

META II

Complex Systems Design and Analysis (CODA)

**Brian T. Murray, Alessandro Pinto, Randy Skelding, Olivier de Weck, Haifeng Zhu,
Sujit Nair, Narek Shougarian, Kaushik Sinha, Shaunak Bopardikar,
and Larry Zeidner**

**United Technologies Corporation
United Technologies Research Center**

**AUGUST 2011
Final Report**

Approved for public release; distribution unlimited.

Table of Contents

1.0 Summary.....	1
2.0 Introduction.....	3
3.0 Methods, Assumptions and Procedures.....	5
3.1 <i>Summary of Contract-Based Formalism.....</i>	5
3.1.1 General definition of contracts.....	5
3.1.2 A language for contract specification.....	6
3.1.3 BNF for a platform definition.....	8
3.2 <i>Definition of a design flow.....</i>	11
3.3 <i>Definition of an abstraction layer.....</i>	11
3.4 <i>Supporting tools for the design activity.....</i>	13
3.5 <i>Challenge problem definition.....</i>	13
3.5.1 Informal description of the mission requirements.....	14
3.5.2 Vehicle assumptions.....	16
3.5.3 Details of the mission phases.....	16
3.5.4 Baseline system architecture.....	18
3.5.5 Formal definition of the mission requirements.....	19
3.6 <i>Thermal management system.....</i>	20
3.6.1 Schematic representation.....	21
3.6.2 Detailed component models.....	22
3.6.3 Abstract component models.....	22
3.7 <i>Electric power system.....</i>	23
3.7.1 EPS library.....	24
3.7.2 Composition Rules/Requirements.....	24
3.7.3 Sample topology.....	28
3.7.4 Evaluation tools: steady state EPS model & dynamic EPS model.....	31
3.8 <i>Gas Turbine Engine Analysis.....</i>	34
3.8.1 Level 1 - Gas Path Thermodynamics.....	34
3.8.2 Level 0 – an intermediate step in analysis.....	35
3.8.3 Abstraction Levels in the Gas Turbine Engine.....	35
3.8.4 Optimization for the Gas Turbine Engine Analysis.....	36
3.8.5 Components.....	36
3.8.6 Assumptions and Background.....	37
3.8.7 Variables, Parameters and Constraints.....	37
3.8.8 Objective.....	37
3.8.9 Auto-generation from Architecture Enumeration Engine.....	37
3.8.10 Results.....	38
3.8.11 Remarks.....	39

Section	Page
3.8.12 Level 1 Matlab Simulation and Optimization.....	39
3.9 <i>Software and mapping modeling</i>	39
3.9.1 Decision-diagram-based complexity of software models.....	39
3.10 <i>Abstraction layers design</i>	43
3.11 <i>General criteria for the selection of abstraction layers</i>	44
3.12 <i>Definition and computation of abstract models</i>	45
3.13 <i>Application to thermal management</i>	46
3.14 <i>Manufacturing Modeling and Integration</i>	47
3.14.1 Manufacturing Cost Models.....	49
3.14.2 Feature-based cost model.....	50
3.14.3 Model Integration.....	52
3.15 <i>Architecture Enumeration and Analysis</i>	53
3.15.1 The META Design Flow.....	54
3.15.2 Architecture Enumeration and Evaluation (AEE).....	55
3.15.3 AEE algorithm.....	58
3.15.4 Alternate AEE implementations.....	59
3.15.5 Mapping between abstraction layers.....	59
3.15.6 Optimization.....	60
3.16 <i>Complexity and Adaptability Metrics in Design</i>	61
3.16.1 System Complexity.....	61
3.16.2 Structural Complexity and Graph Energy.....	61
3.16.3 Dynamic Complexity.....	64
3.16.4 Shannon Entropy and Dynamic Complexity.....	65
3.16.5 Some Examples:.....	66
3.16.6 Survey of Other Dynamic Complexity Metrics.....	67
3.16.7 Propagation of Uncertainty and Dynamic Complexity.....	69
3.16.8 Quantifying EPA numbers.....	70
3.16.9 Issues with Correlation with First Cost.....	71
3.16.10 Adaptability Metrics.....	72
3.16.11 Design Flow Modeling.....	76
4.0 Conclusions.....	84
5.0 References.....	85
Appendix	
Introduction.....	89
Constructs.....	89
Device-Rule Statement Syntax.....	90
Flow-Rule Statement Syntax.....	94

Section	Page
List of Acronyms, Abbreviations and Symbols.....101

List of Figures

Figure	Page
Figure 1: Increase in System Requirements Leading to Exponential Increases in Required Functionality	3
Figure 2: Overall View of the Elements of an Abstraction Layer Including Library and Tools ..	12
Figure 3: High-Level Diagram Describing the Challenge Problem	14
Figure 4: Detailed Description of the Mission Phases.....	16
Figure 5: Description of the Mission Phases and Transitions Among Them	17
Figure 6: Baseline System Architecture	18
Figure 7: Inputs, States, Outputs and Parameters of System Requirements Specifications	19
Figure 8: Baseline Architecture of a Thermal Management System.....	21
Figure 9: Line Diagram of the Baseline sSstem	22
Figure 10: Sample Topology Used for EPS Evaluation Tool Development	30
Figure 11: EPS Topology Evolution Tools Summary	31
Figure 12: Power Flow Analysis Governing Power Equations	32
Figure 13: State Space Model of an EPS Topology	33
Figure 14: Netlist to State Space Formulation Process.....	34
Figure 15: Level 0 Engine Architecture.....	39
Figure 16: Model 1, a Finite State Machine	40
Figure 17: The Decision Diagram Corresponding to Model 1	40
Figure 18: Model 2, a Dataflow Model with Non-Deterministic Scheduler	41
Figure 19: Decision Diagram Corresponding to Model 2	41
Figure 20: DD Composition of Model 1 and Model 2. The Scheduler Inputs Replaced Outputs of State Machine	41
Figure 21: Model 3, a Functional Model of a Ring Network with Four Functions connected by FIFO channels.....	42
Figure 22: DD Representation of Model 3, Which is Composed of All DDs of the Components	42
Figure 23: Mapped Model 3: FIFO Channels are Mapped Onto a Bus with Adapters	42
Figure 24: Decision Diagram Representation of Mapped Model 3	43
Figure 25: Mapped Model 3: Functions are Mapped Onto Tasks on Two Embedded Processors	43
Figure 26: Decision Diagram Representation of Mapped Model 3	43
Figure 27: General Principles for Abstraction Layer Design	45
Figure 28: Abstraction of a Pump Component	47
Figure 29: Summary of Manufacturing Information Usage	48
Figure 30: Parametric Models.....	49
Figure 31: Matlab/Simulink Gear Model.....	51
Figure 32: IBR Modeling.....	52
Figure 33: META Design Flow	54
Figure 34: Device Design Rule Based on Device Parameter	56
Figure 35: AEE Device Design Rules (excerpt).....	57
Figure 36: Visualization of AEE Algorithm Trace Through Design Space Binary Tree.....	58
Figure 37: Graph Energy	63
Figure 38: Structural Dependencies.....	63
Figure 39: Superlinear Relationship Between Development Cost and Structural Complexity	64
Figure 40: Dynamic Complexity as Uncertainty	65
Figure 41: Entropy	66

Figure	Page
Figure 42: Poincare Section for (1) non-chaotic regime vs. (2) chaotic regime for damped, driven pendulum system	67
Figure 43: Computing adaptability metric.....	74
Figure 44: Simplified illustration of EPS wiring	75
Figure 45: Vensim System Dynamics Model of Product Development.....	76
Figure 46: Idealized Sequential Project (Current Practice) – Completion Time 51 Months.....	77
Figure 47: Realistic Project with Changes (Current Practice): Completion Time 87 Months	78
Figure 48: A simple Electric Power System (EPS) example.....	94

List of Tables

Figure	Page
Table 1: Requirements for Each Phase of the Mission	18
Table 2: EPS Library: Four Basic Building Blocks.....	28
Table 3: Sample Topology Load List	29
Table 4: Netlist Data Structure	30
Table 5: AEE Device Table	56
Table 6: AEE Flow Table	56
Table 7: AEE Inputs Table	56
Table 8: Flight mission segments	73

1.0 SUMMARY

Cost and schedule for the design of ground and aerial vehicles for military applications have been steadily increasing at each new generation. Most importantly, cost and schedule overruns have become common to large defense programs aiming at building military systems with enhanced performance. Stringent performance requirements (e.g. maneuverability, survivability, adaptability) are the main drivers of these problems due to innovative materials and architectural solutions that have not been tested in previous products.

The ability to predict cost and schedule relies on a streamlined design flow where most of the re-design cycles can be done in a virtual environment without the use of hardware prototyping. The virtual environment should allow the designer to assemble systems using computer models and to perform analysis and trade studies so that correctness, requirement satisfaction and optimality can be proved and only limited testing is needed at the hardware level.

A virtual environment that provides a flat view of a system would be of little help to designers. Large systems comprise sub-systems belonging to different application domains, each requiring special tools and design flow which may need to also take into account organizational structures. Moreover, the sheer number of components in a system may prevent effective analysis or design space exploration due to the large number of choices involved. For this reason, it is already common place to use different views of a system at different abstraction layers. For example, an airframe provider may not care about the details of how heat is extracted from individual components, but it may instead care about the total weight of a thermal management system and the total volume occupied by its most important components.

The Platform-Based Design (PBD) paradigm is an intellectual framework which formalizes the organization of a design flow across multiple abstraction levels and domains. Requirements flow down from one level to the next and the correct interface specification among sub-systems is of critical importance in this framework and Contract-Based Design (CBD) is a methodology that comes to help. However, deploying these methodologies is non-trivial and requires expertise in several disciplines such as languages, verification and analysis algorithms, optimization and software development. A great deal of domain knowledge is also required and need to be considered as an orthogonal axis so that languages can be useful to model components in each of the application domains; verification and analysis can be done efficiently by leveraging facts that are known in each domain; optimization can quickly explore the design space by ignoring solutions that are known to be sub-optimal; and software can be customized to the need of engineers operating in a particular domain. Moreover, each domain requires a specific decomposition of complex designs into abstraction layers.

The CODA design methods and tools aim at providing guidelines and experimental software for addressing the design challenge of future military systems. CODA follows the principles set forth by the PBD and CBD paradigms and attempts the instantiation of both, unveiling the difficulties, the benefits and the possible approaches for their effective use. The main goal of CODA is the reduction of design and development cycle by 5X. While this number is difficult to validate (as no prototype has been built), the CODA study provides evidence that such speed-up could in principle be obtained through the use of “carefully selected” multiple abstraction levels.

The method followed in this program is the following. A challenge problem has been defined first. The challenge problem, a Unmanned Aerial Vehicle (UAV) tasked with a specific mission, exposes many of the characteristics of large systems such as heterogeneity and large

number of components. The UAV comprises the body of the aircraft, and the propulsion, electrical, and thermal sub-systems. The PBD methodology has then been studied with the intent of providing a clear and practical definition of an abstraction layer and all the activities to be conducted thereof. In CODA, abstraction layers are bridged through the use of design space exploration. This is mainly architectural exploration which is done by first enumerating the architectures that are valid at that abstraction layer, and then ranking them according to different complexity metrics. Several complexity metrics have been defined in CODA so that designers can look at different axes of the trade space and decide where to focus the design effort.

To apply the PBD methodology, component models at different abstraction layers have been developed for each of the sub-systems of the UAV. Components are then grouped into libraries. Composition rules can be associated to libraries to define families of valid architectures. A contract-based specification language has been developed to formally defined components and rules.

2.0 INTRODUCTION

The increased frequency of cost and schedule overruns observed in the execution of defense programs is a sign of systematic mismanagement of the inherent complexity associated with the design of these systems. Although difficult to quantify and often abused, the term “complexity” is directly related to two characteristics of a design: the number of design parameters, and the level of understanding of the interactions among them. There is evidence that these two abstract metrics have increased exponentially over the years. Figure 1 shows some examples of trends in modern defense system performance requirements. Because of the expansion of the performance envelope, the requirement description is typically longer (including more parameters to be taken into account) and the corresponding implementation typically involves the selection of more components. Controlling some of the parameters in the system (e.g., for stability) requires aggressive control strategies for which a formal development process is either not available or immature. Moreover, the interaction among components is typically not well understood. For example, one of the challenges related to the coupling of systems can be found in the thermal management requirements of modern fighter aircraft. Fuel is generally used as a heat sink to cool avionics and other thermally sensitive components. However, due to limitations on the maximum allowable temperature of aviation fuel, a shortage of heat capacity is often experienced, leading to operational challenges during ground operation where fuel and air flow rates are low.

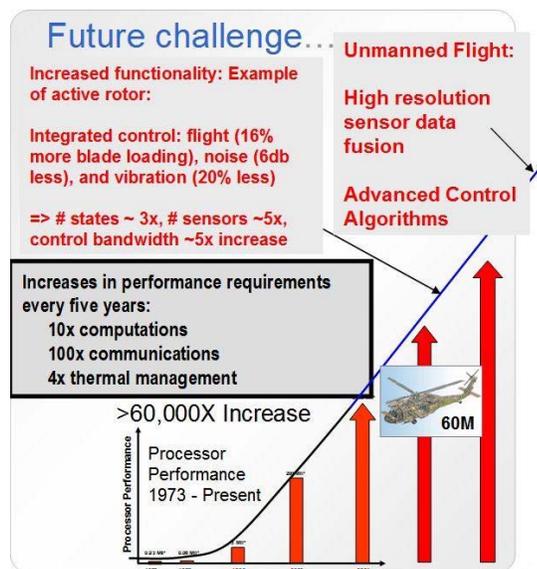


Figure 1: Increase in System Requirements Leading to Exponential Increases in Required Functionality

To provide a solution for the management of complexity in modern cyber-physical systems, the United Technologies Research Center (UTRC) team developed a three-pronged approach. The fundamental element will be the development of a new design paradigm, rooted in platform-based design (PBD), which lays the foundation for a formal representation of cyber-physical systems. This foundation provides the starting point for the implementation of an entirely new approach to the design of defense systems, leading in the long term to the development of

complete correct-by-construction methods that guarantee system performance. The second element is the development of new analytical techniques to better characterize complex systems; particularly, their dynamic and uncertainty characteristics. The methods present both a near-term approach to improving existing design efforts, as well as a long-term benefit as the knowledge base on the nature of complex systems grows and can be applied to future design efforts. The third element is the development of new methods for quantifying and understanding complexity and adaptability in cyber-physical systems. These metrics will enable designers to utilize complexity and adaptability as objective functions particularly within the early stages of the design process, leading to a better understanding of the effect of these characteristics on cost, schedule, and capability.

3.0 METHODS, ASSUMPTIONS AND PROCEDURES

We follow the Platform-Based Design paradigm and we also adopt the Contract Based Design methodology. We will first revise some concepts such as design flows and abstraction layers and we will then develop tools and methods to demonstrate our approach. We first define a challenge problem which will drive the development of models and tools; then, we develop models for all the sub-systems of a small-size UAV. We then preset design space exploration methods as well as metrics to rank architectural solutions.

3.1 Summary of Contract-Based Formalism

We use the notion of contracts associated to models for system design and integration. A contract provides the ability to formally define the interfaces of a model. In particular, a contract specifies the assumptions on the environment of a component (i.e. the rest of the system) under which the component is able to perform. The behavior and performance of the component are defined by the guarantees. In this section, we first define a simple notion of contracts that we will use in the rest of the report. The general definition captures both steady state and dynamic behavior. We then restrict our work to steady state models and we develop a language to specify contracts. We call the language tinyCSL and will be used as the main vehicle for formally defining component libraries and composition rules. A platform is a library (i.e. a set of component contracts) and a set of rules that must be satisfied by any composition of library elements. A platform defines a family of possible architectures.

3.1.1 General definition of contracts

We begin with a definition of contracts by dividing variables into different kinds, which extends on the similar notion from [4]. A contract C is a tuple (V, A, G) in which $V(U, X)$ is the set of variables partitioned into uncontrollable variables U taking on values in D_U and controllable variables X taking on values in D_X (thus V takes on values in $D_U \times D_X$), $A \subseteq D_U$ is a set of assumptions on the uncontrollable variables and $G \subseteq D_U \times D_X$ is a set of guarantees or relations that hold between the uncontrollable variables U and the controllable variables X .

Given two contracts, we first need to check whether they are compatible or not.

Two contracts $C_1((U_1, X_1), A_1, G_1)$ and $C_2((U_2, X_2), A_2, G_2)$ are *compatible* if: $X_1 \cap X_2 = \emptyset$, and

$$G_1 \uparrow^{U_1 \cup U_2, X_1 \cup X_2} \subseteq A_2 \uparrow^{U_1 \cup U_2, X_1 \cup X_2}, \quad \text{and} \quad G_2 \uparrow^{U_1 \cup U_2, X_1 \cup X_2} \subseteq A_1 \uparrow^{U_1 \cup U_2, X_1 \cup X_2}$$

A simple test for compatibility is to verify that the two sets $G_1 \cap A_2^c$ and $G_2 \cap A_1^c$ are both empty. This condition is easy to check if the assumptions and the guarantees are algebraic functions of the variables.

Once compatibility is checked, we can define the parallel composition of two contracts $C_1((U_1, X_1), A_1, G_1)$ and $C_2((U_2, X_2), A_2, G_2)$, which is another contract $C((U, X), A, G)$ that can be derived via the following rules:

1. Pre-condition: $X_1 \cap X_2 = \emptyset$,
2. $X := X_1 \cup X_2$,
3. $U := (U_1 \cup U_2) \setminus X$,

4. $G := G_1 \uparrow^{U \cup X} \cap G_2 \uparrow^{U \cup X}$, and
5. $A := \left((A_1 \uparrow^{\{U_1 \cup U_2\}} \cap A_2 \uparrow^{\{U_1 \cup U_2\}}) \cup G^c \right) \downarrow_U$.

Using the concept of parallel composition along with compatibility check from Definition 2, the concept of composition can be extended to multiple contracts.

3.1.2 A language for contract specification

The objective of the language is to provide an interchange model that is able to capture the specification and the library of component at each abstraction layer. In this language we do not provide any notion of time. We restrict the language to static properties, although we will provide some guidelines and the necessary infrastructure to model modes of operation. The language targets static and quasi static properties and it is used in preliminary design. We first define the language informally using a pseudo-BNF and plain English language to define its meaning. We start with the definition of a platform (keywords are written in bold font):

```

platform <ID> {
  library <ID> {
    component <ID> {
      var <ID> : <TYPE>
      ...
      par <ID>: <TYPE> value <VAL>
      ...
      view <ID> {
        assumption {
          constraint : <CONSTR>
          ...
        }
        constraint: v = model(<PATH>, <ARGS>)
        }
        guarantee {
          constraint : <CONSTR>
          ...
        }
        abstracts : <PATH>
      }
      ...
    }
    ...
  }
  rules {
    <CONSTR>
    ...
  }
}

```

A platform contains a library and a set of rules. The library contains several components. Each component is characterized by a set of variables with a given type. In addition to variables, a component has parameters that have a type and a value associated with them. A component contains a set of views, each characterized by assumptions and guarantees. These are lists of constraints on the variables of the components. We also allow a variable to be defined by an external model for which we provide a path and the arguments that need to be passed. The model should be a function that returns a value for the variable. A view is an abstraction of a detailed

model that can be also provided as part of the description of a view. The set of rules define the valid instances of components from the library. The rules can refer to an architecture instance to its components and can use quantifiers such as "for all components of type comp1 in the architecture, the sum of their weights should not be greater than x". Moreover, an architecture instance comes with a connection contract that simply defines what are the variables that are equal to each other. For example, consider an architecture with two components comp1 and comp2. Also, let comp1 have two variables comp1.x and comp1.y. Let comp2 also have two variables comp2.x and comp2.y. Then, a connection contract can be expressed as a guarantee such that

```
comp1.x=comp2.x
comp1.y=comp2.y
```

The rules should define what the allowed connection contracts are. For example, "for ACGen and DCGen, the voltage variables of ACGen cannot be connected to the voltage variable of DCGen".

In BNF form, the platform definition language is the following :

```
(1) <platform> ::= platform <identifier> '{' <library> <rules> '}'
(2) <library> ::= library <identifier> '{' <component>+ '}'
(3) <rules> ::= rules '{' <rule_assumption> <rule_guarantee> '}'
(4) <component> ::= component <identifier> '{' <line>* <view>* '}'
(5) <line> ::= { <variable> | <parameter> } ';'
(6) <variable> ::= <input_variable> | <output_variable> | <internal_variable>
(7) <input_variable> ::= var input <identifier> ':' <type> [ : <unit> ]
(8) <output_variable> ::= var output <identifier> ':' <type> [ : <unit> ]
(9) <internal_variable> ::= var internal <identifier> ':' <type> [ : <unit> ]
(10) <type> ::= boolean | integer | real | complex | <matrix>
| <enum> | <range> | component
(11) <matrix> ::= matrix '(' <numerical_value> ',' <numerical_value> ')'
(12) <enum> ::= enum '{' <enum_element>* '}'
(13) <range> ::= range '[' <numerical_value> ',' <numerical_value> ']'
(14) <unit> ::= <identifier>
(15) <enum_element> ::= <identifier>
(16) <parameter> ::= par <identifier> ':' <type> [ : <unit> ] : value
<numerical_expression>
(17) <view> ::= view <identifier> '{' <assumption> <guarantee> [ <abstraction>
] '}'
(18) <assumption> ::= assumption '{' <constraint>* '}'
<rule_assumption> ::= assumption '{' <rule_CONSTR>* '}'
(19) <constraint> ::= <CONSTR> ';'
(20) <guarantee> ::= guarantee '{' <constraint>* '}'
<rule_guarantee> ::= guarantee '{' <rule_CONSTR>* '}'
(21) <rules> ::= rules '{' <rule_constraint>* '}'
(22) <rule_constraint> ::= <identifier> ':' <rule_CONSTR> ';'
(23) <abstraction> ::= abstracts ':' <PATH>
(24) <PATH> ::= <platform_name> '/' <library_name> '/' 'component_name'
[ '/' <view_name> ]
(25) <platform_name> ::= <identifier>
(26) <library_name> ::= <identifier>
(27) <component_name> ::= <identifier>
(28) <view_name> ::= <identifier>
(29) <identifier> ::= <letter>{<letter>|<digit>|'_' }*
(30) <value> ::= <numerical_value> | <boolean_value>
(31) <numerical_value> ::= <digit>+ [ '.' <digit>+ ] [ e<digit>+ ]
(32) <numerical_expression> ::= <numerical_value> { <numerical_operator> <numerical_value> } *
(33) <numerical_operator> ::= '+' | '-' | '*' | '/'
(34) <boolean_value> ::= true | false
(35) <CONSTR> ::= <variable_assignment> | <boolean_assertion>
<rule_CONSTR> ::= <ins_quantified_constraint>
(36) <variable_assignment> ::= <identifier> '=' <identifier> <operator> <identifier>
| <identifier> '=' <identifier> <operator> <numerical_expression>
| <identifier> '=' <identifier> <operator> <boolean_value>
| <identifier> '=' <value>
```

```

(37) <boolean_entry> ::= <identifier> | <identifier> '=' <value> | '(' <identifier> '='
<value> ')'
(38) <boolean_literal> ::= [NOT] <boolean_entry>
(39) <boolean_expression> ::= <boolean_literal> { {AND | OR} <boolean_literal> }*
(40) <boolean_assertion> ::= if <boolean_expression> then <boolean_expression>
[ else <boolean_expression> ]
| [NOT] <boolean_expression> { AND | OR } [NOT] <boolean_expression>
(41) <operator> ::= '<' | '>' | '<=' | '>='

```

3.1.3 BNF for a platform definition

In general <CONSTR> stands for an AMPL type constraint expression. For now the BNF syntax of CONSTR is limited.

