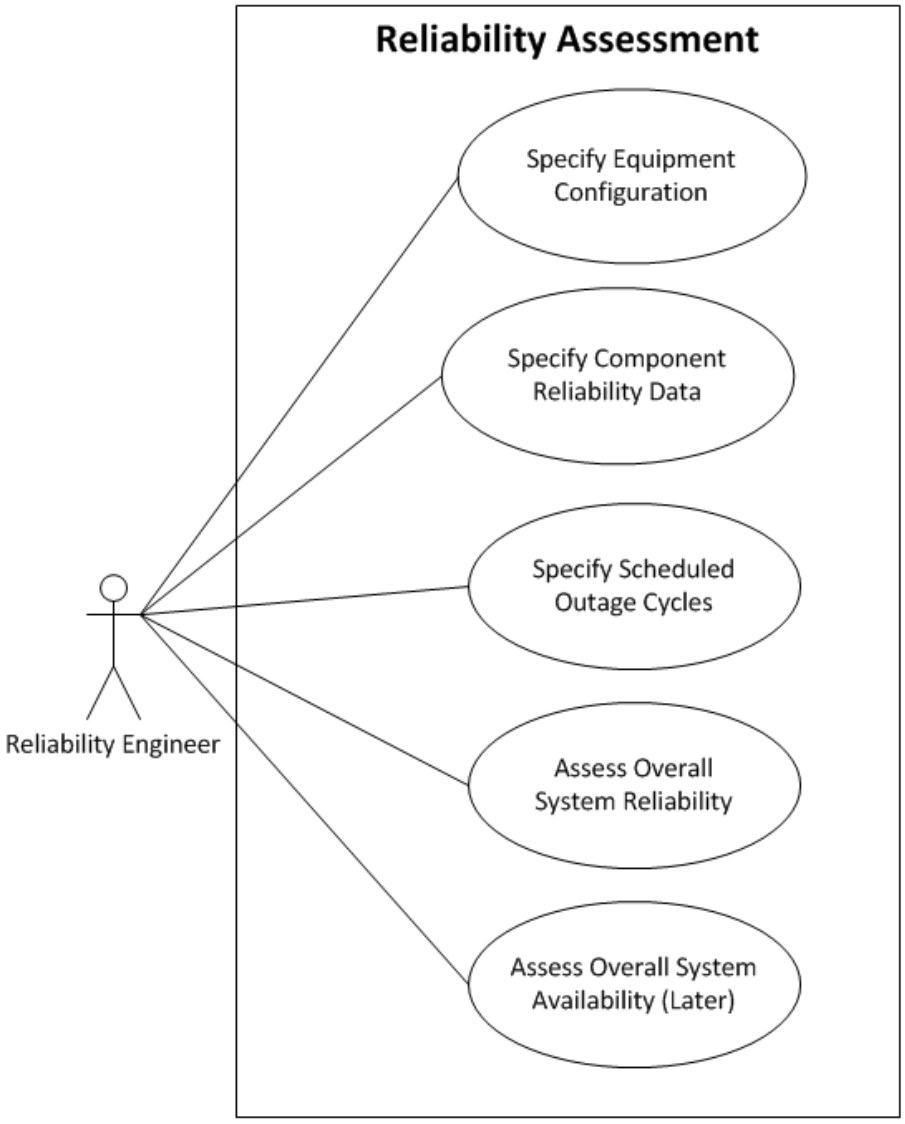
3.1 Use Case 1: Goal Specification Execution for Multiagent System

Actors: Design Engineer, Application

Description: A design engineer wants to test the impacts of a goal specification tree for a simulated real-time system without the overhead of a massive interactive development environment like Eclipse.

Preconditions: The design engineer has access to a computer running python and can write programs in this domain specific language.

Postconditions: System simulates the real-time execution of a goal specification tree and provides information regarding the achievement of (or failure to meet) liveness goals given sample sensor data streams.