The BNF for a platform instantiation is the following

```

(42) <platform_istantiation> ::= specification <identifier> ':' <platform_name> '{'
{<component_instantiation> ';' } *
{<variable> ';' } *
<assumption> <guarantee> '}'
(xx) <assumption> ::= assumption '{' <ins_constraint>* '}'
(xx) <ins_constraint> ::= constraint ':' <ins_CONSTR> ';'
(xx) <guarantee> ::= guarantee '{' <ins_constraint>* '}'
(43) <component_instantiation> ::= <instance_name> instanceOf <component_name>
'{' <instance_var> '=' <value>* ';' '}'
| <instance_name> instanceOf COMPONENT
'{' <instance_var> '=' <value>* ';' '}'
(44) <instance_name> ::= <identifier>
(45) <component_name> ::= <PATH>
(46) <instance_var> ::= <identifier> '.' <identifier>
(47) <ins_CONSTR> ::= <ins_variable_assignment> | <ins_boolean_assertion>
| <connection>
| <ins_quantified_constraint>
(48) <ins_variable_assignment> ::= <instance_var> '=' <instance_var> <operator> <instance_var>
| <instance_var> '=' <instance_var> <operator>
<numerical_expression>
| <instance_var> '=' <instance_var> <operator> <boolean_value>
| <instance_var> '='
{ <numerical_value> | <boolean_value> | <identifier> }
(49) <ins_boolean_entry> ::= <instance_var>
| <instance_var> '=' {<numerical_expression> | <boolean_value> }
(50) <ins_boolean_literal> ::= [NOT] <ins_boolean_entry>
(51) <ins_boolean_expression> ::= <ins_boolean_literal> { {AND | OR} <ins_boolean_literal> } *
(52) <ins_boolean_assertion> ::= if <ins_boolean_expression> then <ins_boolean_expression>
[ else <ins_boolean_expression> ] ';'
| [NOT] <ins_boolean_expression> { AND | OR }
[NOT] <ins_boolean_expression> ';'
(53) <connection> ::= <instance_var> '=' <instance_var>
| <identifier> '=' <instance_var>
(54) <ins_quantified_constraint> ::= <quantifier>
'(' <component_instantiation> { ',' <component_instantiation> } * ')'
'{' {<ins_variable_assignment> | <ins_boolean_assertion> }
<ins_quantified_constraint>+ '}'
:= for '(' <component_instantiation> [ ':' <filter> ]
{ ',' <component_instantiation> [ <filter> ] } * ')'
<set_operator> '(' <instance_var> ')'
{'=' | <operator>} {<numerical_value> | <boolean_value>
}

```

```

(55) <filter> ::= <ins_boolean_expression>
(56) <quantifier> ::= forall | exists
(57) <set_operator> ::= Sum | Product | Max | Min | Avg

```

Comments start with '%' and extend to the end of the line, i.e. until a new line character is found.

The semantics of the language define what are the valid architectures that may be instantiated from a given platform, i.e. what is the meaning of the rules specified in the platform definition and platform instantiation. Constraints, both in the platform definition and instantiation evaluate to either **true** or **false**.

A numerical expression evaluates to a numerical value with the usual meaning of operators +, -, *, /. A <variable_assignment> (line 36 in the platform definition BNF) creates identifiers for numerical expressions. The right hand side of a <variable_assignment> is a numerical expression that will evaluate to a numerical value once all variables are assigned to a particular value. All variables do not have a value when a platform is defined. At the time the identifier on the left hand side of a <variable_assignment> is just a placeholder for the symbolic numerical expression on the right hand side which may have variables.

During a platform instantiation, variables are bound to other variables in other component instance by means of connections. A <boolean_expression> (line 39 in the platform definition BNF) evaluates to a **true** or **false** value depending on the values of boolean literals that occur in it and the standard semantics of AND and OR. A <boolean_entry> is either a boolean identifier or a comparison of a variable with a value (line 37 in the platform definition BNF). A <boolean_literal> is a boolean entry or its negation and also evaluates to a boolean **true** or **false** value depending on the value of the boolean entry and the standard semantics of NOT.

A <boolean_assertion> also evaluates to **true** or **false** depending on the values of <boolean_expression>s in it. The expression if X then Y else Z evaluates to **true** if X and Y are both true or X is false and Z is true. Here X, Y, Z are boolean expressions. All <boolean_assertion>s must evaluate to **true** in a valid architecture definition. In other words, X being true or false forces one of Y or Z to be true or false. Both ways of stating a <boolean_assertion> semantically capture the same information.

A platform instantiation, i.e. an architecture definition, is a set of component instantiations with connections between component instances. A connection (line 53 in the platform instantiation BNF) is a statement that an output variable of one component instance feeds to an input variable of another component instance or it can be specified by means of a `connection` object. One output can go to multiple inputs.

The semantics of <ins_boolean_expression>, <ins_boolean_assertion> and <ins_variable_assignment> is the same as the corresponding constructs in the platform definition. The only difference is that instead of dealing with variables these deal with variables in component instances, i.e. <instance_var> (line 46 in the platform instantiation BNF).

The rules in a platform definition intuitively encode what the criteria are for a platform instantiation to be valid so they must refer to component instances. Since component instances are not known at platform definition time, rules must be quantified to talk about possible instances of components and how they relate to one another. Thus rules are specified using

<ins_quantified_constraint> which makes an assertion over all instances of a component (using **forall**) or makes an existential assertion over some instance of a component (using **exists**). An <ins_quantified_constraint> can also state set constraints, such as the sum of some attribute of all instances of a component (line 54 in the platform instantiation BNF). A quantified constraint also evaluates to **true** or **false**.

Quantified constraints can be recursive, i.e., a forall constraint can include another forall or exists constraint inside it. The quantifications are over component instances. Since the universe of component instances in any platform instantiation is finite, the quantification can be removed by enumerating component instances. Thus rules specified as part of a platform definition can be translated into quantifier free constraints to create a contract of rules that a valid architecture must satisfy.

The rules in a platform definition can be transformed into a contract, $C_R=(A_R,G_R)$. A_R is the set of assumptions and G_R is the set of guarantees corresponding to the rules. A_R and G_R are obtained from rules by eliminating the quantifiers as described above. This is illustrated in the following example.

Consider the following quantified rule which states that the power drawn from a generator is within the generator capacity. p_in is the variable corresponding to the power drawn by a load.

```
power_balance_constraint :
forall (g instanceof generator) {
  for( c instanceof connection : c.in = g ){
    Sum(out.p_in) <= g.p_out ;
  }
}
```

Here connection is also a component defined as follows

```
component connection {
  var in : component;
  var out : component;
  var availability : real;
  view logic {
    assumption {
    }
    guarantee {
      constraint : 1 - 10e-6 < availability
                  < 1 - 10e-5;
    }
  }
}
```

Say, in a given platform instance, we have two generators - G_1, G_2 and three loads l_1, l_2, l_3 . Say G_1 is connected to l_1 and G_2 is connected to l_2 and l_3 . Then the above rule may be rewritten as the following constraint:

$$l_1.p_{in} \leq G_1.p_{out} \text{ AND}$$

Once the set of rules R in a platform have been transformed into a set of clauses corresponding to a particular instance, the rules define a contract $C_R=(A_R,G_R)$. A_R derives from the assumption section of the rules and G_R derives from the guarantees section of the rules in the platform definition.

Before the conditions on a platform instance or system architecture can be enunciated, recall that assumptions and guarantees in the rules in a platform definition are formulas quantified over component instances. Variables that occur in these formulas refer to variables inside components, e.g., $c.i$ refers to the variable i of component c .

A variable is said to be an input variable of a platform instance or system architecture iff

- the variable is an input variable of a component instance in the platform instantiation, and
- the variable is bound to an input variable of the platform instance by means of a connection. Recall that a platform instance defines a set of input, output and parameter variables which are independent of component instances.

Denote the set of input variables of a platform instance, S , by $Inputs(S)$.

A behavior of a component assigns a history of values to the ports of the component. An implementation M is a set of behaviors of a component. M implements the component correctly if it satisfies the component contract $C=(A,G)$. M satisfies C , written as $M\models C$ iff $M\cap A\subseteq G$.

The definition of refinement of contracts is revisited here from [4]. A contract $C'=(A',G')$ refines $C=(A,G)$, written as $C'\leq C$ iff the set of models that satisfy C' is a subset of models that satisfy C . In other words, $C'\leq C$ iff $A'\supseteq A$ and $G'\subseteq G$. A system architecture or platform instance, S , is said to be valid iff

- C , the composition of the contracts of all component instances in S is consistent.
- $C\leq C_R$ where C is the composition of the contracts of all component instances in A , and $C_R=(A_R,G_R)$ is the contract corresponding to rules in the platform of which S is an instance.
- Variables in A_R are input variables of the platform instance, i.e., $Var(A_R)\subseteq Inputs(S)$.

3.2 Definition of a design flow

A design flow is built as a sequence of design activity conducted at different abstraction layers. The design flow proceeds from the specification of requirements down to a level where a system can be considered ready for manufacturing. The layers are related to-down by flowing requirements downwards, and bottom-up by abstracting performance and behavior upwards.

3.3 Definition of an abstraction layer

Our design flow relies on the successive refinement of a high level specification into a concrete artifact. Each refinement step is named abstraction layer. In this section we define an abstraction layer and the design activities that are typically used to refine a system specification. As a result of the design activity, a refinement design is obtained which defines a new specification for the level below. In defining an abstraction level, we will explicitly refer to the notion of contract.

The elements of an abstraction layer are shown in Figure 2. The pivotal point is the *platform* which is defined by a set of library elements (or components) and a set of composition rules. The components represent both agents and communication among them. For example, an electric power system library contains agents such as generators, power converters and inverters, transformer rectifier units, and communication media such as busses and contactors. The distinction among these is not fundamental. The fundamental distinction is between the models representing the components (e.g. their dynamic behavior) and the model representing the way in

which they interact. This distinction is a semantic one. In addition to the components, additional rules may be specified in the platform. These rules apply to any architecture that is generated by putting together library elements. For example, one may impose a rule on an engine platform requiring any engine to have a fan. Composition rules restrict the set of possible architecture that can be generated using the library elements.

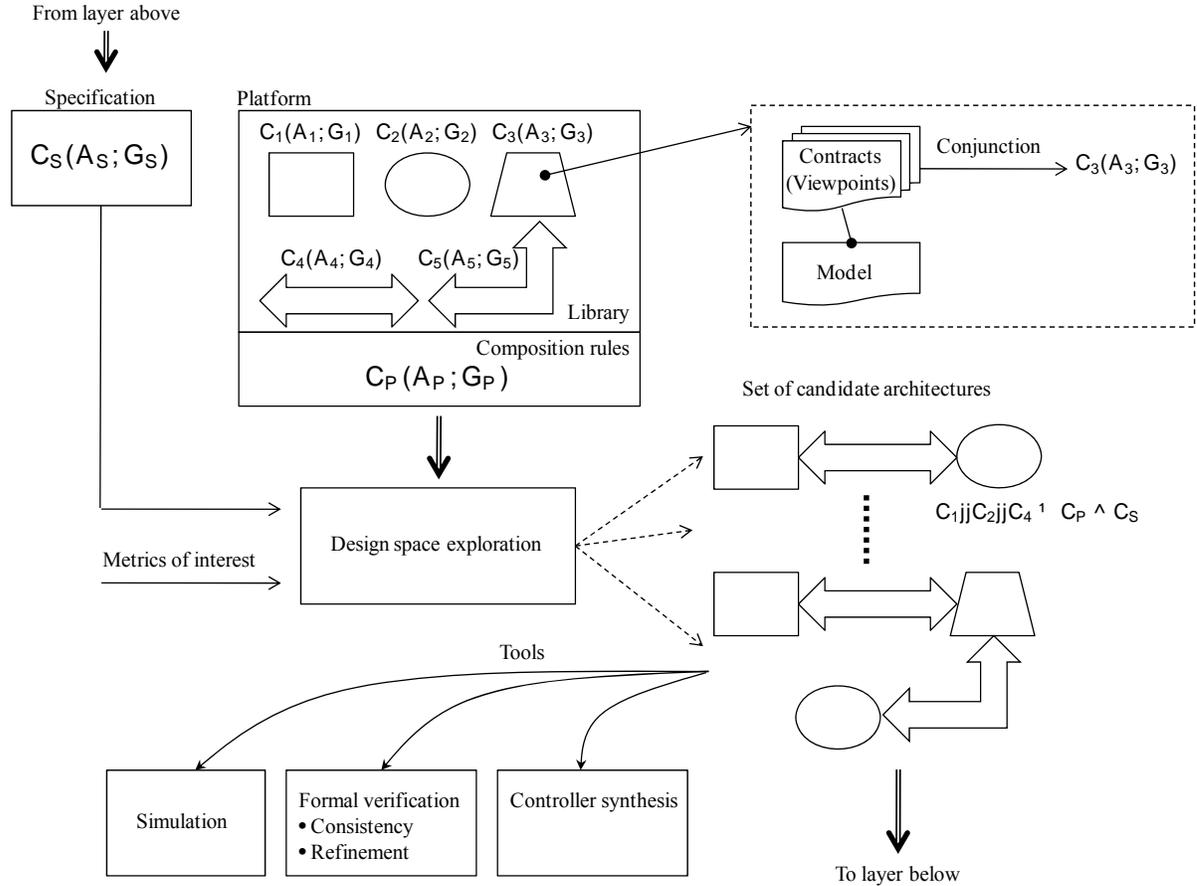


Figure 2: Overall View of the Elements of an Abstraction Layer Including Library and Tools

The way in which libraries are defined today is not a standardized process. The languages used to capture components and their properties vary depending on the system to be designed. Moreover, much effort is spent in developing high fidelity models with little attention to their abstractions. In our view of the library, each component can have a model associated with it which can also be a detailed simulation model. However, several abstract view of the model should be provided as contracts. For example, a detailed simulation model of a power generator could be associated with a performance view, namely rated power and overloading conditions; an efficiency view, namely a relation between the rated power, the loading condition and the efficiency number; and a weight and space view, namely a relation between the rated power and the weight and geometry of the generator. These views can be captured by contracts, and contracts can be joined to yield the overall contract associated with the generator.

3.4 Supporting tools for the design activity

Given a platform, design space exploration can be used to select a set of architectures with different trade-offs among a set of metrics provided by the system engineers. The goal of the design exploration activity is to select non-dominated (with respect to the metrics such as cost, weight, performance) compositions of library elements that satisfy composition rules and at the same time fulfill a set of requirements coming from the previous abstraction layer. Notice that the previous abstraction layer might just be the input to a design flow rather than the result of a design step. Such specification should also be represented by a contract that needs to be refined by any selected architecture. In the example of Figure 2, the first architecture is the composition of three components and its contract can be computed as composition of contracts $C_1 \parallel C_2 \parallel C_3$. The composition must be a refinement of both the specification and of the rules of the platform which means that it satisfies the specification and at the same time is a valid composition of library elements.

Typically, design space exploration yields architectures that are correct by construction with respect to those metrics and contracts that are taken into account during optimization. Because of the complexity of components models, often time some of the constraints and quantities associated with components are either abstraction or not considered. Thus, each candidate architecture needs to go through validation which takes into account the details left behind during optimization. Validation can be done in several ways. One of the most common tools used for validation is simulation. In this context, generating a simulation for an architecture means bringing together heterogeneous models which is one of the challenges to be addressed by the infrastructure supporting the meta-language. Besides simulation, formal methods can also be applied to check properties of the system. The types of verification problems to be solved in this context are compatibility checking and refinement checking among contracts.

The need for controller synthesis stems from the fact that the current design space exploration tools do not take into account the full system dynamics. A controller might be among the components of a platform but its behavioral view might not be used during design optimization. It is possible to include some performance bound due to limitations in the complexity of the controller, but the detailed control parameters need to be fixed as a further refinement. The type of controllers to be synthesized are both discrete (for fault management) and continuous controllers.

The result of the design activities at one abstraction layer is a set of architecture instances with their associated contracts. These contracts form the specification to be handed over to the next abstraction layer.

3.5 Challenge problem definition

The design methodology and tools to be developed under the DARPA META 2 contract will provide significant reduction of design cost and schedule for future aircraft over traditional methods. This novel design flow will be demonstrated on a challenge problem that encompasses the most critical sub-systems that constitute an aircraft: the propulsion system (i.e. the engine), the electric power system, the thermal management system, and the control and communication platform that are important elements for more-electric aircraft. The high level diagram of the prototypical aircraft is shown in Figure 3.

We start from an abstraction level where the system can be decomposed into 4 sub-systems. The propulsion system comprises the engine as its main component. The engine exposes several

interfaces including the control interface, the fuel interface, the mechanical shaft interface, and the heat interface – that can be refined into an oil interface to be connected to the thermal management system. We consider generators as part of the electric power sub-system. Generators are connected to the engine shaft and provide electric power to the aircraft. Storage elements might be present on board of the aircraft. The thermal management system uses several heat sinks to reject the heat produced by the engine, by the electric power system and by the environmental control system. The control sub-system includes also the embedded control platform that supports the execution and interaction among control algorithms distributed on the aircraft.

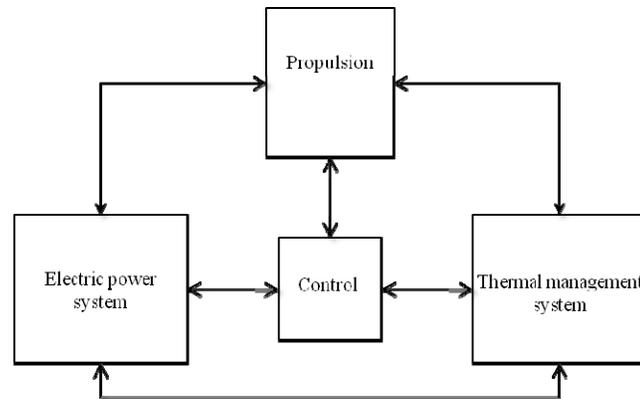


Figure 3: High-Level Diagram Describing the Challenge Problem

The objectives of this study are the followings:

- Determine what are the requirements driving the complexity of these sub-systems
- Define a set of requirements at the system level that will be used as reference for a design flow which will demonstrate the advantages of the proposed methodology. These requirements will be such that the system is expected to be complex, meaning that the solution is not obvious to expert designers.
- For what possible, partition the requirements over the 4 subsystems. For example, the weight of the aircraft together with the type of missions to be executed will provide inputs to determine the maximum thrust that the engine(s) should provide.

The derived requirements will be used as the input for the sub-sequent refinement steps leading to an optimal design.

3.5.1 Informal description of the mission requirements

A mission defines the abstracted expected dynamic behavior of an aircraft. It can be captured in terms of mission phases (e.g. take-off, ascend, fast descend etc.) and transitions among them. Notice that in general not all transitions are necessarily allowed. Each phase is characterized by a set of requirements for the aircraft. Some of them are imposed by the mission itself. For example, the aircraft might be required to execute short take-offs (within a given maximum distance), or it might be constrained to climb at a certain rate so that a target altitude can be reached in a relatively short time. Each phase can also be characterized by quantities such as flight time, availability and safety requirements, or more generally with probabilities that a mission phase can be safely accomplished.

The mission that we selected for this challenge problem comprises two major segments: an Intelligence, Surveillance and Reconnaissance (ISR) segment, and a high speed strike segment. These two segments impose very different requirements on the components of the aircraft. At high altitude loiter the TMS will have a large low temperature heat sink, while during strike operations only a high temperature sink will be available. Thus the TMS will have to adapt to a radically changing environment. The EPS will have to provide power at high altitude where engine power available to drive a generator may be limited, and will be subject to a rapidly changing load characteristic and engine shaft speed during strike operations. This leads to a relatively complex EPS & TMS with a rich set of potential technology and component options.

The proposed aircraft would allow performing this type of missions with high accuracy (i.e. with high probability of begin able to identify and engage a target). Today, two vehicles are used. One vehicle constantly monitors a zone from high altitude. If targets are identified, then another vehicle is required to intervene. Besides logistic problems of having several types of vehicles, the success rate is low because the target might move from the position where it was first identified.

For the ISR segment, endurance is the most important concern. Endurance is achieved by running the aircraft in a very efficient region where the engines consume very little fuel (they run almost idle). Typically, this is achieved with high-bypass engines. The electrical power system will have to provide the necessary power for sensors which might be very high (laser sensors and radars). The thermal management system can rely on cold outside air (although air density is low and fuel flow rate is low as well).

In the strike mode, the aircraft has to reach a very high speed. In this regime, the engines will have to provide very different performance. For a safe strike maneuver, the speed of the aircraft needs to be very high (Mach 1.5 or above) which requires low bypass engines. The electrical power system and the thermal management system will have to switch to a different regime. In particular, the TMS will now have to provide a very high fuel rate and reject high heat with minimal availability of heat sinks (due to high speed and low thermal signature requirements). Notice that a design of the aircraft user the two regimes might not be the best solution. In fact the heat accumulated in the fuel after a strike might be rejected during a subsequent high altitude flight. This makes the design tools in use today inadequate since typically only steady state conditions are considered.

The requirements being very different in the two phases poses challenges in the selection of an optimal architecture because it is difficult to have mechanical parts (and electric parts) that are efficient and high performance at the same time. For example, gear pumps are more efficient at low flow rates, but generate a large amount of heat at high flow rates, whereas variable volume pumps don't. Thus, one might end up selecting hybrid architectures with both components that are activated depending on the mission to be performed. This makes the system more complex.

	Alt (ft)	Mach	Dur (min)
Engine Start	0	0	0
Taxi	0	0	5
Takeoff/Accel	0	0.3	1
Climb	19500	0.5	15
Cruise Out	39000	0.75	65
Loiter	36000	0.6	185
Descent S	20500	1.4	10
Stike	5000	1.4	20
Climb S	20500	0.5	15
Loiter	36000	0.6	185
Climb In	42500	0.65	15
Cruise In	49000	0.75	65
Descent	24500		18
Land	0	0	1

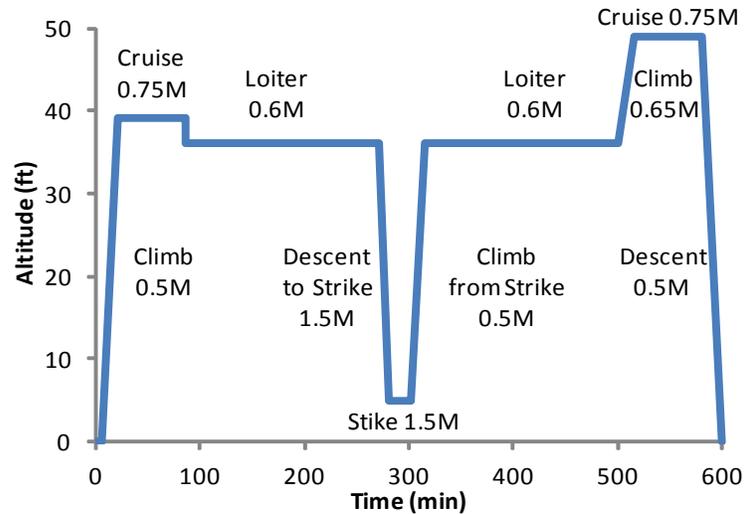


Figure 4: Detailed Description of the Mission Phases

3.5.2 Vehicle assumptions

We will assume that the engine has been selected, meaning that there exists an engine capable of delivering a certain level of performance required for the mission. We also assume that the airframe is given and that the payload has been fixed. We fix the following high level properties of for the aircraft that will set the context of the challenge problem:

- Take-off Gross Weight (TOGW) 1360 kg (3000 lbm)
- Mission endurance 10 hr
- Maximum altitude 15.24 km (50 kft)
- Cruise speed 0.75 Mn
- Strike speed 1.5 Mn
- Payload 900 kg (2000 lbm)
- Low observability

3.5.3 Details of the mission phases

Figure 5 show a detailed description of one sample trajectory of the mission profile. The mission includes the taxi mode where the vehicle is waiting to take off; a climb phase to ascend to a target altitude and to get to a loitering phase. After roughly three hours, the aircraft rapidly descend to strike phase which lasts as long as 20 minutes. It then climbs again to continue its ISR mission and eventually lands.

To capture a range of possible mission, we include probabilistic transitions among the phase so that several possible behaviors can be captured.

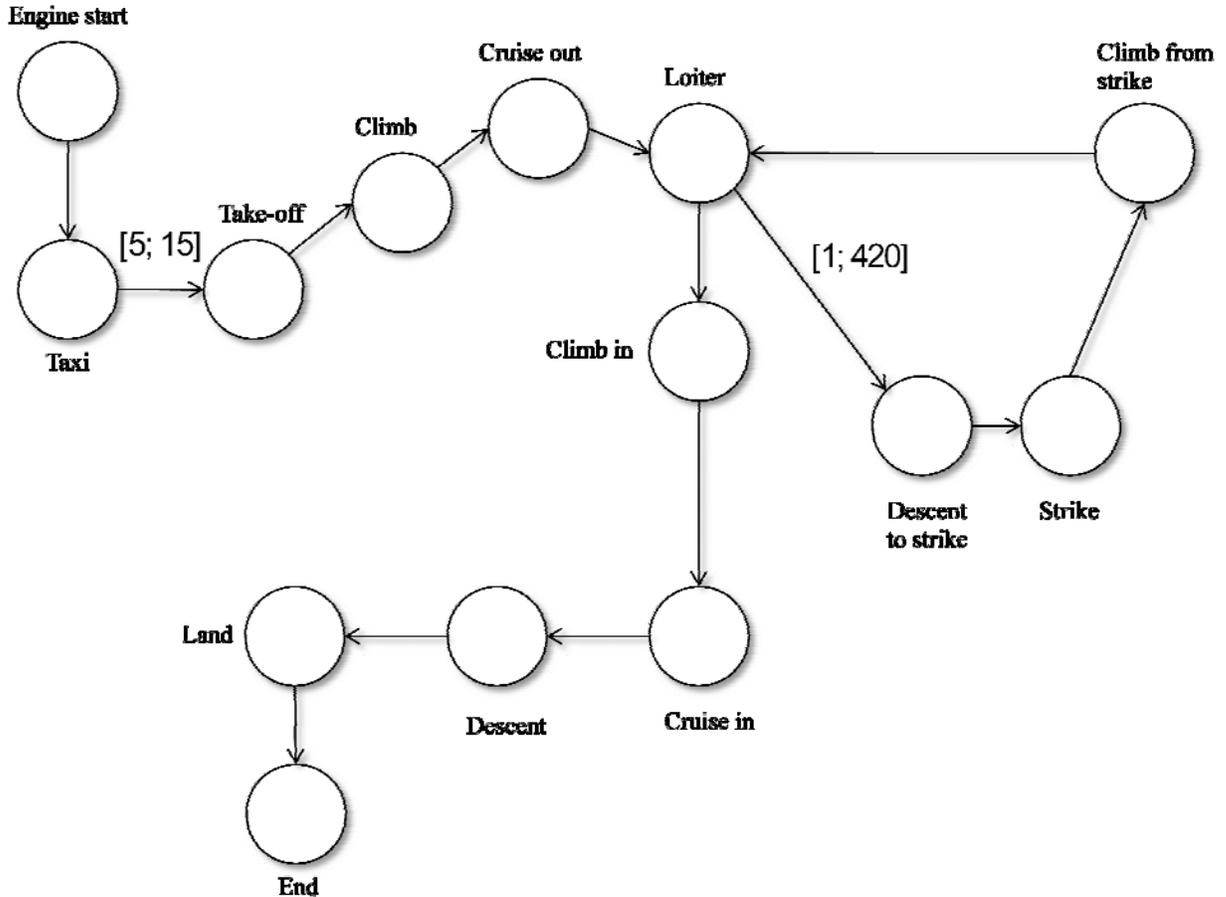


Figure 5: Description of the Mission Phases and Transitions Among Them

Figure 5 shows a representation of the mission where each phase is represented as a mode of operation. Transitions among modes can be guarded by some conditions that might refer to the state of the aircraft. This formalism will be used to formally capture the mission. The intent in this document is to show what are the parameters that we introduce to capture of family of missions. We assume that the taxi time is uniformly distributed between 5 and 15 minutes, and that the strike phase can begin anytime between 1 and 420 minutes from the beginning of the loitering mode. Other parameters might be considered later if the mission specification will need to be refined.

Table 1 describes the requirements in terms of power and heat loads that are fixed as assumed known in the system. The first 5 columns summarize the known power requirements that include weapons, sensors and vehicle management system, avionics and radar, and actuators. The second part of the table describes the heat loads that are assumed fixed. The first column is the heat load that needs to be rejected by the low temperature air circuit. The actuators and the payload are cooled using a low temperature liquid. The last column is the heat load coming from the engine that is transferred to the oil which in turn needs to be cooled to guarantee a maximum temperature of 325 F.

Table 1: Requirements for Each Phase of the Mission

Designation	Power System					Heat Load in (kW)				
	115 VAC	28 VDC	270 VDC			Low Temp Liquid Cooled (86 F)				High Temp Liquid Cooled
	Weapons Interface (kW)	Sensors & VMS(kW)	Avionics & Radar (kW)	Actuation (kW)	Sub-Systems (kW)	Low Temp Air (120 F)	Actuation Power	Payload	Total	Engine Oil (325 F)
Engine Start	0.00	0	0	7.0	0.0	3.0	0.0	0.0	0.0	11.0
Taxi	0.50	2.5	30.0	13.0	6.0	3.0	0.7	27.0	27.7	30.0
Take Off/Accel	0.50	2.5	30.0	64.0	6.0	3.0	1.5	27.0	28.5	32.0
Climb	0.50	2.5	30.0	15.0	6.0	3.0	1.5	27.0	28.5	28.0
Cruise Out	0.50	2.5	30.0	4.0	6.0	3.0	0.4	27.0	27.4	24.0
Loiter	0.50	2.5	30.0	3.0	6.0	3.0	0.3	27.0	27.3	24.0
Descent to Strike	2.5	2.5	20.0	56.0	6.0	3.0	5.6	16.0	21.6	32.0
Strike	2.5	2.5	20.0	56.0	6.0	3.0	5.6	16.0	21.6	32.0
Climb from Strike	2.5	2.5	20.0	56.0	6.0	3.0	5.6	16.0	21.6	32.0
Loiter	0.50	2.5	30.0	3.0	6.0	3.0	0.3	27.0	27.3	24.0
Climb to Cruise In	0.50	2.5	30.0	15.0	6.0	3.0	1.5	27.0	28.5	26.0
Cruise In	0.50	2.5	30.0	4.0	6.0	3.0	0.4	27.0	27.4	24.0
Descend	0.50	2.5	30.0	3.0	6.0	3.0	0.3	27.0	27.3	18.0
Land	0.50	2.5	30.0	3.0	6.0	3.0	0.3	27.0	27.3	18.0

3.5.4 Baseline system architecture

Figure 6 shows the architecture of a system that we will use as a baseline to compare the performance of our design. The system is divided into the electric power system (green and yellow), a thermal management system (brown), and an environmental control system (upper light cyan box) which in this design also integrates an auxiliary power unit. The engine is represented in the center of the figure.

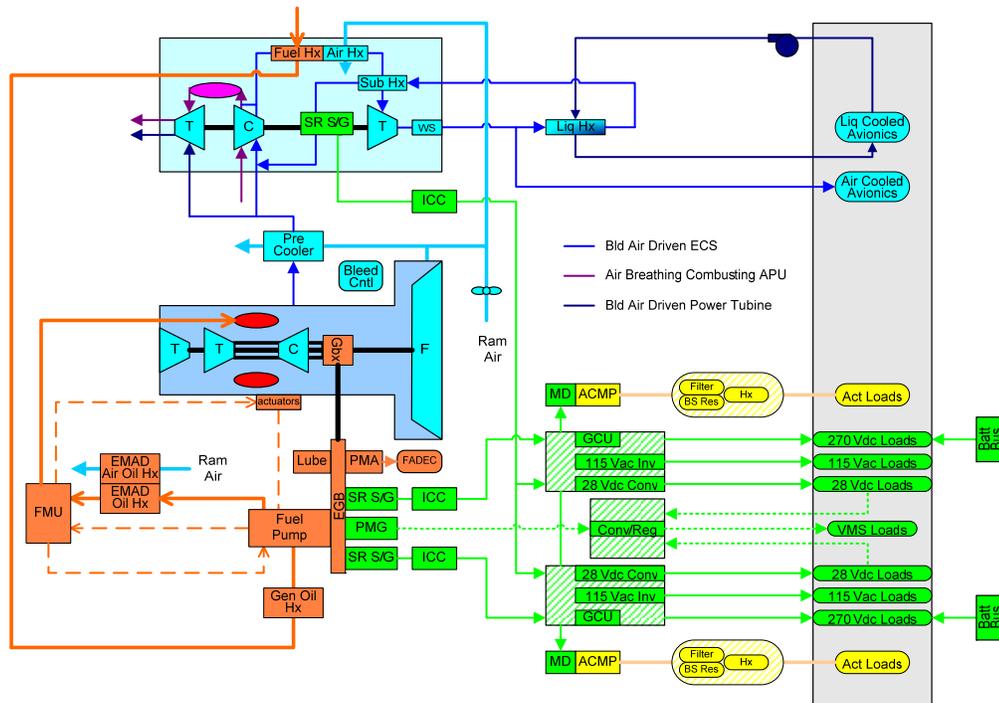


Figure 6: Baseline System Architecture

3.5.5 Formal definition of the mission requirements

The functional requirements are captured in the form of a contract. A contract is specified by a set of variables, a set of assumption and a set of guarantees. In this section we present a single model for the assumptions and guarantees of the system.

Figure 7 shows the inputs, outputs, states and parameter set of the formal representation of the functional requirements. The contract for the system requirements includes explicitly the fuel as an essential component of an aircraft. The fuel fixes most of the boundary conditions that need to be satisfied since it is the only source of energy and it is also used as a fluid for some of the actuators and as heat sink. The inputs to the system requirement specification are the total heat delivered to the fuel tank H_T , the fuel consumed by the engine(s) F , the temperature at the combustor V , the thrust provided by the engine m_f , and the voltage provided by the generators T_f . The outputs are the altitude h , the speed v , the heat generated by the known loads H , the temperature of the fuel in the tank T , the total fuel mass in the tank M , and the current absorbed by the actuators I . Notice that the voltage T_f and the current I are actually vectors where each entry corresponds to a specific actuator or load. Also notice that inputs and outputs are waveforms namely functions from the positive real numbers to appropriate domains. These functions don't need to be continuous, but we assume only left continuity.

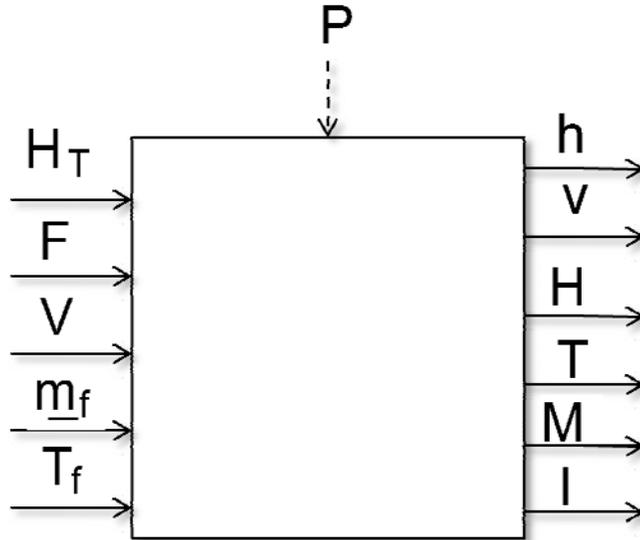


Figure 7: Inputs, States, Outputs and Parameters of System Requirements Specifications

This contract has a certain parameter set that includes the total weight of the aircraft not considering fuel and the weather conditions. We distinguish state variables from parameters although parameters can be seen as state variables that do not change in time.

The contract is specified as follows. Let \mathbf{u} be the vector of input signals and let \mathbf{x} be the vector of state signals. Also, let the set of discrete modes of the system be \mathcal{M} , standing for Engine Start, Taxi, Take Off, Climb, Cruise Out, first Loiter, Descent to Strike, Strike, Climb from Strike, second Loiter, Climb In, Cruise In, Descend, Land, End of Mission, and Error, respectively. In each mode $m \in \mathcal{M}$, the state variables evolve according to a set of (possibly stochastic) differential and algebraic equations $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, m)$, (where \mathbf{f} is a possibly multi-dimensional random process). The discrete state m is an accepting state

indicating that the assumptions of the contract have been violated. Transitions among discrete states $Tr \subseteq Q\{Err\} \times Q\{Err\} \cup Q \times \{Err\}$ are associated with guard conditions and reset maps. In general, guard conditions can be probabilistic.

A switching function $Sw : Q \times R^n \rightarrow Dist(Q)$ maps the hybrid state to a probability distribution over the discrete state space. A reset map $Res : Q \times R^n \times Q \rightarrow Dist(B(R^n))$ maps a hybrid state to a probability distribution over the set of states of a new mode. The set of transitions includes transitions to the accepting state. These transitions are taken whenever assumptions are violated. For example, in each mode there is a transition to the accepting state where the guard is define by $M \leq 0$, meaning that the hybrid dynamical system transitions to the accepting state if the aircraft runs out of fuel. The same transitions can be enables under conditions on the maximum fuel temperature, climb rate, strike speed etc.

The guarantees provided by the contract are modeled by the dynamics in each mode and by transitions not leading to the accepting state. Consider for instance the TX state and for the sake of this discussion consider the loads lumped as in [1]. The state evolves as follows:

$$\begin{aligned} \dot{h} &= 0, \dot{v} = 0 \\ \dot{i}_{28} &= \frac{v_{28}}{L} - R i_{28} \\ H &= H_A + H_L \\ H_A &= 3 \text{ kW}, H_L = 27.7 \text{ kW} \\ \dot{M} &= -\dot{m}_f \\ \dot{T} &= \frac{H_T}{c_f M} - \frac{\dot{m}_f(T_f - T)}{M} \end{aligned}$$

With initial conditions $h_0 = 0, v_0 = 0, i_{28,0} = 0, T_0$ a random parameter that depends on the weather conditions, and M_0 the initial fuel mass. We have only considered a lumped RL circuit as 28 DC load in this case. The relations between current and voltage of the other loads also need to be considered. A transition from the TX mode to the Err mode is guarded by the following condition:

$$\begin{aligned} |v_{28} - 28| &\geq 1 \\ T &\geq 315 \end{aligned}$$

Other conditions can be included by constraining for instance the derivative of the voltage. Clearly, the electric power system, the environmental control system and the thermal management system will be connected in a closed loop with this contract. The probability of entering the accepting state is the probability that the contract is violated by the system implementation.

domain, we will model the transition functions of software modules. We consider switching logics rather than continuous time controllers because the former is more complex than the latter.

3.6 Thermal management system

A thermal management system (TMS) consist of a set of heat loads and heat sinks where generally heat is either added to the system or removed from the system through heat exchangers, pumps and fans for pressurizing the flow, and finally pipes/ducts and valves for distributing and controlling the flow. For the challenge problem air vehicle there are two primary heat sinks available, ram air and fuel, and one intermediate heat sink available, oil (oil ultimately has to reject heat to ram air or fuel). For this example problem, four heat loads are considered:

A baseline fuel based TMS is shown in Figure 8. This system uses fuel as the primary heat sink. Fuel is taken out of the tank, routed through various aircraft system load heat exchangers, driven by a main fuel pump (MF Pump) which provides the pressure rise necessary to move fuel through the main loop and pressurize the main engine fuel nozzle. The actuation pump (Act Pump), which further boost fuel pressure, moves fuel in a recirculating loop, supplying engine actuators with flow. This pump is sized for the worst case actuator demand flow, thus at most operating points it provides more fuel flow than is required by the actuators. The excess flow is returned to the main fuel loop downstream of the main fuel pump. The flow exiting the engine actuators is returned to the main fuel loop upstream of the main fuel pump. The flow exiting the engine actuators is returned to the main fuel loop upstream of the main fuel pump. After the split to the actuator pump, the main loop supplies fuel to the engine oil cooler (Eng Oil). The fuel is then burned in the engine (To Engine). If the fuel temperature after the engine oil cooler exceeds the maximum allowable fuel temperature, excess flow is taken from the tank. After the engine oil cooler, the excess flow is returned back to the tank, while the burn flow goes to the engine. The return fuel is cooled back to tank temperature by the air fuel cooler. This fuel cooler is supplied with outside ram air through a ram air scoop. During ground operations a fan pulls air through the scoop and fuel cooler.

There are six types of components in the TMS; heat exchangers, pumps, fans, pipes/ducts, valves, heat loads and splitters/mergers. For each type, at each abstraction layer, component models are formulated that predict the component outlet state (pressure, temperature and flow rate) and power consumption, given inlet conditions and a set of characteristic parameters (e.g. pump efficiency or a pump efficiency map). At specific mission sizing conditions, these component models also size the component, providing weight and off-design characteristic parameters.

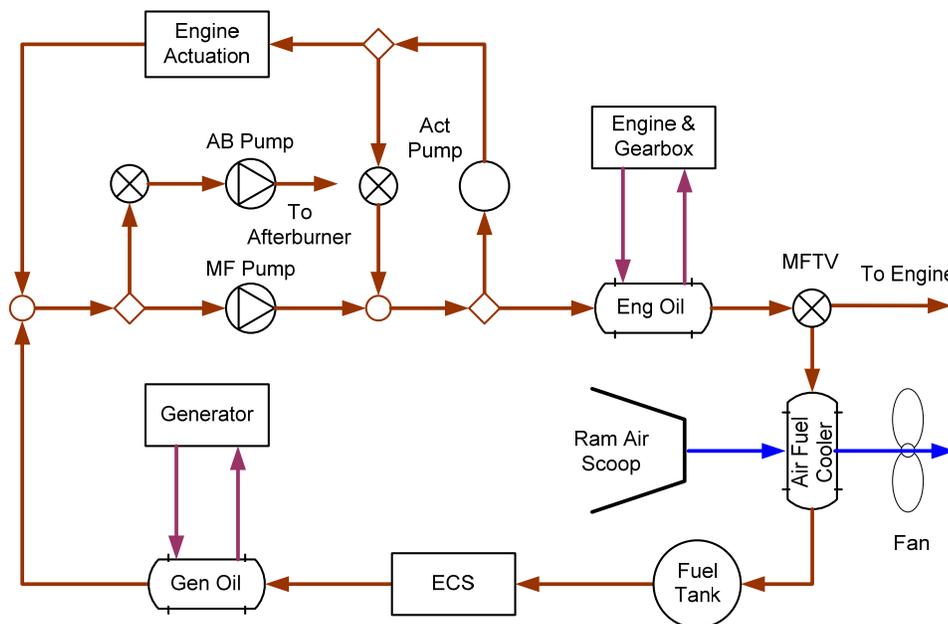


Figure 8: Baseline Architecture of a Thermal Management System

3.6.1 Schematic representation

A system model is formed by joining component inputs and outputs, according to a set of rules. To facilitate this, we use a simple line diagram. For the example in Figure 8, the simple line diagram is shown in Figure 9 where sources are represented by solid circles and sinks by hatched circles. Components are connected between the sources and the sinks.