3.2 Use Case 2: Reliability Assessment

Actors: Reliability Engineer, Application

Description: A reliability engineer wants to determine whether investing in additional scheduled maintenance will payoff in given the additional system reliability.

Preconditions: The reliability engineer has access to a computer running python and can write programs in this domain specific language.

Postconditions: System can calculate the expected system reliability given different levels of scheduled maintenance cycles and resulting equipment reliabilities. (This information would then feed into additional availability and economic calculations in order to determine which strategy is most effective.

Sample Programs

/ RELIABILITY TRIAL 1 –
/ LOW MAINTENANCE
 	
/ Tests using the language and the
/ tree capabilities to calculate
/ system reliability.
 	
/ SAME CONFIGURATION
 	
plant = unit1 and unit2;
unit1 = boiler1 & turbine1 & bop1;
boiler1 = boiler1S or boiler1F;
boiler1F = bfp1 & otherboiler1;
bfp1 = bfp11 or bfp12;
turbine1 = turbine1S or turbine1F;
unit2 = boiler2 & turbine2 & bop2;
boiler2 = boiler2S or boiler2F;
boiler2F = bfp2 & otherboiler2;
bfp2 = bfp21 or bfp22;
turbine2 = turbine2S or turbine2F;
 	
/ Data set Low Maintenance
 	
boiler1S (1, 1, 580, 1);
boiler2S (1, 150, 580, 1);
 	
bfp11 (59, 980);
bfp12 (59, 980);
bfp21 (59, 980);
bfp22 (59, 980);
 	
otherboiler1 (24, 1000);
otherboiler2 (28, 1000);
 	
turbine1S (1, 1,580,1);
turbine1F (39, 1548);
 	
turbine2S (1, 150,580,1);
turbine2F (27, 1700);
 	
bop1 (10, 7200);
bop2 (35, 7200);
 	
run reliability;
	/ RELIABILITY TRIAL 2 –
	/ AVERAGE MAINTENANCE
	
	/ Tests using the language and the
	/ tree capabilities to calculate
	/ system reliability.
	
	/ SAME CONFIGURATION
	
	plant = unit1 and unit2;
	unit1 = boiler1 & turbine1 & bop1;
	boiler1 = boiler1S or boiler1F;
	boiler1F = bfp1 & otherboiler1;
	bfp1 = bfp11 or bfp12;
	turbine1 = turbine1S or turbine1F;
	unit2 = boiler2 & turbine2 & bop2;
	boiler2 = boiler2S or boiler2F;
	boiler2F = bfp2 & otherboiler2;
	bfp2 = bfp21 or bfp22;
	turbine2 = turbine2S or turbine2F;
	
	/ Data set Average Maintenance
	
	boiler1S (1, 1,620,1);
	boiler2S (1, 150,620,1);
	
	bfp11 (59, 1200);
	bfp12 (59, 1200);
	bfp21 (59, 1200);
	bfp22 (59, 1200);
	
	otherboiler1 (20, 1880);
	otherboiler2 (24, 1880);
	
	turbine1S (1, 1,620,1);
	turbine1F (34, 1548);
	
	turbine2S (1, 150,620,1);
	turbine2F (22, 1700);
	
	bop1 (10, 8000);
	bop2 (35, 8000);
	
	run reliability;
		/ RELIABILITY TRIAL 3 –
		/ TOP DECILE MAINTENANCE
		
		/ Tests using the language and the
		/ tree capabilities to calculate
		/ system reliability.
		
		/ SAME CONFIGURATION
		
		plant = unit1 and unit2;
		unit1 = boiler1 & turbine1 & bop1;
		boiler1 = boiler1S or boiler1F;
		boiler1F = bfp1 & otherboiler1;
		bfp1 = bfp11 or bfp12;
		turbine1 = turbine1S or turbine1F;
		unit2 = boiler2 & turbine2 & bop2;
		boiler2 = boiler2S or boiler2F;
		boiler2F = bfp2 & otherboiler2;
		bfp2 = bfp21 or bfp22;
		turbine2 = turbine2S or turbine2F;
		