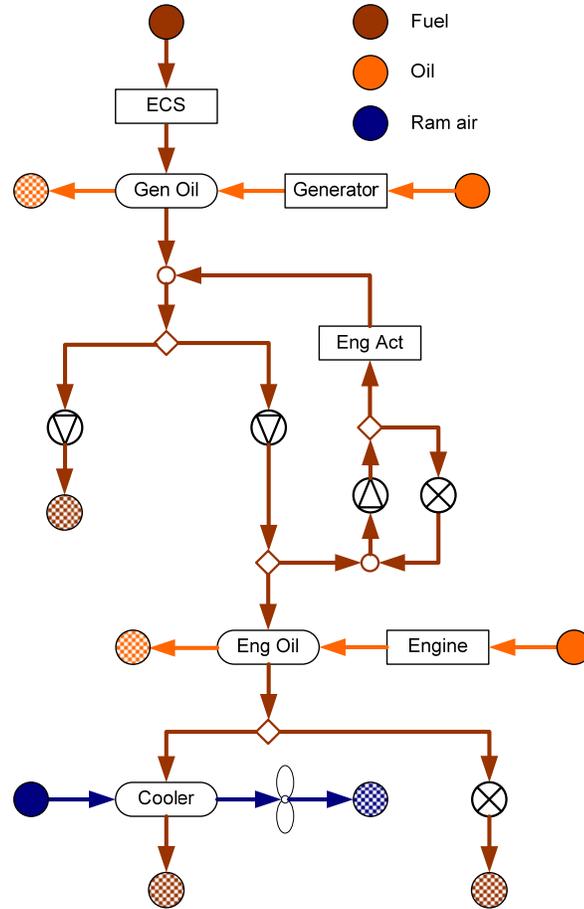


Figure 9: Line Diagram of the Baseline sSystem

3.6.2 Detailed component models

The most detailed abstraction level that we consider includes several details such as the detailed geometry of each component, the materials and the type of fluid. These models are already available as part of the HEATS tools developed at UTRC. In this tool, models are available as both equation-based models and scaled performance maps based on their sizing point. Heat Exchangers models are physics based and can be used for sizing and performance analysis.

3.6.3 Abstract component models

We will develop abstract models starting from detailed ones in later sections in this document. However, standard abstractions can be constructed by making some additional assumptions on the environment in which components are going to be used. Specifically, we assume

the pressure rise across a pump to be fixed. Efficiency, power consumption and fuel temperature rise are calculated from the characteristics of the pump. The weight is derived as function of the maximum input power. We use the following convention for variables: X_j^i refers to the X variable at the *input* of component j . We use F for flow rate, P for pressure, T for temperature, Q for heat and W for power. For a pump we use the following abstract model

$$F_i^i = F_i^o, \Delta P_i = P_i^o - P_i^i$$

$$Q_i = F_i \Delta P_i (1/\eta - 1), Q_i = F_i C_p (T_i^o - T_i^i), W_i = f(F_i \Delta P_i / \eta_i)$$

where η is the efficiency of the pump and C_p is the specific heat of the fuel. Assuming $\Delta P_i = 6.9 \text{ Mpa}$, the fluid density $\rho = 800 \text{ Kg/m}^3$ and efficiency $\eta = 0.3$ off design point and $\eta = 0.6$ at design point, then we obtain the following simple model:

$$W = \frac{\Delta P_i F_i}{\rho \eta}$$

$$\text{Weight} = 0.3 W_{max}$$

where W_{max} is the power required for the maximum fuel rate which is 1.9 Kg/s for a main fuel pump, 3.2 Kg/s for an afterburner pump, and 0.64 Kg/s for the actuator pump. Moreover, the efficiency for the actuator pump is 0.05 off design point and 0.6 at design point.

Heat Exchangers are modeled by using a fixed effectiveness number. The liquid side pressure drop is also considered fixed while the air side pressure drop varies. The weight is derived as function of effectiveness and the air side pressure drop. We use the superscript h to the hot side of the heat exchanger and c for the cold side. The following set of equations represents the model:

$$Q_i = M_i^h C_{p_{ih}} (T_i^{ho} - T_i^{hi}) = \eta_i M_i^c C_{p_{ic}} (T_i^{co} - T_i^{ci})$$

$$F_i^{ih} = F_i^{oh}, \Delta P_{ih} = P_i^{oh} - P_i^{ih}$$

$$F_i^{ic} = F_i^{oc}, \Delta P_{ic} = P_i^{oc} - P_i^{ic}, W_i = f(\eta_i, \Delta P_{ic})$$

The parameters and weight models for the heat exchangers depends on their geometric properties and type (e.g. plate fin or tube fin). Consider for example the air/fuel heat exchanger used to cool down fuel right before sending it back to the tank. Then, a surrogate model for the weight is the following:

$$\text{Weight} = (0.8 - 0.22 \ln(\Delta P_{ic})) (639 \text{ eff} f^{3.3})$$

where eff is the effectiveness of the heat exchanger and the pressure drop on the cold side (i.e. air) is assumed to be $\Delta P_{ic} = 3450 P_{ic}$

Fans are modeled using a fixed efficiency number. The weight is derived as a function of maximum input power

$$F_i^i = F_i^o, \Delta P_i = P_i^o - P_i^i$$

$$Q_i = F_i \Delta P_i (1/\eta - 1), Q_i = F_i C_p (T_i^o - T_i^i), W_i = f(F_i \Delta P_i / \eta_i)$$

The power and weight models are as follows:

$$W_i = \frac{F_i C_p T_i^i \left(\rho_f \frac{\gamma - 1}{\gamma} - 1 \right)}{\eta}$$

$$\text{Weight} = 250 W_i$$

Nominal values for the parameters are $\gamma = 1.4$ and $\eta = 0.7$.

Finally, pipes and ducts are assumed to be ideal (i.e. no pressure drop):

$$F_i^i = F_i^o, T_i^i = T_i^o, P_i^o = P_i^i$$

These models will be checked and refined using automatic tools later in the section dealing with abstraction layers.

3.7 Electric power system

An aircraft electric power system (EPS) consists of four basic building blocks: generators, switches & transmission-lines, converters, and electrical loads. These building blocks are

interconnected to form aircraft EPS topologies. The main function of an EPS topology is to generate and deliver stable continuous power to its electrical loads. During EPS operation, certain parameters must be maintained within predefined limits. Those parameters include voltage, current, power, frequency, and heat throughout the system. Other parameters associated with an EPS topology include weight, volume, and cost, which depend on power flow and those parameters can be approximated during the design phase via look-up tables. For example, after electrical load power requirements are defined, power flow analysis can be used to define or fine tune the required power rating of transmission lines and switches for a given topology. Then, power flow analysis can be used again to ensure that requirement compliance is maintained. This document presents a method to evaluate an aircraft EPS topology with the use of two evaluation tools; a steady state EPS model and a dynamic EPS model. The tools evaluate the power flow throughout the EPS topology which reveals whether an EPS topology meets design requirements. Furthermore, it is expected that many EPS topologies will satisfy the design requirements which will allow for an optimal EPS topology to be chosen among them. For example, the optimal EPS topology may be optimized for weight, cost, volume, losses (heat) or some combination of them.

3.7.1 EPS library

This document focuses on the tools developed to evaluate an EPS topology, but first a method for creating a topology is established. Typically, an aircraft EPS topology is composed of a primary and a secondary distribution system. A simplified generalization of the two systems is that the primary distribution system will source major loads, while the secondary system sources many smaller loads that can be lumped together and modeled as major loads to the primary distribution system. The modeling work done for the META II project has focused on the primary distribution system components and their interconnections. The major components that make up an EPS topology have been identified from aircraft systems literature and expert knowledge from Hamilton Sundstrand [1], [2]. Table 2 summarizes the various components that make up a primary distribution system which are categorized into the four basic building blocks of an EPS. Each EPS component will have variants, for example a generator is defined as any source of electric power such as a primary generator, auxiliary power unit generator, ram air turbine generator, external power source, or a battery. Transmission lines will vary in their wire gauge (AWG) which depends on the power flow requirement. Switches vary with power requirement and their functionality, for example, a battery and a load may use different switch types despite the same or similar power level requirement. The fourth component type is electric loads which are used as inputs to an EPS topology formulation. Many factors are considered when construction an EPS topology, currently legacy designs and expert knowledge are the main drivers in the design process. This work formulates a sample topology by interpreting a set of composition rules derived from experts at Hamilton Sundstrand in combination with trends found in aircraft systems literature.

3.7.2 Composition Rules/Requirements

Formulation of an EPS topology begins with the electrical load definitions which define the system power requirements. The EPS topology must be able to support the loads in all operating modes. Furthermore, steady state and dynamic bus voltage limits must be upheld in all operating modes. In addition to the EPS library of Table 1, electrical load specifications and composition rules are required in order to create a candidate EPS topology. The electrical load specifications must be defined for all operating modes such as maintenance-mode, take-off-mode, cruise-mode, emergency-mode, landing-mode, etc. Electrical load specifications include their rated capacity

in kVA and their power factor (PF), their reliability (essential vs. non-essential), and their electricity type which indicates the required input voltage magnitude and type whether it is an AC or DC load. The composition rules were derived from the EPS requirements list below:

- Satisfy power quality under all conditions (voltage levels, distortion, etc)
- Load capacity of any component shall not be exceeded in steady state
- AC buses shall never be paralleled
- Faulted buses must be permanently isolated
- Meet all load reliability requirements while aircraft is available for revenue dispatch
- All buses must be kept powered in case of one failure beyond Minimum Equipment List (MEL)
- A select set of critical buses must remain powered in the case of two failures beyond MEL
- Breaks in an AC subsystem must be less than x milliseconds
- Breaks in a DC subsystem must be less than y milliseconds
- Only part of the aircraft shall be powered when in maintenance mode (e.g. left DC side)
- Do not parallel transformer rectifier units (TRU) in steady state
- Minimize contactor actuations during transfers
- Use the APU only during take-off-mode, landing-mode or emergency-mode
- In-air start of APU limited to less than z feet of altitude
- No single point failure shall cause the loss of all critical busses
- Power to select loads must be maintained for thirty minutes if loss of all mechanical sources

The EPS requirements list is examined in order to extract details that can be used to formulate an EPS topology. However, first the minimum equipment list (MEL) is further explained; a MEL is defined as the list of EPS components that are required to be operational in order for an aircraft to be dispatched for revenue. For convenience, a MEL can be defined by the components that an aircraft may fly without. For the items listed above, the MEL is defined by one failed generator and one failed converter (TRU). Further explanation/interpretation of the EPS requirements is provided next.

Satisfy power quality under all conditions (voltage levels, distortion, etc). Once a topology is finalized, the steady state voltage magnitude at every bus must be maintained within a given limit of the nominal voltage magnitude. For the sample topology in the next section, $\pm 6.5\%$ was chosen. A tolerance of $\pm 6.5\%$ was chosen because it would allow the rated current ($2 \times 350\text{A}$) to flow through 1/0 AWG wire up to 100ft while staying within tolerance). The tolerance will vary from one EPS design to another, but ultimately the tolerance must be specified by the customer. Topology voltage drops are calculated via load flow analysis with the steady stated evaluation tool. Transient evaluation is also performed on a topology to observe the behavior of the bus voltages when switches are actuated. Acceptable voltage waveforms as a function of time must be specified by the customer or component supplier for each component. In addition to the voltage requirement, a frequency variation limit is imposed not to exceed $\pm 2.5\%$ of the nominal. The acceptable variation for the sample topology is based on electric power systems literature [3], however the actual limit for a given EPS design must also be provided by the customer.

Load capacity of any component shall not be exceeded in steady state. This requirement addresses the distribution of loads among the various power sources. Composition rules must

ensure that the topology does not overload any component (e.g. generator, switch, transmission line, and converter). Composition rules will try to evenly distribute the loads among generators by imposing a maximum load limit per generator. Non-primary distribution buses will also have a limited power flow capacity which will drive the topology design. Also maximum load per TL and SW are determined by their respective rated capacity.

AC buses shall never be paralleled. Never paralleling AC buses means that no two or more AC source should ever have a common conductive path. For example, if an EPS consists of two primary generators and two buses where Gen1 is powering Bus1 and Gen2 is powering Bus2, then Bus1 and Bus2 should never have a common conductor connecting them. However, a “standby” connector (switch and transmission line) is allowed between AC buses. That will allow for either bus to be powered in case one of the generators fail.

Faulted buses must be permanently isolated. This requirement implies that a topology must be designed in such a way that in the case of a bus failure (e.g. short to ground), sufficient switches must exist so that the faulted bus can be electronically disconnected from the EPS network.

Meet all load reliability requirements while aircraft is available for revenue dispatch. Load reliability is addressed in terms of independent lanes of electricity (ILE) and independent sources of electricity (ISE). The required ILE of a load is defined as the number of independent conductive paths between the load’s bus and the powering generator’s bus. Essential loads will generally require more than 1-ILE, however it should be noted that only 1-ILE is active at any given time. The other ILE’s are put in place to supply the load in case the first lane fails. The required ISE of a load refers to the number of sources that can power a load via independent or common lanes of electricity. The sample topology consists of loads with two reliability levels; (a) essential loads require 2-ILE and 3-ISE and (b) non-essential loads require 1-ILE and 2-ISE. For any other EPS design, the customer must specify the ILE and ISE for each load. An aircraft is considered available for revenue dispatch as long as the MEL is fully functional. In other words, for this example as long as only one generator and one converter has failed, then the aircraft EPS topology must be capable of powering all loads and must meet their ILE and ISE requirements.

All buses must be kept powered in case of one failure beyond Minimum Equipment List. When one failure (any component) beyond the Minimum Equipment List (MEL) occurs all buses must continue to be powered. Sufficient switches must be put in place to isolate a failed component and ILE’s must be added to energize any bus that may have lost power due to the failure. In the case that a bus fails (e.g. thermal fatigue or short to ground), then that bus must be isolated via switches and the loads connected to that bus will be lost. It is assumed that the lost load functions are fulfilled by mechanical backup systems. If a generator fails, then it may be necessary to shed non-essential loads because the total generation capacity may be significantly reduced. A list of loads to be shed in case of a generator failure beyond MEL must be provided by the customer. In summary, in case of one failure beyond the MEL, the topology must be able to power all buses (except for failed buses). Load shedding of non-essential loads is allowed and once the aircraft lands, the failure must be repaired before it can be dispatched for revenue again. Lastly, the ILE and ISE requirements are reduced when the additional failure occurs. Only one ILE and two ISE are required for essential loads while non-essential loads can lose their ILE.

A select set of critical buses must remain powered in the case of two failures beyond MEL. This requirement requires the topology to have SW’s and TL’s in place to isolate two additional failures beyond MEL and power select buses as specified by the customer. In the sample

topology, all essential buses are considered critical and should be powered at all times (except if the failures are essential buses). Further load shedding is allowed if generation capacity is reduced and once the aircraft lands, the failures must be repaired.

Breaks in an AC subsystem must be less than x milliseconds. This requirement sets the maximum time allowed for the EPS controller to detect a de-energized bus and to actuate the appropriate switches to re-energize the bus. The composition rules are not directly impacted by the time requirement since it is assumed that the EPS controller will be able to complete the actuation. However, the time requirement is used to carry out dynamic simulations of the switch actuations. The simulation results are then used to determine whether bus voltages are maintained within specified limits.

Breaks in a DC subsystem must be less than y milliseconds. Similar to the AC actuation time, the detection and actuation of a de-energized bus must be completed within a time limit specific by the customer. The specified also used for dynamic simulations and the results are used to determine whether bus voltages are maintained within specified limits.

Only part of the aircraft shall be powered when in maintenance mode (e.g. left DC side). In maintenance mode, the aircraft is on the ground and is power through its external power connection (EPC) to a ground power unit (GPU). A list of loads that must be powered while in maintenance mode is provided by the customer. The total maintenance mode load capacity will determine how much power will be needed from the GPU(s). Normally one GPU will deliver a max power of 90kVA, therefore if the load capacity exceeds 90kVA then an addition EPC will be needed. The limit of 90kVA is typical of most airport GPU's according to aircraft systems literature [4].

Do not parallel transformer rectifier units (TRU) in steady state. Similar to the requirement that prohibits paralleling AC buses, the idea is to not parallel DC buses that are energized by TRUs while in steady state operation. For example, if a topology contains two TRU's and two buses were TRU-1 powers bus-1 and TRU-2 powers bus-2, then bus-1 and bus-2 should not have a common conductive path. The exception to this rule is during transient operation, if another requirement exists which states that no break can occur on dc buses, then it is necessary to momentarily parallel TRU's while the appropriate switch actuation occurs.

Minimize contactor actuations during transfers. Minimizing contactor/switch actuations is desired for two reasons; (a) to minimize wear of the actuator and (b) to minimize dynamics in the EPS. Metrics for comparison may be number of switches actuating, total time of actuation, or a measure of the voltage/current transients.

Use the APU only during take-off-mode, landing-mode or emergency-mode. This requirement will drive the power rating requirement of the APU generator. The load capacity during take-off-mode must be supplied by the APU generator, therefore the APU generator capacity must be sized accordingly. The sizing of the APU generator is also influenced by the MEL. When a primary generator fails, the APU generator must be able to offset the loss of generation capacity which will allow for the aircraft to be dispatched for revenue service. A primary generator is typically sized to operate at 80% of rated generating capacity when power all assigned loads [1], [2].

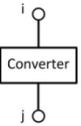
In-air start of APU limited to less than z feet of altitude. The maximum altitude may be expressed as a percentage of the cruise altitude in a flight mission/profile. This requirement means that if the aircraft is above the maximum APU starting altitude then the aircraft must climb down before starting the APU and then climb back to cruising altitude. Load shedding

may be necessary (if primary generator has failed) and only critical loads may be powered until the APU generator is operational.

No single point failure shall cause the loss of all critical buses. If one component fails, then at least one critical bus must be energized. Testing this requirement may require power availability tests of all critical buses after removing an EPS component. The test shall be repeated for all EPS components.

Power to select loads must be maintained for thirty minutes if loss of all mechanical sources. If all mechanical loads sources are lost, then all generators will fail. Therefore backup battery power must exist in order to power all critical loads for a minimum of thirty minutes. Therefore, the battery capacity will be determined by the critical loads' capacity.

Table 2: EPS Library: Four Basic Building Blocks

EPS Component	Symbol	Component Options
1. Generator		Primary Generator (G), Auxiliary Power Unit Generator (G _{APU}), Ram Air Turbine Generator (G _{RAT}), External power source (P _{EXT}), Battery Power (Batt)
2. Switches & Transmission-Lines		Generator Circuit Breaker (GCB), Bus Tie Breaker (BTB), Auxiliary Power Breaker (APB), External Power Contactor (EPC), Electronic Load Control Unit (ECLU), Transmission Line (TL)
3. Converters		DC to AC Converter - Static Inverter (DC/AC), AC to DC Converter - TRU (AC/DC)
4. Electrical Loads		Load 115VAC - 230VAC 3Ø (LD _{AC}), Load 28VDC (LD _{DC})

3.7.3 Sample topology

Thus far, definitions for an EPS component library and composition rules have been presented. To develop an EPS sample topology, a set of aircraft loads is required which will serve as the input to the topology development. A set of thirty six loads was derived from typical loads found on a Boeing-777 commercial aircraft [5]. All loads are listed in Table 3. The table contains the load capacity and power factor at rated power. Some of the loads are essential loads while others are non-essential loads. The essential loads are those powered by buses whose label is preceded by and “E” (e.g. E_B3 signifies essential bus number three); (a) essential loads require 2-ILE and 3-ISE and (b) non-essential loads require 1-ILE and 2-ISE. The EPS requirements from the previous section were applied to the load list in Table-2 to produce the EPS sample topology illustrated in Figure 10. The topology consists of three sources and thirty six loads with a network of switches and transmission lines interconnecting them. In order to evaluate the topology with the steady state and dynamic evaluation tools, a topology netlist format was created to capture the EPS topology network interconnections. Table 4 lists the netlist content which captures the necessary data for evaluation. There are five sets of data

(matrices) describing the interconnection of the sources, switches, transmission lines, converters, and loads. In addition to connectivity information, the data contains load capacity data and the type of electricity require by each component. A sixth matrix (EPS_Code) captures the different types of electricity that may exist in a topology. For example, a topology may consist of AC power at different voltage magnitudes and/or number of phases. All the information contained in the netlist utilized by the evaluation tools.

Table 3: Sample Topology Load List

Bus No	Load Index	Load Description	Power (kVA)	Power Factor
E_B3	Load 1	Navigation Lights (NL)	2.56	0.97
	Load 3	Fuel Pump (FP)	6	0.97
	Load 5	Avionics (A)	2.56	0.97
	Load 7	Ice/Rain protection (IRP)	2.6	0.97
	Load 9	Equipment Cooling Fan (ECF)	4.64	0.9
	Load 11	Fuel boost pump (FBP)	7.04	0.9
E_B4	Load 13	Hydraulic pump (HP)	14.24	0.9
	Load 15	Window Heat (WH)	0.8	0.97
E_B11	Load 17	DC Load	6.08	1
B7	Load 19	Navigation Lights (NL)	2.56	0.97
	Load 21	Fuel Pump (FP)	6	0.97
	Load 23	Avionics (A)	2.56	0.97
	Load 25	Ice/Rain protection (IRP)	2.6	0.97
	Load 27	Equipment Cooling Fan (ECF)	4.64	0.9
	Load 29	Fuel boost pump (FBP)	7.04	0.9
B8	Load 31	Hydraulic pump (HP)	14.24	0.9
	Load 33	Window Heat (WH)	0.8	0.97
B13	Load 35	DC Load	6.08	1
E_B5	Load 2	Navigation Lights (NL)	2.56	0.97
	Load 4	Fuel Pump (FP)	6	0.97
	Load 6	Avionics (A)	2.56	0.97
	Load 8	Ice/Rain protection (IRP)	2.6	0.97
	Load 10	Equipment Cooling Fan (ECF)	4.64	0.9
	Load 12	Fuel boost pump (FBP)	7.04	0.9
E_B6	Load 14	Hydraulic pump (HP)	14.24	0.9
	Load 16	Window Heat (WH)	0.8	0.97
E_B12	Load 18	DC Load	6.08	1
B9	Load 20	Navigation Lights (NL)	2.56	0.97
	Load 22	Fuel Pump (FP)	6	0.97
	Load 24	Avionics (A)	2.56	0.97
	Load 26	Ice/Rain protection (IRP)	2.6	0.97
	Load 28	Equipment Cooling Fan (ECF)	4.64	0.9
	Load 30	Fuel boost pump (FBP)	7.04	0.9
B10	Load 32	Hydraulic pump (HP)	14.24	0.9
	Load 34	Window Heat (WH)	0.8	0.97
B14	Load 36	DC Load	6.08	1

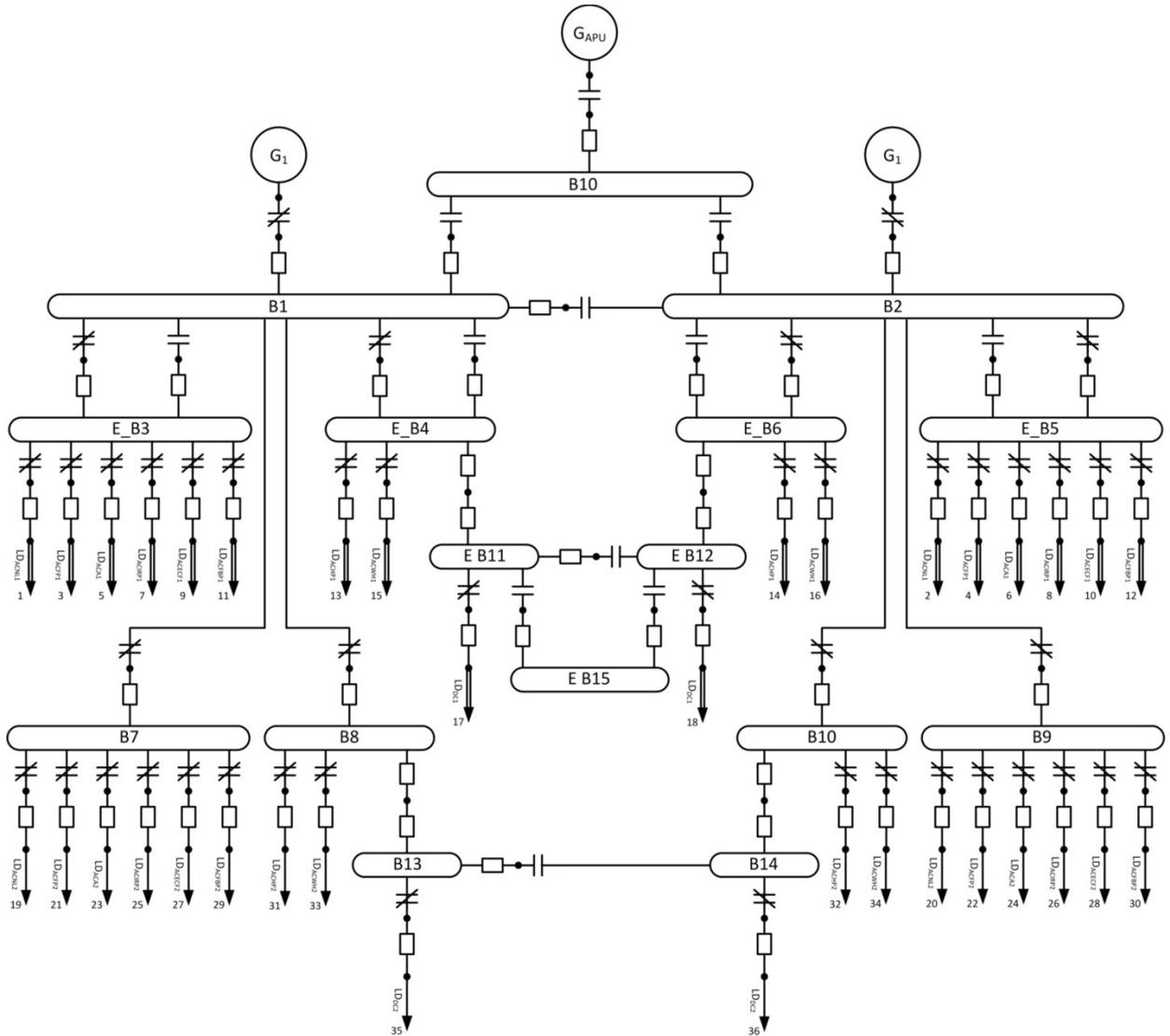


Figure 10: Sample Topology Used for EPS Evaluation Tool Development

Table 4: Netlist Data Structure

Input	Matrix Columns					
	1	2	3	4	5	6
Source_data	Bus No.	Bus Code	EPS Code	Real power	Reactive power	-
SW_data	SW No.	EPS Code	Node 1	Node 2	Impedance	-
TL_data	TL No.	EPS Code	Node 1	Node 2	Impedance	-
Conv_data	Conv No.	Bus In	Bus Out	EPS Code In	EPS Code Out	Efficiency
Load_data	Bus No.	Bus Code	EPS Code	Real power	Reactive power	-
EPS_Code	Code No.	Voltage	No. of Phases	DC = 1, AC = 2	-	-

3.7.4 Evaluation tools: steady state EPS model & dynamic EPS model

Given electrical load specifications, components from the EPS library were interconnected in way to uphold the composition rules to finally yield a sample EPS topology. Performance metrics can be assigned to the topology through the application of two evaluation tools, steady state and dynamic EPS models. Both types of models calculate the voltage and power at each bus, which are then compared to the bus voltage/power limit requirements. In general, when evaluating multiple feasible topologies, it is expected for all the feasible topologies to meet the voltage limit requirements since those limits were abstracted into the composition rules. However, some topologies will have less voltage deviation from nominal compared to others and that is how one EPS topology may be considered superior. Figure 11 below summarizes the evaluation tools.

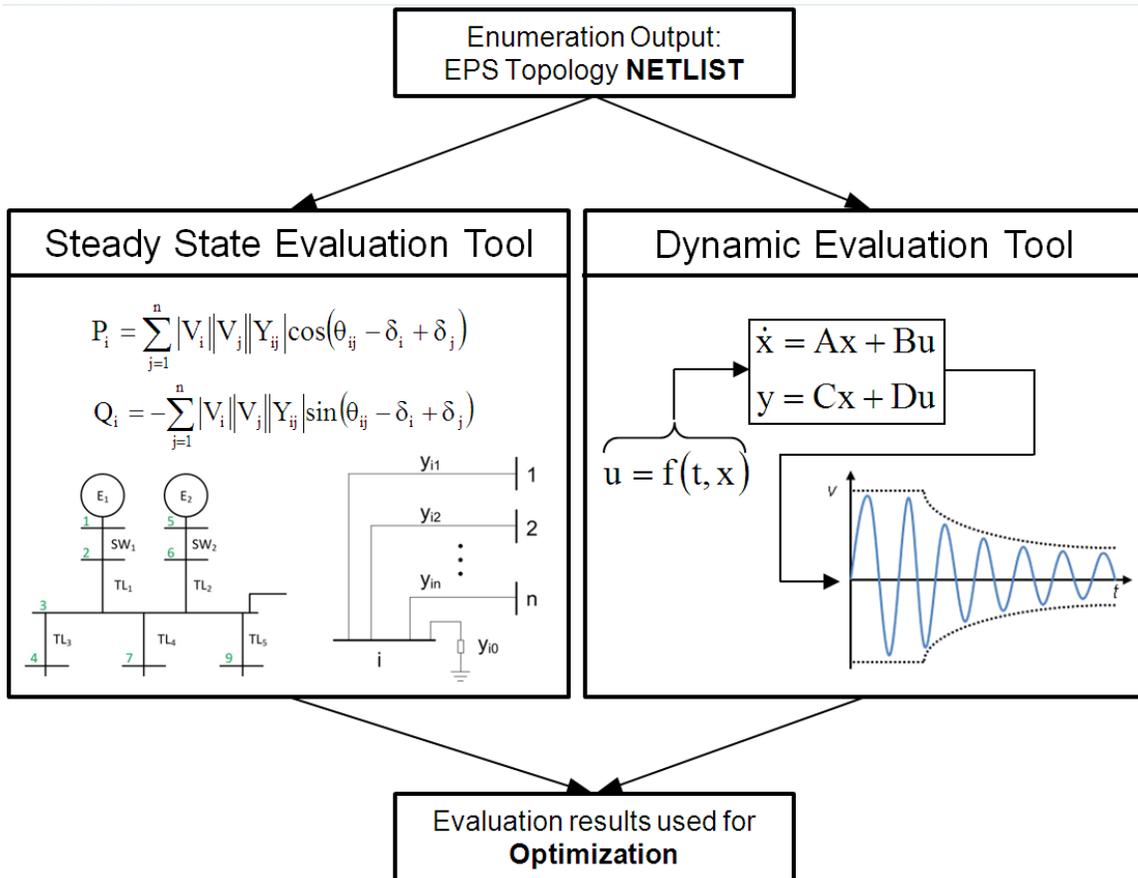


Figure 11: EPS Topology Evolution Tools Summary

3.7.4.1 Steady-state EPS model

An EPS topology is a circuit network interconnecting sources and loads. Load power requirements are known rather than impedances which results in network equations in terms of power known as power flow equations. Power flow equations are complex nonlinear algebraic equations; therefore they must be solved by iterative methods. A set of equations (1) & (2) shown in Figure 12 are written for each bus in the topology. The equations are coupled through the admittance values (Y_{ij}) which captures the component interconnections. Solving the two

equations for all nodes, produces the power flow solution [2]. With the power flow solution in hand, the voltage and current for all EPS components under steady state conditions are known. The steady state analysis is carried out for any combination of switch states. For example, if the steady state bus voltages are of interest only during maintenance-mode where only part of the EPS is powered, then the switch data matrix in the netlist can be modified to “turn-off” the appropriate switches which models a maintenance mode state and repeat the load flow analysis. Other switch states may model other modes such as take-off-mode, cruise-mode, emergency-mode, etc. Each set of results reveals how well the bust voltages adhere to the bus voltage limits that are defined in the EPS requirements.

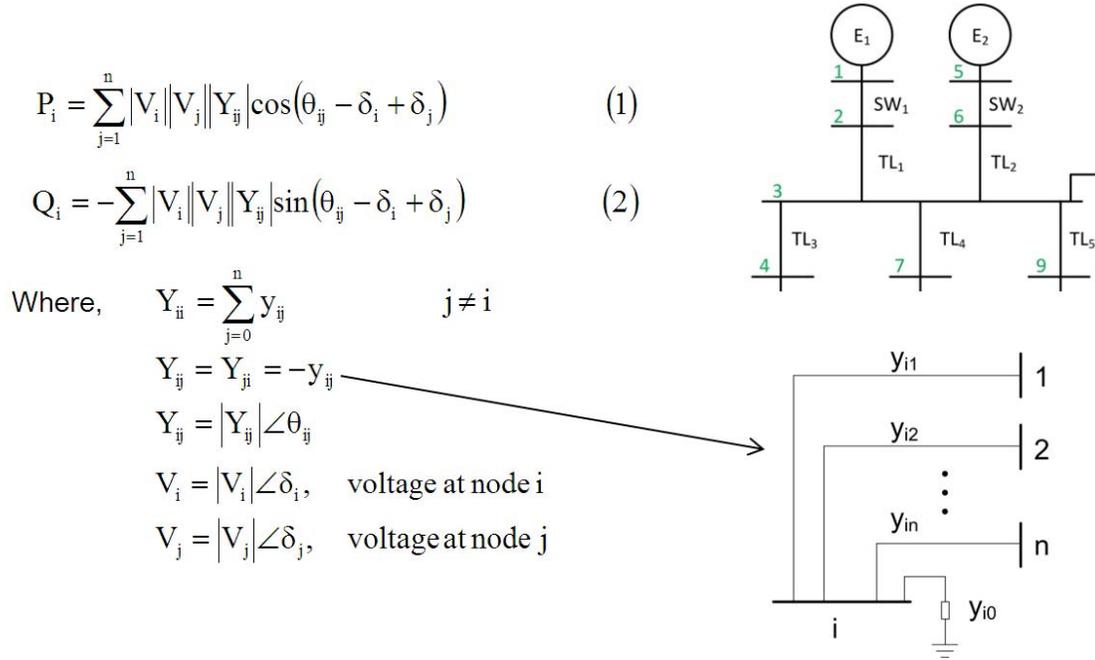


Figure 12: Power Flow Analysis Governing Power Equations

3.7.4.2 Dynamic EPS model

Similar to the steady state EPS model, the input for the dynamic EPS model is the netlist outlined in Table 4. The procedure created to automatically formulate the state space model of an EPS topology (Figure 13) is outlined in Figure 14. Common graph theory techniques were implemented in order to represent the EPS network with a set of interconnecting branches. A matrix called the connectivity matrix “C” was generated to capture the branch interconnections [5]. Equation 3 in Figure 4 shows voltage loop equations in terms of the representative branches can be used to formulate a governing equations that can be represented in reduced-row-echelon form (RRE). The RRE representation of the governing equations yield the state space coefficient matrices “A” and “B” which are then used to evaluate dynamic behavior of the EPS topology [3], [6]. Once the network is represented in with a state space model, one or more external sources can power the model which represents a complete EPS topology. All dynamic EPS components are modeled with an inductance (L), resistance (R), and a voltage source (S) connected in series. For each dynamic simulation, an EPS topology network is represented by a state space model whose states are the electrical load currents (the number of states will equal the number of loads). Solution of the state space yields the instantaneous load currents which are

used to calculate voltages of interest throughout the EPS topology. The results are used to verify that the EPS dynamic requirements are met and can be fed to an optimization tool to reveal the better performing topology(s).

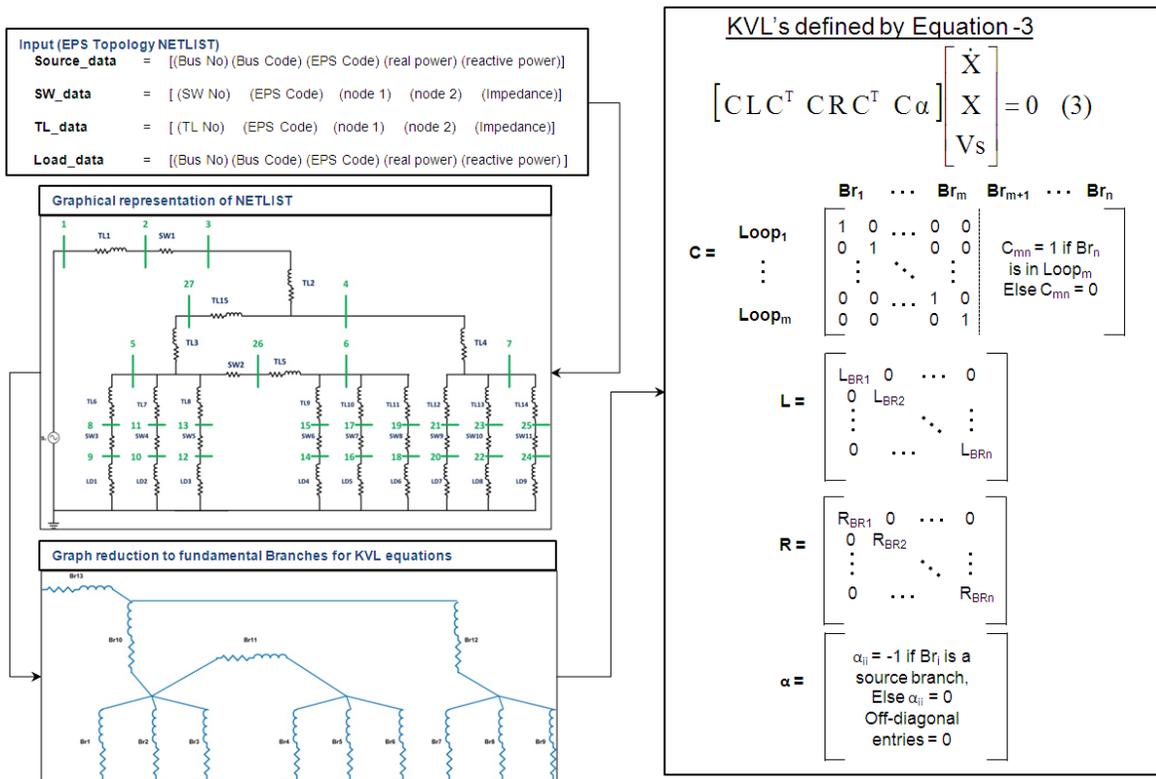


Figure 13: State Space Model of an EPS Topology

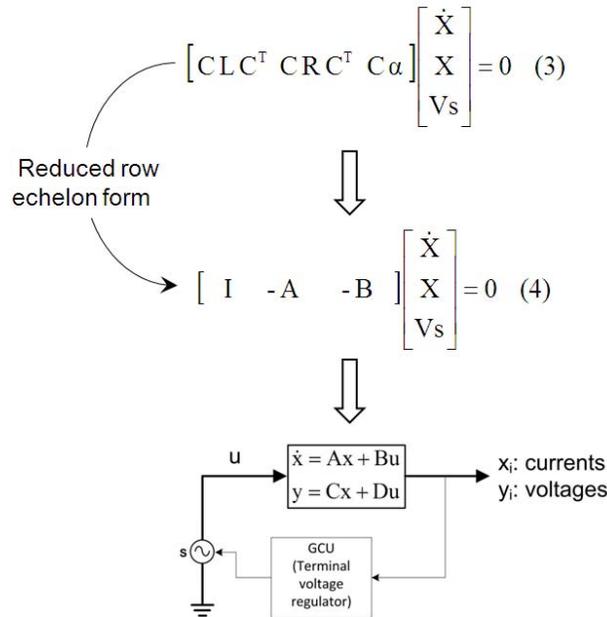


Figure 14: Netlist to State Space Formulation Process

3.8 Gas Turbine Engine Analysis

The EPS analysis described above considers in detail the electrical power distribution system that could be used in the primary challenge problem of aircraft design. Another key subsystem in aircraft design is the system that provides propulsion to the aircraft. Depending upon the aircraft mission, with currently available technologies, propulsion can involve rocket engines, internal combustion engines (that drive propellers), or gas turbine engines of three possible forms ---turbojets, a turbofan s and turboprops. The proposed military mission narrows the choices so that the primary motive power should most appropriately come from a gas turbine engine. With this in mind, a sub problem was started that involved the application of the META II methodology for analyzing design choices for various designs for a gas turbine engine.

A collaborative effort involving staff from UTRC, from Pratt & Whitney Aircraft, MIT, and IBM/Haifa Research was started to adapt the techniques of the META II design flow to an analysis of the Gas Turbine Engine. We were working “backwards” in many ways from detailed information on particular designs to obtain models at a higher level of abstraction than our starting points. We suspect that this will be the case in many other applications of the methodology --- the difficulty is in understanding the appropriate content for each abstraction layer and providing it from more detailed point solutions.

3.8.1 Level 1 - Gas Path Thermodynamics

Initial analysis focused on 1 dimensional gas path thermodynamics. This is because of the essentials of the behavior of a gas turbine engine are determined by a thermodynamic cycle called the “Brayton Cycle” and the various subsystem linkages between components necessary for executing this thermodynamic cycle. The gas turbine engine must first compress air and channel this air into a combustor. The combustor will then mix the air with fuel and burn the fuel, causing rapid heating and expansion of the gas. The energy of this gas is then transformed into thrust. This can be done directly, where the exhaust gas provides all the thrust, or it can be

done indirectly, where the exhaust gas drives a turbine which in turn drives a turbofan or turboprop. The difference between a turbofan and turboprop is that a fan is contained inside the engine nacelle, and a turboprop is not.

In this analysis, the number of compressors and turbines are variable, as are the types of linkages between turbines, compressors, and propulsors.

3.8.2 Level 0 – an intermediate step in analysis

Our first conjectural level was that Level 1 should enumerate the various possible combinations of these components – essentially fixing the topology for the system. It turned out that the modeling for this was difficult and begged for the creation of another abstraction layer. Thus “Level 0” was created which focuses on the source of propulsive thrust. To do this, a subsystem was created called “The Core” which consists of all compressors, combustors, and turbines grouped together into a single subsystem. (This is a common industry grouping of engine components). This simplification allowed some significant design choices to be made without incurring the computational complexity of more detailed models.

3.8.3 Abstraction Levels in the Gas Turbine Engine

For physical systems with high part count and modeling complexity it is often necessary to divide the design process into multiple layers defined primarily by the granularity of subsystem decomposition. This is particularly true for highly coupled multi-disciplinary problems such as gas turbine engine design.

In the design of abstraction layers a number of preliminary methodological guidelines were followed. The first was based on limitations of human interaction with any design tool and the second on the computational resources required to explore a given design space fully at any given layer of abstraction. Additionally it was found that an abstraction layer is only as useful as the number of lower level effects it expresses within itself. A good approach for abstraction layer design would therefore be at the intersection of all of these considerations.

When notionally designing the *level 0* and *level 1* abstraction layers for the engine problem, it was found that both the physical and structural models were much easier to create and interact with, provided the number of components in every subsystem was of the order of 10. Thus going down from the black box representation of the engine, every subsystem was divided into roughly 10 components in the layer below or less if the fundamental structure of the design so dictated. A good example of this would be a decomposition of for example utility systems at *level 0* to the electrical power, turbine cooling, lubrication, sensing, fuel, starting and control systems at *level 1*. In contrast a compressor subsystem naturally decomposes into 2 or 3 compressors.

The computational resources required within the design process also limit one in the design of abstraction layers. This limitation is coupled with the domains in which design is being carried out and the rules or constraints placed on architecture enumeration. Thus both the rules and the algorithm used for architecture enumeration must efficiently eliminate the vast majority of infeasible architectures. In addition the optimization and possibly simulation steps which aid in the selection of a set of best architectures must do so at a fidelity which useful but also computationally viable. This can drive one to design abstraction layers which minimize the amount of analysis that must be completed per reduction of overall design space.

The final point that is being broadly speaking applied to levels 0 and 1 or the gas turbine design problem is that lower abstraction layers must always be refinements of upper ones.

3.8.4 Optimization for the Gas Turbine Engine Analysis

To complete the gas turbine engine design task a number of different models were created that described different views of the propulsion system. This section refers primarily to those models that directly expressed the physics governing the problem. For the abstraction layer called *level 0* the physics was represented in an optimization model based in AMPL and JAVA. The primary function of this model was to automatically optimize different architectures outputted in the form of adjacency matrices from the Architecture Enumeration Engine. For *level 1* transient models based on component performance maps were created. Work is currently being done to provide an auto-generating capability to these level 1 models as well as converting them to optimization models.

The AMPL optimization takes as an input a given engine architecture and returns the parametric instantiation which has the lowest thrust specific fuel consumption (Fuel Flow (kg/s) / Thrust (N)) at takeoff conditions. At this level of abstraction the core was treated as one element due to the fact that the compressor subsystem, turbine subsystem and combustor subsystem would always be connected in the same order in order to achieve the “Brayton” thermodynamic cycle. The primary aspect of the architecture that was changed was the type and location of the “Propulsor” element whose primary role was to provide the majority of the thrust.

All the governing equations used for the optimization were based on the Level1 Matlab transient models. The primary difference was that values such as efficiency, pressure ratio and mass flow through turbomachinery components were treated as parameters at level 0 whereas at level 1 they were calculated from component performance maps. An additional difference was that the work done by the turbines was constrained to be equal to that required by the compressors and propulsor at level 0 whereas this was not the case at level 1 giving rise to spool dynamics.