		/ Data set Top Decile Maintenance
		
		boiler1S (1, 1,700,1);
		boiler2S (1, 150,700,1);
		
		bfp11 (59, 1400);
		bfp12 (59, 1400);
		bfp21 (59, 1400);
		bfp22 (59, 1400);
		
		otherboiler1 (12, 2000);
		otherboiler2 (18, 2000);
		
		turbine1S (1, 1,700,1);
		turbine1F (34, 1548);
		
		turbine2S (1, 150,700,1);
		turbine2F (22, 1700);
		
		bop1 (10, 8760);
		bop2 (35, 8760);
		
		run reliability;

RELIABILITY = 98.74%	RELIABILITY = 99.01%	RELIABILITY = 99.07%

3.2 Analysis of Results

Initial results indicate that increasing scheduled maintenance to average levels may have a significant reliability impact. Additional availability and economic evaluation is warranted. However, because the additional planned downtime for the increased scheduled maintenance associated with the top decile maintenance strategy actually outweighed the reductions in forced outages, it is unlikely that top decile would be warranted, even allowing for some savings due to being able to schedule the downtime rather than being surprised by it.

Detailed results:

CASE 1. Searching through tree. Top goal is: plant

rel of boiler1S = 0.933789954338 (cycYrs = 1.0 stDay = 1.0 outHrs = 580.0 firstYr = 1.0)
rel of bfp11 = 0.943214629451 (mtbf = 980.0 mttr = 59.0)
rel of bfp12 = 0.943214629451 (mtbf = 980.0 mttr = 59.0)
rel of bfp1 = 0.996775421692
rel of otherboiler1 = 0.9765625 (mtbf = 1000.0 mttr = 24.0)
rel of boiler1F = 0.973413497746
rel of boiler1 = 0.998239706472
rel of turbine1S = 0.933789954338 (cycYrs = 1.0 stDay = 1.0 outHrs = 580.0 firstYr = 1.0)
rel of turbine1F = 0.975425330813 (mtbf = 1548.0 mttr = 39.0)
rel of turbine1 = 0.998372910031
rel of bop1 = 0.998613037448 (mtbf = 7200.0 mttr = 10.0)
rel of unit1 = 0.995233212308
rel of boiler2S = 0.933789954338 (cycYrs = 1.0 stDay = 150.0 outHrs = 580.0 firstYr = 1.0)
rel of bfp21 = 0.943214629451 (mtbf = 980.0 mttr = 59.0)
rel of bfp22 = 0.943214629451 (mtbf = 980.0 mttr = 59.0)
rel of bfp2 = 0.996775421692
rel of otherboiler2 = 0.972762645914 (mtbf = 1000.0 mttr = 28.0)
rel of boiler2F = 0.969625896587
rel of boiler2 = 0.997988929226
rel of turbine2S = 0.933789954338 (cycYrs = 1.0 stDay = 150.0 outHrs = 580.0 firstYr = 1.0)
rel of turbine2F = 0.984365952519 (mtbf = 1700.0 mttr = 27.0)
rel of turbine2 = 0.998964869002
rel of bop2 = 0.995162404976 (mtbf = 7200.0 mttr = 35.0)
rel of unit2 = 0.992133011146
rel of plant = 0.98740372372

==========
Overall Reliability for plant is 98.740372372 %
==========

Case 2. Searching through tree. Top goal is: plant

rel of boiler1S = 0.929223744292 (cycYrs = 1.0 stDay = 1.0 outHrs = 620.0 firstYr = 1.0)
rel of bfp11 = 0.953137410643 (mtbf = 1200.0 mttr = 59.0)
rel of bfp12 = 0.953137410643 (mtbf = 1200.0 mttr = 59.0)
rel of bfp1 = 0.997803897719
rel of otherboiler1 = 0.989473684211 (mtbf = 1880.0 mttr = 20.0)
rel of boiler1F = 0.987300698795
rel of boiler1 = 0.999101191011
rel of turbine1S = 0.929223744292 (cycYrs = 1.0 stDay = 1.0 outHrs = 620.0 firstYr = 1.0)
rel of turbine1F = 0.978508217446 (mtbf = 1548.0 mttr = 34.0)
rel of turbine1 = 0.998478892102