Another difference was that gas constants were defined as variables which were related to temperature via the polynomial expressions in [25]. The relationships used for propellers were taken from [26].

3.8.5 Components

Propulsor

Propulsor Type	Functional Description
Single_Fan_(DTC)	Primary provider of thrust as well as first stage for compression for core (DTC: ducted to core indicating a gas connection)
Single_Fan_(nDTC)	Primary provider of thrust (nDTC: not ducted to core indicating that there is no gas connection)
Multiple_Fans_(DTC)	Primary provider of thrust as well as first stage for compression for core (DTC: ducted to core indicating a gas connection)
Multiple_Fans_(nDTC)	Primary provider of thrust (nDTC: not ducted to core indicating that there is no gas connection)
Single_Front_Prop	Primary provider of thrust as well as first stage for compression for core (Front: gas connection to core)
Single_Rear_Prop	Primary provider of thrust (Rear: no gas connection to core)
Multiple_Front_Props	Primary provider of thrust as well as first stage for compression for core (Front: gas connection to core)
Multiple_Rear_Props	Primary provider of thrust (Rear: no gas connection to core)

Diffuser

Diffuser Type	Functional Description
---------------	------------------------

Fan_Diffuser	Recovers total pressure/temperature of flow Slows down flow for entry to Fan
Core_Diffuser	Recovers total pressure/temperature of flow Slows down flow for entry to Core

Nozzle

Nozzle Type	Functional Description
Fan_Nozzle	Expands fan exit flow to ambient (constrained to operate without shocks)
Core_Nozzle	Expands core exit flow (pressure ratio constrained to operate with shocks)

Core

Core Type	Functional Description
Core	Takes fuel and air and converts it to thrust and shaft power

3.8.6 Assumptions and Background

- Propulsor element is always driven by core.
- Due to the fact that all the components used at level 0 were component subsystems rather than the components themselves, the multiplicity of each was assumed to be 0 to 1.
- Distributed propulsion systems with 1 core driving multiple fans or props were taken into account by creating subsystem blocks such as “Multiple_Fans_(DTC)” or “Multiple_Props_Front”. The reasons for this were rooted in assembling the AMPL code templates into a single optimization code. The bypass ratio bounds within each of these components were different depending on the whether they were abstractions of distributed propulsion systems “Multiple”, or the legacy “Single” case.
- Mechanical transfer efficiencies between the Core and the Propulsor for distributed propulsion systems were assumed to be considerably lower than their conventional counterparts due to complex gearing and transmission systems.
- Arbitrary linear relationships were used to compute component cost based on performance.

3.8.7 Variables, Parameters and Constraints

Most bounds on variables in the optimization code have a 1 to 1 mapping with the assumptions and guarantees expressed in the contracts written in TinyCSL for the above engine components. The primary purpose of the level was for preliminary analysis of distributed propulsion systems. Thus all of the parameters that were assumed here are based on expected results from Level 1 analysis and were also chosen in a manner consistent with the Level 0 TinyCSL engine contracts.

3.8.8 Objective

The pressure ratios, temperatures, bypass ratio and core mass flow (a global variable) were varied via the “ipopt” solver in AMPL until the desired thrust was provided at the lowest possible fuel consumption.

3.8.9 Auto-generation from Architecture Enumeration Engine

The AMPL optimization model was highly simplistic in order to allow relatively easy and robust auto-generation of optimization code from adjacency matrices. Each component was represented via AMPL model and data templates. The model templates contained all the thermodynamic relationships and constraints required to compute outputs from input. The data templates served to provide automatic “initial guessing” of all state variables as well as parameter assignments.

- Every variable within a model template that was an input was noted by “_%”.
- Every variable that was internal to the component model or data templates for or an output was noted by “_\$”.

We wrote Java code which automatically replaced these by indices from the adjacency matrix from AEE and assembled them into a single optimization code. In further work these replacements will be multi-domain rather than only gas flow. Certain rules were coded into Java which also took into account when a component was to be connected to the atmosphere. In the case of the Fan_Diffuser and all the props the input was hard coded to be from the “Atmosphere” template. For the case of the Core_Diffuser there was a rule in Java that connected it to atmosphere only when there was no input from a propeller or propellers in front of it.

3.8.10 Results

The following are 2 examples from the twelve generated architectures at level 0 (Figure 15).

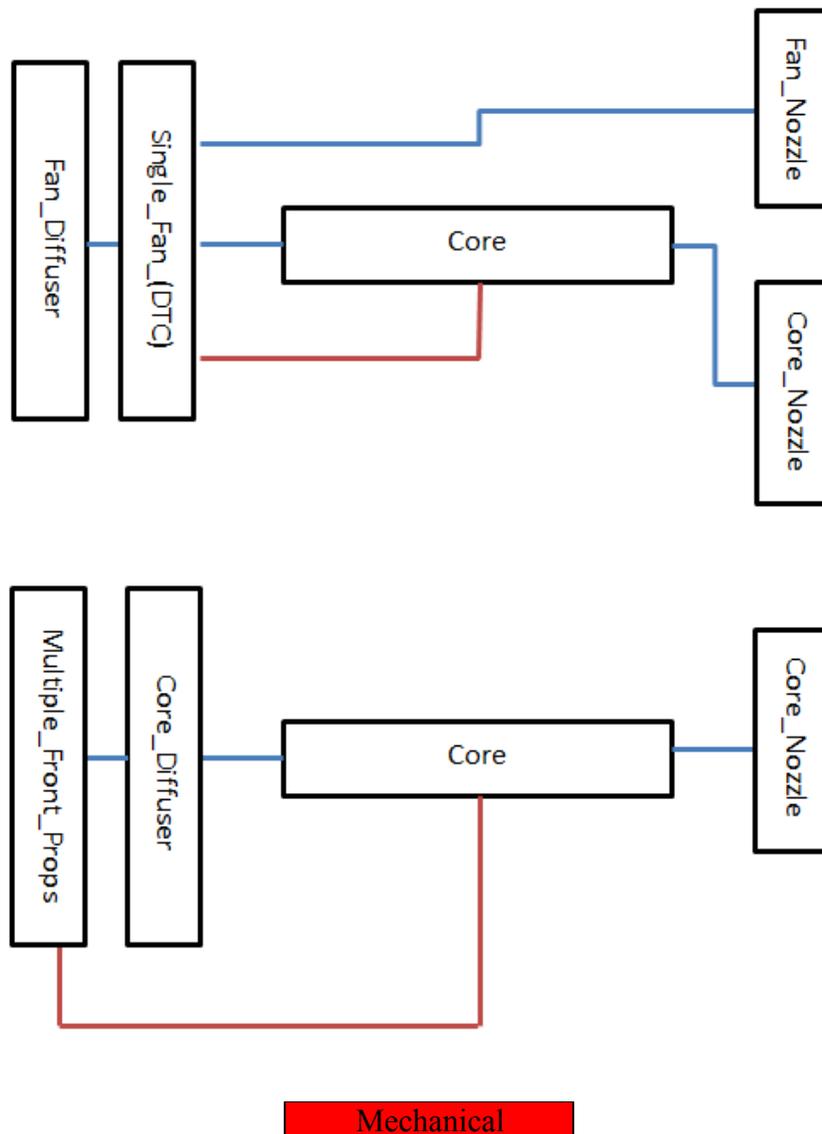


Figure 15: Level 0 Engine Architecture

For 4 of the 12 generated architectures an additional bypass ratio constraint imposed via the “Embedded” component which indicated that the engine must be placed within the engine fuselage.

3.8.11 Remarks

Not all the constraints were explicitly carried over from TinyCSL. The temperature range constraints that were imposed in the TinyCSL code were only satisfied due to the choice of high efficiencies in the components being used in optimization.

The constraints that were omitted were imposed differently. For example there were constraints in AMPL that made the inlet temperature of the compressor always smaller than the exit. In addition there was a constraint that stipulated that the Propulsor exit pressure was greater than the atmospheric.

Ideally these two approaches should completely converge with a 1-1 mapping between optimization constraints in any environment and parametric design contracts.

3.8.12 Level 1 Matlab Simulation and Optimization

The primary purpose of the Matlab Level 1 models is to provide the ability to fly flight profiles for various types of engines and in so doing allow the calculation of performance throughout the operating envelope. Work is currently being done to link these directly to architecture enumeration and create optimization models based on the transient ones.

It is expected that similarly to the level 0 AMPL-JAVA optimization, Level 1 Matlab optimization and simulation models will take as an input an adjacency matrix and aircraft flight profile and will output an estimate of total fuel consumption.

The automatic passing of mission information has not yet been included for any of the models but is currently an active area of work.

3.9 Software and mapping modeling

In this section, we present our decision diagram based complexity metric in an intuitive way. We will focus on answering two questions: how this approach can be applied to different cases and why the decision diagram reflects the complexity of that case. Specifically, we will intuitively show and explain how the approach models and reveals the complexity of state transitions, scheduling, and mapping.

3.9.1 Decision-diagram-based complexity of software models

From past research works on Binary Decision Diagrams (BDD), we know that any logic function can be represented using BDDs (we assume all the BDD or Decision Diagrams (DD) mentioned in this report are variable-ordered.) which reflects the complexity of the logic function and has numerous good properties. Inspired by this, we use a more general DD to represent the complexity of models. If a model can be represented by a discrete function, it can be also represented by a Decision Diagram (DD). The overall flow is shown as follows:

Model → *Discrete Transition Function* → *Decision Diagram*

Since Discrete Functions and Decision Diagrams (DD) are equivalent, the main problem is how to convert a model into DTF or DD that reflects the complexity.

Complexity of Internal Logics. One basic complexity of a model is the internal logics that transform data or signals. In order to measure the complexity, we represent this transformation in DD, which would reveal how the inputs and outputs are related. Simple relations would result in simple DD while complex relation would cost more nodes, edges, and paths in the DD representation.

Figure 16 shows a finite state machine with 3 states and 3 transitions. It can be represented as discrete function (Figure 17) by adding the current states as the inputs and the next states as the outputs. The function thus represents all the possible transition of the FSM. By representing the discrete function in this way, the complexity of the model is exposed. Because complex transitions would result in larger DD and similar transitions would be ‘compressed’ in DD.

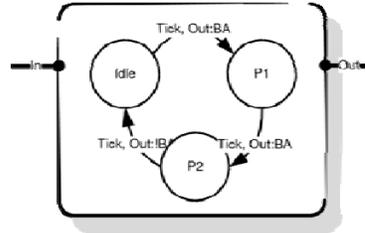


Figure 16: Model 1, a Finite State Machine

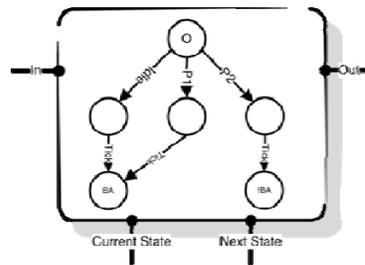


Figure 17: The Decision Diagram Corresponding to Model 1

Complexity of Scheduling. Besides internal logics, scheduling is another source of complexity. In our model, scheduling is considered as a special function that takes the states of the system as input and outputs the next process to be scheduled. It is usually combined with the DD of other components (functions) in the discrete function. As shown in Figure 19, a data flow model (Figure 18) with a scheduler can be represented as a DD composed of scheduler DD and component DDs.

In this example, suppose we have three components: a controller feeds two embedded programs. The scheduler executes the controller if no control signal has been generated, otherwise it executes either EP1 or EP2. This scheduling process can be represented as a function by introducing a random input variable (scheduling input as shown in Figure 19) to eliminate the non-determinism. And like BDD, we enumerate every possible scheduling by enumerating the scheduling input to represent the complexity of the non-determinism.

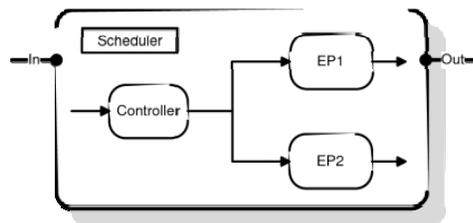


Figure 18: Model 2, a Dataflow Model with Non-Deterministic Scheduler

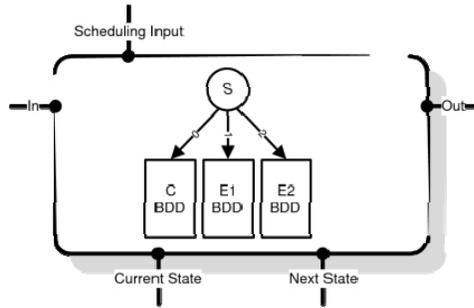


Figure 19: Decision Diagram Corresponding to Model 2

Complexity of Composition. If a system is composed of more than one sub-system of which complexities are known, the complexity of the system can be calculated by composing the corresponding Decision Diagrams. As shown in Figure 20, input variable is replaced by an output DD if this input is determined by that output.

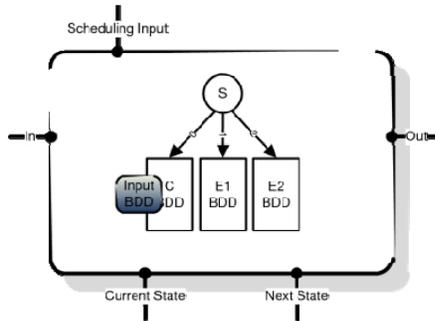


Figure 20: DD Composition of Model 1 and Model 2. The Scheduler Inputs Replaced Outputs of State Machine

Complexity of Mapping. Mapping is the bridge that connects the functional model and architectural model. Implementing the same functional model on different architectures would result in different complexities. To model this effect, the mapping is seen in two ways:

- A couple of functional components are replaced by a architectural component with more implementation details;
- The scheduling of functional components is constrained by the resources on the architecture.

Figure 21 shows a functional model of a ring network with four functions connected by four FIFO channels and its DD, which is composed of all the component DDs and scheduler DD.

- Suppose these four FIFO channels are mapped to a bus with adaptors as shown in Figure 22. The necessary changes in the view of complexity are shown in Figure 23. Figure 21: the DDs of FIFO channels are replaced by the DDs of the bus with adaptors. The DD of scheduler is updated with the bus scheduling.
- Suppose the four functions are mapped onto four tasks respectively and these four tasks are allocated onto two processors as shown in Figure 24. In the view of complexity, we

need to update the DD of scheduler with constraints given by the shared processors as shown in Figure 25.

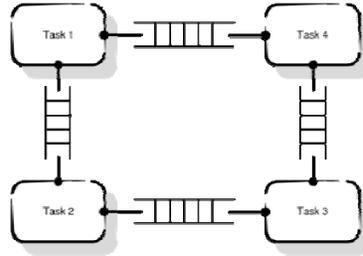


Figure 21: Model 3, a Functional Model of a Ring Network with Four Functions connected by FIFO channels

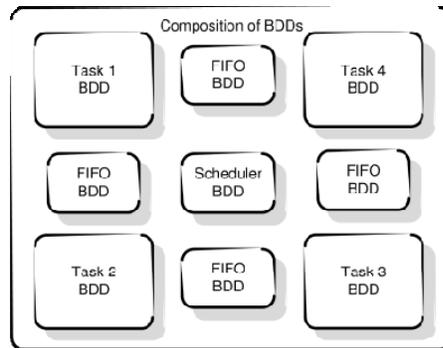


Figure 22: DD Representation of Model 3, Which is Composed of All DDs of the Components

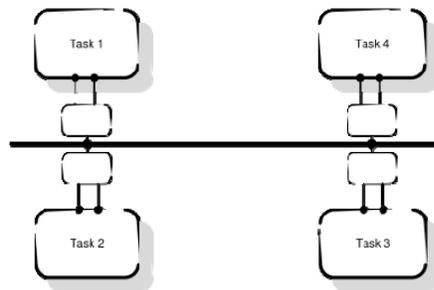


Figure 23: Mapped Model 3: FIFO Channels are Mapped Onto a Bus with Adapters

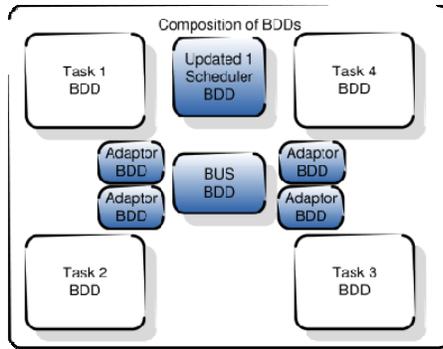


Figure 24: Decision Diagram Representation of Mapped Model 3

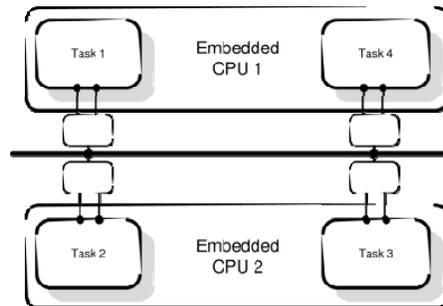


Figure 25: Mapped Model 3: Functions are Mapped Onto Tasks on Two Embedded Processors

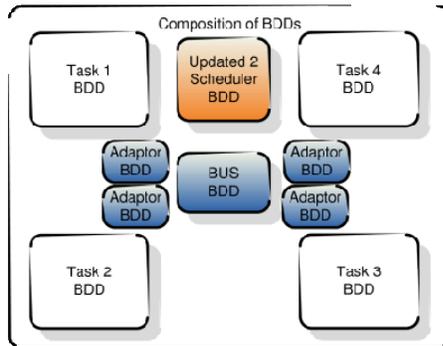


Figure 26: Decision Diagram Representation of Mapped Model 3

3.10 Abstraction layers design

The design paradigm that we use requires the definition of abstraction layers. This means deciding what the order of design decisions is and what the types of models that are used at each layer to make those decisions are. There is currently a lack of computer aid to define abstraction layers for a given application domain. In this section we show how contracts are used in the context of creating abstractions of components. We should through an example that meaningful abstractions are the ones that serve the purpose of making decisions based on accurate metrics.

Thus, the abstraction process needs to provide the user with feedback on the accuracy of metrics (or conversely a measure of the information loss) that will be used in design decisions. We think that rather than automatically derive a design flow, a system engineer should be provided with a variety of tools to judge the quality of an abstraction which can only be done in reference to the metrics of interest. For example, when deciding on the architecture of a system to minimize weight, the abstract models for the components need to provide an accurate weight metric, while other metrics (e.g. temperatures) might be inaccurate or even absent.

3.11 General criteria for the selection of abstraction layers

We identify two axes along which abstractions can be defined (see Figure 27). One axis is *scope*. Scope abstraction consists in grouping components together to form abstract modules (or components). For example, one may look at the interconnection of diodes, capacitors and resistors that transform a sinusoidal waveform into a constant voltage source (DC), or it can group these components together into a transformer rectifier unit (TRU). The other axis is *fidelity*. The same component can be modeled at different level of fidelity. For example, one may loop at an actual voltage and current at the output of a generator, or it can simply look at the presence or absence of power flows. However, we are typically more interested in the general concept of *time scale* while we consider hiding details still as a scope abstraction. Time scale abstraction is very common in industry where models are typically organized into steady state and dynamical models. In general, one can have a range of time scales that are considered where more detailed models take into account faster time scales.

The current approach to the design of system is center around reference architectures that are taken as starting point for any design. In terms of scope, reference architecture is typically very detailed. The design process then proceeds by refining models and adding details until the reference architecture is finally implemented. Changes are made to the reference architecture to accommodate requirements. The complexity of making such changes is high because of the large number of details that are already present in the reference architecture.

The design flow we advocate follows the joint refinement of the scope and time scales for a system. For example, consider the case of an electric power system. The abstraction process would hide the details of the actual topology of the system and only present a view which contains power sources (generators, batteries and APUs), power sinks (i.e. the loads) and connections among them. Each component would have properties such as voltage levels, power requirements and reliability. This view would be refined into a more detailed topology where each connection becomes an actual path. The properties of the connection (i.e. maximum voltage drop or reliability) must be maintained in the refinement process (vertical contracts) so that any analysis done at the abstract level is maintained true after refinement. The fidelity of the models is also increased during the refinement process going from a simple logic view (presence/absence of power), to steady state models, to transient models.

The number of abstraction layers should be minimized. In fact, having many abstraction layers does not help in terms of the optimality of the final architecture because abstract models are in general inaccurate and decisions made at high abstraction layers might rule out promising part of the design space. However, having very few abstraction layers might make design spaces too large to be explored efficiently.

- General abstraction principles: scope and time scale
- Minimize the number of abstraction layers subject to computational complexity

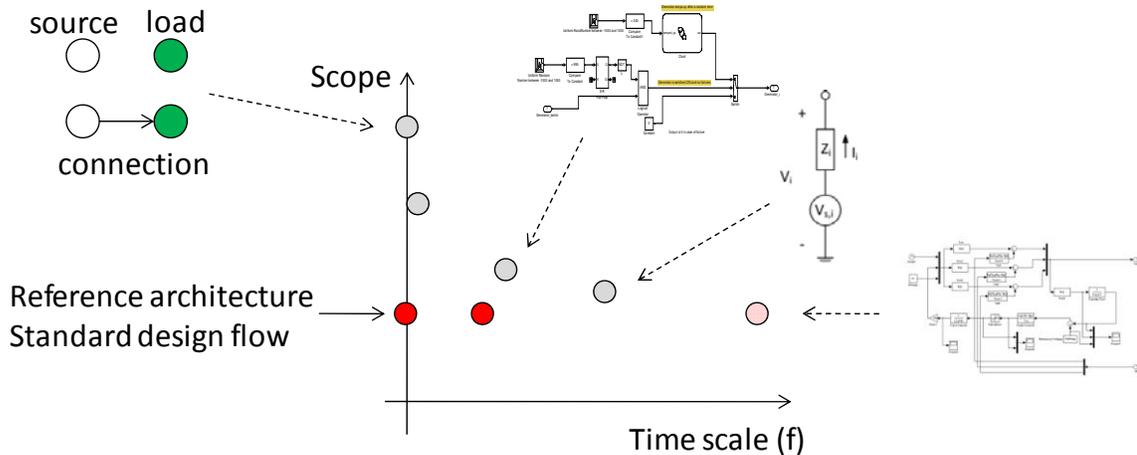


Figure 27: General Principles for Abstraction Layer Design

3.12 Definition and computation of abstract models

In this section, we formalize a procedure for abstraction of contracts in the context of steady state models. In the context of contract based design, abstraction is defined as follows. Consider a contract $C(V(U, X), A, G)$ where V is a set of variables associated with the model. The set of variables is divided into uncontrollable variables U (provided by the environment) and controllable variables X determined by the model associated with the contract. The assumption $A \subseteq D_U$ is a subset of the set of all possible values of the uncontrollable variables U . The guarantees $G \subseteq D_U \times D_X$ is a relation between the uncontrollable variables and the controllable ones.