rel of bop1 = 0.998751560549 (mtbf = 8000.0 mttr = 10.0)
rel of unit1 = 0.996336030261
rel of boiler2S = 0.929223744292 (cycYrs = 1.0 stDay = 150.0 outHrs = 620.0 firstYr = 1.0)
rel of bfp21 = 0.953137410643 (mtbf = 1200.0 mttr = 59.0)
rel of bfp22 = 0.953137410643 (mtbf = 1200.0 mttr = 59.0)
rel of bfp2 = 0.997803897719
rel of otherboiler2 = 0.987394957983 (mtbf = 1880.0 mttr = 24.0)
rel of boiler2F = 0.985226537664
rel of boiler2 = 0.998954389652
rel of turbine2S = 0.929223744292 (cycYrs = 1.0 stDay = 150.0 outHrs = 620.0 firstYr = 1.0)
rel of turbine2F = 0.987224157956 (mtbf = 1700.0 mttr = 22.0)
rel of turbine2 = 0.999095773737
rel of bop2 = 0.99564405725 (mtbf = 8000.0 mttr = 35.0)
rel of unit2 = 0.993703655365
rel of plant = 0.990062755242

==========
Overall Reliability for plant is 99.0062755242 %
==========

Case 3. Searching through tree. Top goal is: plant

rel of boiler1S = 0.920091324201 (cycYrs = 1.0 stDay = 1.0 outHrs = 700.0 firstYr = 1.0)
rel of bfp11 = 0.959561343386 (mtbf = 1400.0 mttr = 59.0)
rel of bfp12 = 0.959561343386 (mtbf = 1400.0 mttr = 59.0)
rel of bfp1 = 0.998364715051
rel of otherboiler1 = 0.994035785288 (mtbf = 2000.0 mttr = 12.0)
rel of boiler1F = 0.99241025353
rel of boiler1 = 0.99939351341
rel of turbine1S = 0.920091324201 (cycYrs = 1.0 stDay = 1.0 outHrs = 700.0 firstYr = 1.0)
rel of turbine1F = 0.978508217446 (mtbf = 1548.0 mttr = 34.0)
rel of turbine1 = 0.998282620116
rel of bop1 = 0.998859749145 (mtbf = 8760.0 mttr = 10.0)
rel of unit1 = 0.996539572841
rel of boiler2S = 0.920091324201 (cycYrs = 1.0 stDay = 150.0 outHrs = 700.0 firstYr = 1.0)
rel of bfp21 = 0.959561343386 (mtbf = 1400.0 mttr = 59.0)
rel of bfp22 = 0.959561343386 (mtbf = 1400.0 mttr = 59.0)
rel of bfp2 = 0.998364715051
rel of otherboiler2 = 0.991080277502 (mtbf = 2000.0 mttr = 18.0)
rel of boiler2F = 0.989459578842
rel of boiler2 = 0.999157728903
rel of turbine2S = 0.920091324201 (cycYrs = 1.0 stDay = 150.0 outHrs = 700.0 firstYr = 1.0)
rel of turbine2F = 0.987224157956 (mtbf = 1700.0 mttr = 22.0)
rel of turbine2 = 0.99897909938
rel of bop2 = 0.996020466174 (mtbf = 8760.0 mttr = 35.0)
rel of unit2 = 0.994165565465
rel of plant = 0.990725327942

==========
Overall Reliability for plant is 99.0725327942 %
==========

4 Future Work

Our reliability studies often involve fault tree diagrams spanning 50 pages or more, and the initial test results indicate that this approach could be expanded to larger systems. I would much prefer to use this language than our current system. It would offer significant cost-saving over our current method to extend this into availability and to employ the benefits of automatically combing partial capacity states. Adding in a Monte Carlo simulation would also be fun.