Consider a contract C and a contract C' . Then C' is an abstraction of C if and only if $V' \subseteq V$, $A' \subseteq A$ and $G \subseteq G'$ (where we have omitted projections operators to avoid further complexity in the definition). In simple words, an abstraction is a quantity which *assumes less and guarantees more* than the original system.

We now formalize an automated procedure to generate an abstraction of a detailed model in the form of a contract C . In this procedure, the user selects some subset of the variables V , e.g., (X', U') . Then, the following procedure will generate the set of assumptions A' and the set of guarantees G' for the abstraction C' as follows.

1. **Inputs:** $C(V(U, X), A, G), X', U'$
2. Initialize: $A' = \emptyset, G' = \emptyset$.
3. For every $u \in U \setminus U'$, generate data points $u_1, \dots, u_N \in A$ using the detailed model C .

4. For every $x' \in X'$, compute the maximum value $g_{max} := \max_i x'(u', u_i)$ and the minimum value $g_{min} := \min_i x'(u', u_i)$ and add the constraints $x' \leq g_{\{max\}}$ and $x' \geq g_{\{min\}}$ to the set G' .
5. For every $u' \subseteq U'$, and such that there is an $a(u') \subseteq A$, add a to A' .

In other words, the above procedure uses the detailed model to generate samples of values of the controllable variables x' using for various values of the variables which the user decides to remove, and computes the maximum and minimum of these values. Then the guarantees are that the variables x' so computed will take values in the interval $g_{\{min\}} \leq x' \leq g_{\{max\}}$.

In order to obtain a further compact approximation of these guarantees, we can determine the best fit polynomial curves which approximate these minimum and maximum values. This helps us determine the *quality of the abstraction*. We adopt the following procedure:

Let $g_{\{max\}} \approx f(\theta, u')$ denote a polynomial parameterized by the coefficients θ . We solve the least squares problem: $\min_{\theta} \sum \left(g_{\{max\}}(u') - f(\theta, u') \right)^2$, to obtain the vector of parameters θ^* . Additionally, this procedure also provides us with the error resulting out of the approximation. This error is simply the value of the objective function at its minimum. This error can be used to compare between different abstractions based on the number of coefficients chosen in the approximation.

In the next section, we demonstrate the application of this methodology to a pump. We began with a very detailed model of a pump involving several variables. The abstract model considers only a few (two in this case) out of all controllable and uncontrollable variables. We then apply the above procedure to generate the maximum and minimum values for the controllable variables. Then, we determine the best fit planes (linear fit) to determine approximations to the set of guarantees in the abstract model.

3.13 Application to thermal management

Figure 28 shows the main concept behind abstraction as well as an application to the steady state model of a pump. The abstraction process removes some uncontrollable and some controllable variables from a contract. As a consequence, each value assigned to the uncontrollable variables of the abstract contract will correspond to a set of possible values for its controllable ones.

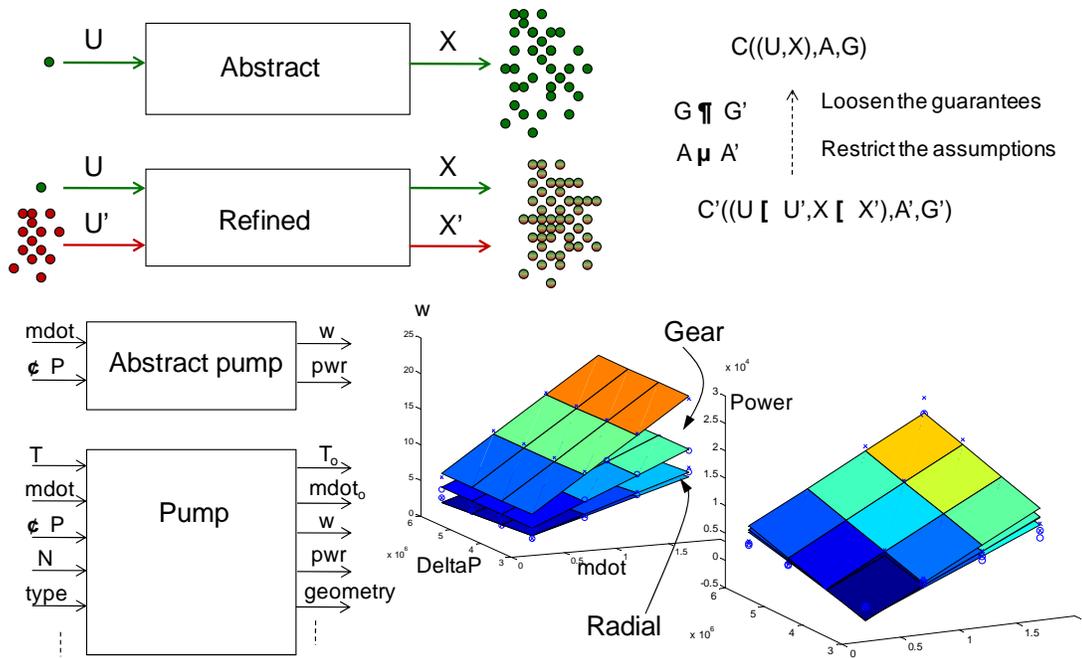


Figure 28: Abstraction of a Pump Component

Consider a pump which is an key component of a thermal management system. The uncontrollable variables include the temperature T , the fuel from rate \dot{m} and the pressure P of the inlet, the assigned pressure across the pump ΔP , the speed N , the type of the pump (e.g. radial or gear) and other parameters such as the properties of the fuel. The controllable variables include the temperature, fuel flow and pressure of the outlet, the weight of the pump w , the power consume by the pump pwr and the detailed geometry of the pump. A desirable model for design space exploration only exposes the weight and the power consumption of the pump as a function of the flow rate and pressure across the pump. Figure 28 shows the result of the abstraction for both the weight and the power consumption. We have computed the upper and lower bounds for these metrics as interpolating hyper planes. We notice two interesting facts for the weight metrics. The weight metrics cannot be abstracted into a usable function for a gear pump (the bounds show a wide range) while the abstraction is fairly good for a radial pump. If the pump type is also abstracted, then the resulting contract shows two disjoint intervals for the weight metrics which will make design space exploration hard (since the weight is not a convex set anymore). The power absorbed by a pump is instead a usable metrics even at high abstraction levels (upper and lower bounds are close).

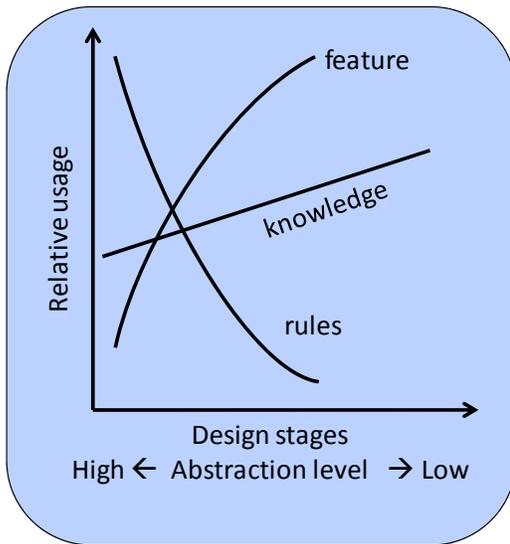
3.14 Manufacturing Modeling and Integration

The objective of this task is to develop approaches to model the manufacturing models that can be integrated with design at various abstraction levels with the goal of significantly reducing product development time for complex cyber-physical system. Corresponding to the platform-based correct by construction design flow and the filter-based enumeration, a multi-level manufacturing modeling framework is adopted.

As shown in Figure 29, knowledge-based design and manufacturing integration is represented for all the abstraction levels. On the early design stage at high abstraction level, no features are unknown. Designs are based on rules and knowledge. For example, the max

temperature should not exceed a material limit which may be 2300F. To integrate manufacturing and cost consideration with design process, manufacturing rules can be written as constraints for design evaluation. Some parametric cost estimation relationships can also be established using historical data.

During the mid level design stages, the usage of rules is gradually reduced and more feature and knowledge are used. For example, feature information and knowledge database of a component or a subsystem can be obtained from database for same or similar components. At detailed design stage, feature based mfg and knowledge based design are dominated because all the parts have specified geometry and function description. Dewhurst DFM and DFA are the best available software for this level mfg cost estimation when detailed CAD and mfg process model is available.



1. Rule-based approach for higher level abstraction.
2. When design progresses to low level abstractions, more details become available. Feature-based approach can be implemented.
3. Knowledge-based transfer, translate, transform can be used for all design stages. The frequency steadily increase with the complexity increases.

Figure 29: Summary of Manufacturing Information Usage

3.14.1 Manufacturing Cost Models

Based on the approach outlined in the previous section, we developed and implemented manufacturing rules, parametric cost models for engine and some components for Level 0 and Level 1 abstraction, and featured-based cost model for some complex engine components.

Manufacturing knowledge and rule implementation

During early design stages, there will be many concepts generated by design enumeration. Design evaluation will use rules to choose feasible ones. At this stage, we implemented manufacturing rules as constraints. It was realized that there can be many manufacturing and material rules written. For the RLC circuit demonstration, we used rules of temperature capability of the wire and the unit cost difference for high temperature capability wires. The length of the wire needed to implement a particular design was also used in cost estimation.

For the engine design, material capability rules were used. The temperature capabilities of a few common materials used in aerospace engine such as aluminum, titanium, nickel alloy and power nickel alloy were contracted as manufacturing rules during design evaluation.

Parametric Cost Estimation for Jet Engine

Parametric cost models for abstraction Level 0 and Level 1 were developed for the engine challenge problem as depicted in Figure 30

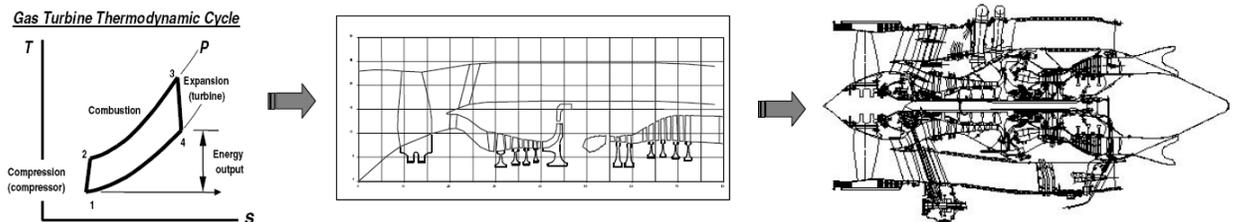


Figure 30: Parametric Models

Level 0: mission level

At Level 0, engine mission profile or engine specification is known. Cost model was established as a function of engine specifications including SFC, thrust, mach number, weight, and altitude. The model was built using regression analysis based on historical engine cost data from the public domain.

Considering all engines, the development cost is:

$$\text{Dev_Cost} = 0.00012 * T + 1163 * \text{TSFC} + 0.31 * W_t - 731.54$$

The above cost model includes all historical data. For new engine development, “newly-designed” engines data may be more suitable. The development cost for new engine is:

$$\text{Dev_Cost} = 0.11 * T + 474.71 * \text{TSFC} - 0.395 * W_t - 138.14$$

Where,

T = Thrust (lbs)
 TSFC = Thrust Specific Fuel Consumption (1/hr)
 Wt = dry engine weight (lbs)
 Dev_Cost = Development Cost in 2001 US\$ (millions)

Level 1 Engine model

In the second level, thermodynamic data of the engine cycle become available. These parameters include pressure, temperature, pressure ratio, efficiency, mass flow rate and area. Instead of using purely empirical data, the cost has been constructed using physics-based model with limited cost data from public domain.

EngineDevCost = (massflowrate/massflowrate_ref)^a * [Cost-fan + Cost-casing + Cost-augment-nozzle_ref] + Cost-core_efficiency + Cost_Machnumber + Cost_TurbineInletTemperature

Where,

EngineDevCost: engine development cost, in 2001 US \$million.

Cost-fan = Cost-fan_ref * (BPR/BPR_ref)^b;

Cost-casing = Cost-casing_ref * (OPR/OPR_ref)^c;

Cost_Machnumber = d * MachNumber;

Cost_TurbineInlettemperature = e * TurbineInletTemperature;

When 0.9 < both efficiencies < 1:

Cost-core_efficiency = CostCompressor_ref * 1 / (1 - EfficiencyCompressor)^2 + CostTurbine_ref * 1 / (1 - EfficiencyTurbine)^2 + Cost-burner;

When both compressor and turbine efficiencies < 0.9:

Cost-core_efficiency = 300.

The value of the baseline_ref variables and coefficients a, b, c, d, e are calibrated with public data:

a=1.5; b=2; c=1.5, d= 60; e= 0.08.

Massflowrate_ref = 3000 lbs/sec, BPR_ref = 8, OPR_ref = 40, Cost-burner = 0.8, Cost-Compressor_ref = 1.5, CostTurbine_ref = 1.5, Cost-fan_ref = 150, Cost-casing_ref = 100, Cost-augment-nozzle_ref = 80.

Cost model for other engine components

Cost models were also developed for a number of engine components including high compressor, low compressor, combustor, diffuser, fan, cod and hot nozzles. These models are coded in Matlab.

These codes have been integrated with engine design optimization codes in AMPL delivered to DARPA.

3.14.2 Feature-based cost model

At more detailed abstraction levels, feature-based models are appropriate.

Gearbox cost model

A cost model template for a simple gearbox was also developed under Matlab/Simulink (Figure 31). This model uses standard gear design parameters as inputs to calculate cost. The idea is to link this with design tools so cost can be estimated automatically.

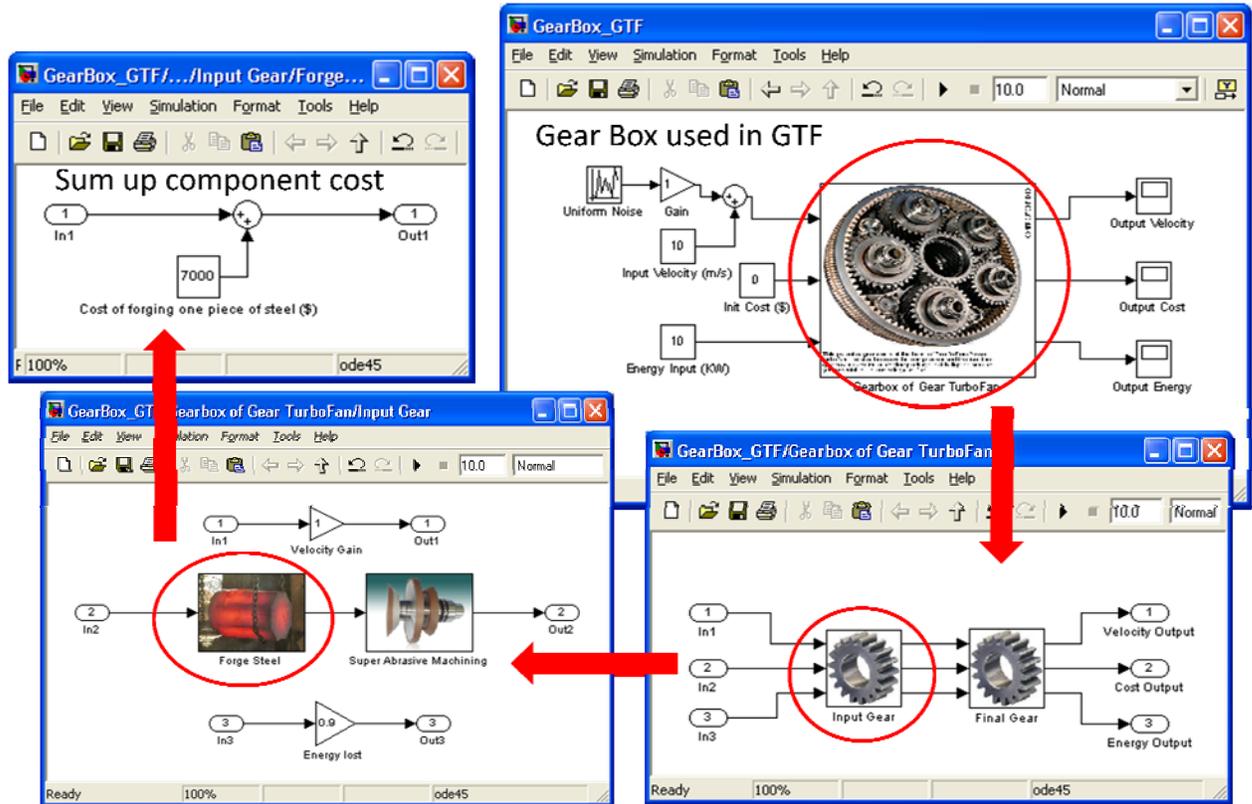


Figure 31: Matlab/Simulink Gear Model

Cost model for the Integrally-bladed rotor (IBR) or Blisk

This is done with a commercial software, Design for Assembly (DFA) and DFM by Boothroyd Dewhurst Inc. The DFA analyzes assembly difficulties and DFM Concurrent Costing analyzes item manufacturing costs. The software estimates cost based on part volume, area, and other geometry data. Data can be transferred link to a CAD model through the SolidView viewer to transfer geometry data into your analysis before you enter dimensions on the first screen. If necessary, geometry can also be sketched using the geometry calculator (Figure 32).

The machine setups for a number of machining processes are included in the Process Chart with the operations listed. The three libraries - material, machine, and operation, are fully editable. Manufacturing cost estimation involves material, process, machine tool, setup, and operation reject. At the top level, the material and principal process is determined. Model is built for an engine disk.

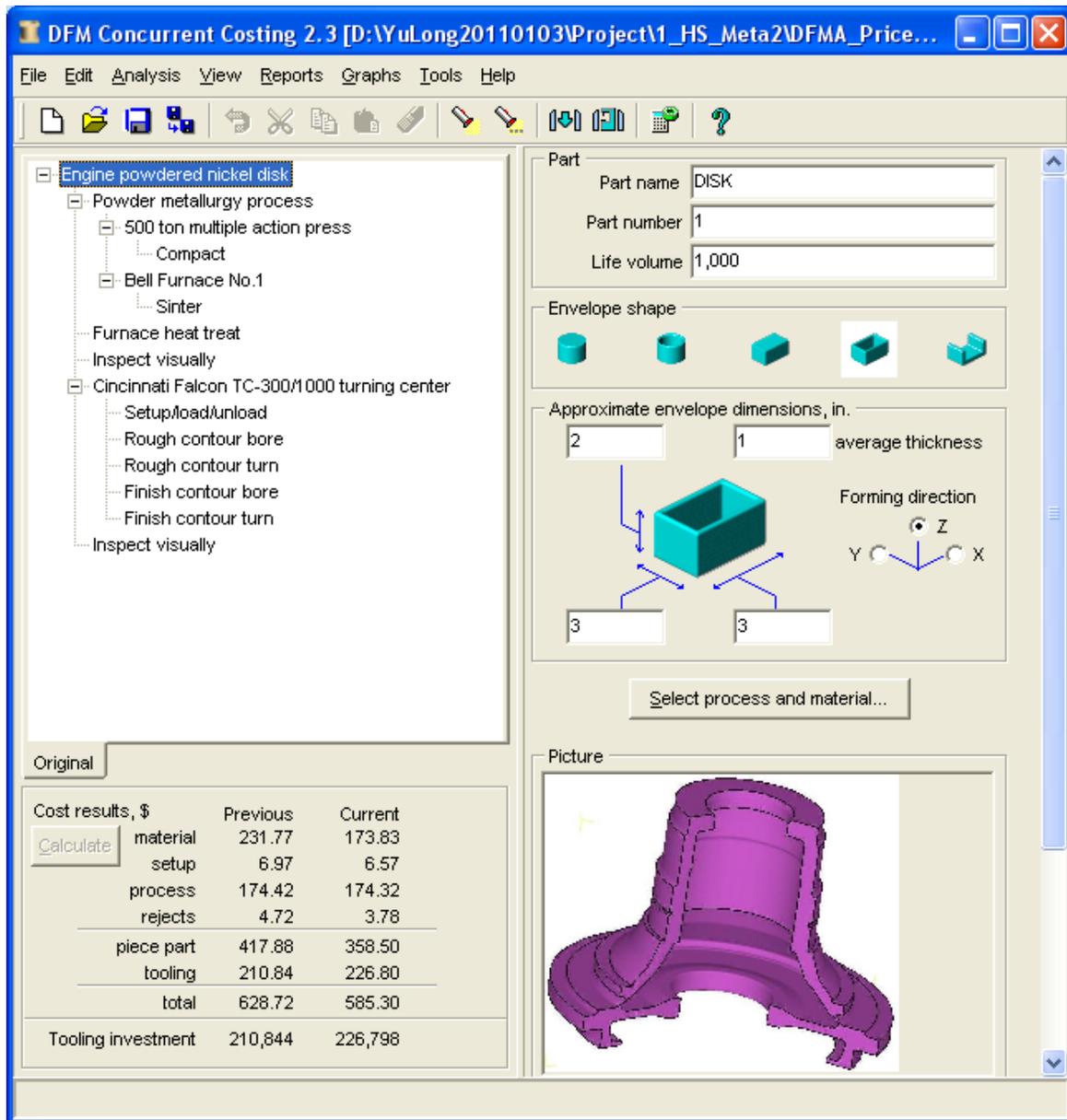


Figure 32: IBR Modeling

3.14.3 Model Integration

Under this task, manufacturing cost is modeled using both codes developed under this program in format of Matlab, and C+ codes and commercial software DFM. Evaluation of PriceH for parametric cost estimation is also performed. The question is how to integrate these models/software with design tools such as Unigraphics to do integrated design and cost estimation. A preliminary model integration demonstration was investigated with Model Center. Model Center® is a visual environment for process integration. With Model Center, an engineering process can be quickly created to perform complex design exploration to find the best design. A simple design under Unigraphics is linked with cost models such as PriceH and public codes written in Matlab and Excel. More work is needed to demonstrate the capability of this integration.

3.15 Architecture Enumeration and Analysis

The META Design Flow is a form of Platform-Based Design in which the flow of product-design decision making proceeds down through a set of increasingly detailed platforms, i.e., layers of abstraction. Each abstraction layer is defined to include the appropriate architectural technology options and design rules to construct all possible relevant architectures at the corresponding level of fidelity, among which pre-designated design decisions must be made. Figure 33 shows the META Design Flow.

At each abstraction layer, the META Design Flow uses Architecture Enumeration and Evaluation (AEE) to rapidly identify all feasible architectures, and uses optimization to find the best architectures, each with its best parametric settings. Only the best architectures from each abstraction layer are brought down to the next layer for more detailed enumeration, evaluation and optimization. Since each successive layer is more detailed and more computationally complex than those above, the more abstract layers above serve to vet those architectures that will be analyzed below. **Error! Reference source not found.** shows how the META Design Flow uses AEE and Optimization to find the best architectures at each abstraction layer.

The META Prototype Tool Chain enables the META Design Flow and is designed to be broadly applicable, flexible, scalable and extensible. It consists of various elements interconnected by a language intended to serve as a Knowledge Interchange Format until a suitably designed META language is implemented. The elements are a repository of abstraction-layer device and flow libraries and design rules, an AEE tool, an Optimization tool, a Verification tool, and a Visualization tool.

The following sections of this report describe the META Design Flow, AEE, Optimization, and the META Prototype Tool Chain.

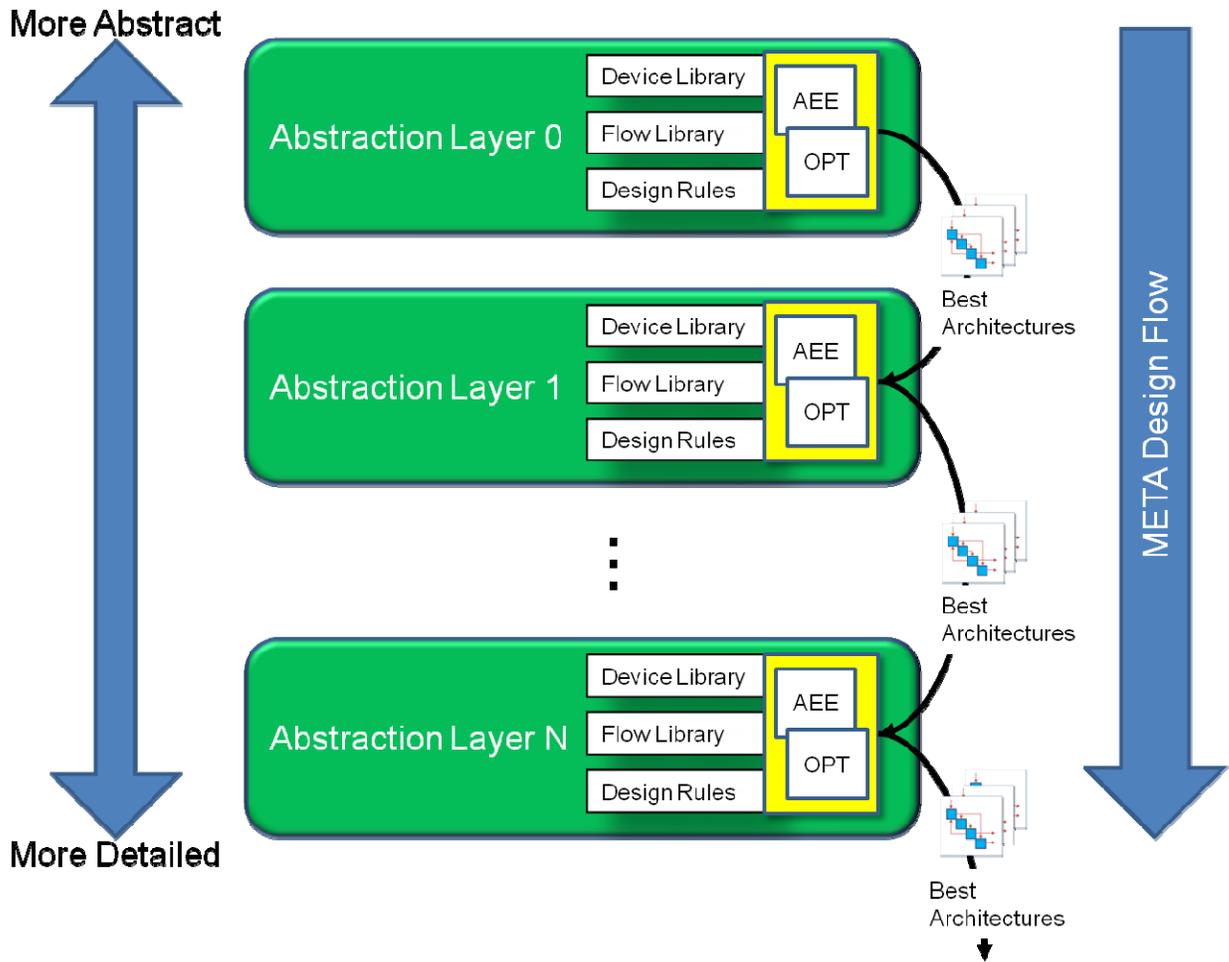


Figure 33: META Design Flow

3.15.1 The META Design Flow

The Design Flow proceeds down through a set of increasingly detailed abstraction layers. Each abstraction layer is associated with specific design decisions which are best made at that level of abstraction. Consequently, each abstraction layer is defined to include the appropriate architectural technology options and design rules necessary to consider all possible relevant architectures to make the design decisions associated with that abstraction level.

The best architectures brought down from one abstraction layer to the next embody all of the design decisions made at higher layers. Bringing more than one architecture down to the next layer constitutes a level of indecision, due to the inaccuracy inherent in abstraction. In an abstracted model, information is omitted, so while within the abstraction one architecture may appear to be better than all others, it is not clear that with the benefit of some of the omitted information, that one architecture would still appear to be best.

In addition, there is value in bringing down a diverse set of best architectures, rather than architectures which are only slightly different from one another in terms of design variable values. Diverse architectures are more likely to involve different types of omitted information, so they introduce more robustness to the next abstraction layer.

3.15.2 Architecture Enumeration and Evaluation (AEE)

AEE is a process for rapid Design-Space Exploration (DSE). AEE finds ALL feasible architectures within a design space, based on a set of design rules. At each abstraction layer, AEE uses design rules to rapidly identify the sparse set of feasible architectures from within the combinatorically large design space at that layer, consisting of all the layer's available technology options and all the ways in which they can be interconnected.

A. Data Inputs

This section describes the data inputs to AEE. The Appendix describes the Design Rule Specification Language in more depth, with a full language definition in BNF notation, an explanation of the syntax, and examples.

An architecture is represented as a set of devices and their interconnecting flows. Both devices and flows can represent technology options. Device types and flow types can be defined in terms of fixed parameters. Device and flow instances, within an architecture, can be defined in terms of variables which are computed during enumeration. Table 5 shows a Device table for the Engine Challenge Problem (Abstraction Layer 0); note that the rows correspond to device types and the columns include parameter values and an indication of which variables exist for instances of specific device types (none for this abstraction layer). Similarly, Table 6 shows a Flow table for the Engine Challenge Problem (Abstraction Layer 0) with similar structure, and eight types of flows.

Design rules can be expressed in many different forms. The forms of design rules that lend themselves best to AEE include:

- Required input flows to devices & required output flows from devices (e.g., each Single_Fan(Ducted to Core) requires DiffusedAir and ShaftPower as inputs) see Table 7
- The number of instances of particular device or flow types (e.g., # of Load1 = 1) see Table 6: the min and max parameters = 1 for Device type Load1 constitute this rule
- Parametric values associated with particular device or flow types (e.g., the bypass ratio of an engine is bounded by the minimum of each device type's maximum bypass ratio (maxBPR) parameter. The device maxBPR parameter (see Table 5 Table 5: AEE Device Table Figure 34) is used in this rule as shown in Figure 34.
- Variable values associated with particular instances of device or flow types (e.g., availability of power supplied from multiple sources to a load must exceed that load's required availability threshold): the Load1 variable Availability_min is used in this rule.

Table 5: AEE Device Table

Device#	Abbrev	Device Description	Block Name	Parameters				Variables (unused)
				min	max	maxBPR	VizOrder	
1	Single_Fan_(DTC)		Fans	0	1	14	2	
2	Single_Fan_(nDTC)		Fans	0	1	14	3	
3	Multiple_Fans_(DTC)		Fans	0	1	30	4	
4	Multiple_Fans_(nDTC)		Fans	0	1	30	5	
5	Single_Front_Prop		Front_Props	0	1	20	6	
6	Multiple_Front_Props		Front_Props	0	1	40	7	
7	Single_Rear_Prop		Rear_Props	0	1	20	12	
8	Multiple_Rear_Props		Rear_Props	0	1	40	13	
9	Fan_Diffuser		Diffuser	0	1	100000	1	
10	Core_Diffuser		Diffuser	0	1	100000	8	
11	Fan_Nozzle		Nozzle	0	1	100000	11	
12	Core_Nozzle		Nozzle	1	1	100000	10	
13	Core		Core	1	1	100000	9	
14	Embedded		EmbeddedEngine	0	1	5	14	

Table 6: AEE Flow Table

Flow#	Abbrev	Flow Description	Block Name	Parameters			Variables (unused)
				Available	Desired Pairing	Color	
1	AmbientAir		Gaspath	1		6697881	1
2	DiffusedAir		Gaspath			9869005	
3	CoreAirIn		Gaspath			16776960	
4	BypassAir		Gaspath			16737843	
5	CoreAirOut		Gaspath			8421376	
6	Thrust		Gaspath		1	4858	
7	Exhaust		Gaspath		1	175866	
8	ShaftPower		ShaftPower			65535	

Table 7: AEE Inputs Table

Device#	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Flow#	Single Fan(DTC)	Single Fan(nDTC)	Multiple Fans (DTC)	Multiple Fans(nDTC)	Single Prop	Multiple Props	Single Rear Prop	Multiple Rear Props	Fan Diffuser	Core Diffuser	FanNozzle	Core Nozzle	Core	Embedded
1					1	1	1	1	1	1				
2	1													
3		1											1	
4			1								1			
5				1								1		
6					1									
7						1								
8	1	1	1	1	1	1	1	1						

41	Y	DefineQuantity	MaxBPR	Function	Expression	Min.Devices.maxBPR			
42	Y	DefineParams	MaxTSFCLEOp5	Comparison	Quantity	ReqMaxTSFC	<=	Constant	0.5
43	Y	DefineParams	BPRGE15	Comparison	Quantity	MaxBPR	>=	Constant	15
44	Y	DefineRule		IF	MaxTSFCLEOp5	THEN			BPRGE15

Figure 34: Device Design Rule Based on Device Parameter

- Line 41 defines a quantity MaxBPR which is the minimum (over all devices in the partial architecture so far) of the devices' maxBPR parameter.
- Line 42 defines a logical premise (may be true or false) called MaxTSFCLE0p6, which is true if the quantity ReqMaxTSFC (defined in the Spec for Layer 0: is ≤ 0.6).
- Line 43 defines another logical premise called BPRGE15, which is true if the MaxBPR defined in line 41 is ≥ 15 .
- Finally, line 44 defines the rule, which must be true for each feasible architecture: If the premise MaxTSFCLE0p6 defined in Line 42 is true, then the premise BPRGE15 defined in line 43 must also be true; in other words, if the maximum required TSFC (a measure of efficiency) is better (lower) than 0.6 then the engine's bypass ratio (a driver of efficiency) must be better (higher) than 15.
- numeric values specified as system-level requirements (e.g., Required Maximum Speed = 0.6 Mach#).

To ensure that ALL feasible architectures are found, AEE considers every architecture in the design space, but does NOT actually build or visit every architecture. AEE builds only feasible architectures, and does so by building them one device or flow at a time, testing only the rules applicable to the added or deleted device or flow type. Once a partial architecture is found to be infeasible, an entire subtree within the overall design-space binary tree can be ruled out and avoided. The earlier that infeasibility can be detected, while an architecture is being built, the larger the subtree that can be avoided, speeding up the enumeration process.

1	Y	DefineQuantity	ReqMaxTSFC	Simple	Constant	1.2
2	Y	DefineQuantity	ReqMaxSpeed	Simple	Constant	0.6
3	Y	DefineQuantity	ReqMaxAltitude	Simple	Constant	20000
4	Y	DefinePremise	ReqLowObservables	Simple	Constant	FALSE

Figure 35: AEE Device Design Rules (excerpt)

AEE not only enumerates the feasible architectures, but can also evaluate partial architectures quantitatively, and use their evaluation to reason about them, for example, enforcing quantitative design rules. AEE enables reasoning about network architectures, reaching through the network to compute necessary quantities to test whether requirements are met.

AEE design rules could include external function calls, though that has not been implemented yet. For example, any electric circuit, thermal system or aerodynamic system has a design rule that say that a device cannot be “shorted”. In other words, an electrical device in a circuit cannot have its ends connected by a wire. The current would have no reason to travel through the device, given a path of lower resistance. Similarly, in a thermal circuit, the inputs and outputs of a heat exchanger cannot be connected by a pipe with minimal pressure drop, because if they were, then the flow would bypass the HX. So a design rule for an RLC circuit could be that no single device or chain of multiple devices can be shorted. An algorithmic procedure can be written to compute whether shorting occurs, and could be invoked by a suitable design rule (i.e., “NoShorting”).

AEE could thus evaluate system constraints written as expressions including external function calls. These would constitute simulation, and could invoke simple algorithms or more

complex analyses, e.g., Computational Fluid Dynamics (CFD) and Finite Element Modeling,(FEM).

3.15.3 AEE algorithm

AEE builds architectures one device at a time, so the design space can be considered as a binary tree, in which each successive layer corresponds to the next device type in the Devices table, with duplicate devices occupying more than one row in the binary tree. (see Figure 36) The AEE algorithm visits very few partial architectures in the design space. In Figure 36, these partial architectures are represented by dots. On any given row, the black dots show all of the possible partial architectures representing a decision either to include or exclude that row's device type. So the first row has 2 black dots (not Embedded, Embedded). The second row has 2 black dots, i.e., 2 partial architectures for each of the partial architectures on the row above, created by either including or excluding the current row's device type, e.g.,

1. not Embedded, not Single_Fan(DTC),
2. not Embedded, Single_Fan(DTC),
3. Embedded, not Single_Fan(DTC),
4. Embedded, Single_Fan(DTC).

The algorithm starts at the root of the binary tree, with no decisions made, and proceeds through the tree accumulating devices to build a series of partial architectures, as shown in the blue trace. At each partial architecture visited, specific design rules are checked until either one fails (red dot), or they are found to be true (yellow dot). If one of the design rules fails, then that partial architecture (dot) represents the root of an entire subtree, the bottom row of which cannot contain ANY feasible full architectures.

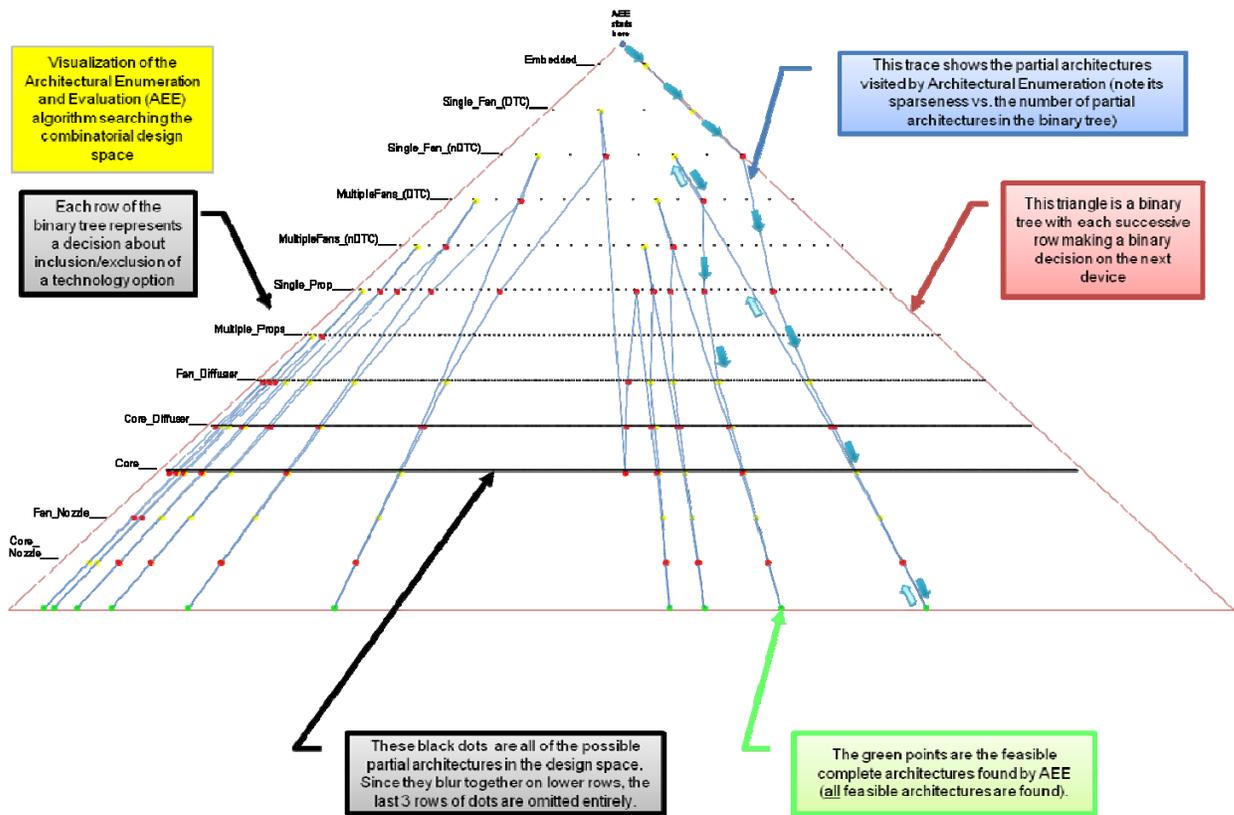


Figure 36: Visualization of AEE Algorithm Trace Through Design Space Binary Tree

Note that the binary tree in Figure 36 consists of 12 rows, so the binary tree's bottom row corresponds to $2^{12} = 4,096$ full architectures, and the entire binary tree interior includes another 4096 partial architectures. There are only 10 full architectures that are feasible (satisfy all design rules). The AEE algorithm visits <5% of the architectures in this binary tree, taking a few seconds. This is a small enumeration design space; the efficiency of the AEE algorithm improves with the size of the design space, as long as there are sufficient design rules so that the feasible set is a sparse subset of the entire design space. The existence of design rules for an application is a function of the maturity of the application. For example, aircraft propulsion systems are comparable in complexity with electric/thermal microgrids, however aircraft propulsion systems have far greater maturity and therefore more numerous design rules, while microgrids are relatively immature (though they include many mature device technologies) so the ways in which the devices are interconnected has relatively few design rules today; in several years, that lack of design rules is likely to change.

3.15.4 Alternate AEE implementations

While the AEE algorithm is extremely efficient and offers many promising approaches to further increase its speed, there are other approaches to consider.

First among these is the use of a general-purpose Constraint Logic Programming (CLP) language and solver. CLP methods include constraint propagation and backtracking, consistency methods, and search heuristics such as forward checking (with and w/o lookahead). We have begun with the primitive Prolog subset of CLP methods and prototyped the Engine L0 Abstraction Layer device enumeration rules. The results were verified and computational speed for this small problem was good. It is not yet clear whether the more relevant and powerful CLP methods enable increased AEE functionality, and what portion of AEE should be done with declarative approaches (i.e., CLP), and what portion is best done by imperative approaches (i.e., standard algorithmic methods).

Architecture-transformation methods can be used to define rules whereby feasible architectures can be transformed into other feasible architectures. This type of rule is not in the form that engineers typically explain their design knowledge. Further, it is not clear how to ensure that this type of design rule will find ALL feasible architectures in the design space.

3.15.5 Mapping between abstraction layers

Once AEE and Optimization have been performed at an abstraction layer, and the best feasible architectures have been selected to be examined further at the next abstraction layer down, each of those best feasible architectures is then used to define an enumeration problem at that next abstraction layer. Each selected architecture bounds the relevant selection of device and flow types. A mapping is needed to automate this process, so that the next abstraction layer's enumeration can be established. For each device in the selected architecture, there needs to be a corresponding list of device types to enumerate at the lower abstraction layer; likewise for flows. Furthermore, if a device in the selected architecture is mapped to one of several sub-networks of interconnected devices, then these configurations can be used for enumeration, reducing the design space significantly. For the Engine application, our abstraction layer definition mapped Layer 0 devices to themselves in Layer 1. So if a device (e.g.,) Fan_Diffuser was present in the L0 architecture, then in the L1 Device table, both the min and max parameters were set to 1. If the Fan_Diffuser was absent in the L0 architecture then at L1, min and max are set to 0.

3.15.6 Optimization

Beyond simulation to enforce constraints, optimization enables multiple objective functions (e.g., design metrics such as Complexity, Adaptability, Performance, First Cost) to be minimized or maximized by varying design variables. In the context of the META Design Flow, at each abstraction layer, once AEE has identified feasible architectures, optimization can be used to find optimal parametric settings (variable values) for each architecture, so that their best incarnations can be compared, to find the best overall architectures.

To enable optimization, any models used within the objective functions must be suitable for the optimization solver's search method. For example, for gradient-based solvers, the models must be continuously differentiable so that the solver doesn't incur numerical instabilities when it encounters discontinuities in the gradient. So models that may be adequate for simulation may not be suitable for optimization. There are solvers that employ heuristic search methods, such as genetic algorithms, which do not require well-behaved models, but which are often significantly slower than gradient-based solvers.

Two specific challenges exist for optimization within the multi-objective META Design Flow context, finding the best diverse set of near-optimal variable values for each architecture, and implementing an optimization of architectures at an abstraction level in terms of a hybrid "gray-box model" consisting of some combination of:

- White-box models: for which the underlying physical mechanisms are available to the solver, characterized in terms of algebraic equations (e.g., written in AMPL)
- Black-box models: for which the underlying mechanisms are not available to the solver, but have been characterized in terms of an executable computer code (e.g., CFD, circuit simulation, FEM, material properties)

A parametric-optimization algorithm was developed, implemented and tested to find the most diverse set of near-optimal solutions, with diversity measured in the design-variable space. Prior work has demonstrated diversity in the objective space, but what is of importance in design is diversity in the design variable space; i.e., radically different solutions that produce high-performing results. This algorithm was tested with an engine application, and found extremely diverse solutions within the range of near-optimality proposed.

An approach was explored, using the AMPL modeling language, to define black-box models (for which equations are not available to the solver) and include them as elements along with white-box models (for which equations are available to the solver) to constitute hybrid "gray-box" full-system optimization models, and a method (e.g., finite difference) for computing local gradients, so that a solver (e.g., IPOPT) using AMPL is provided with results from the black-box models and their gradients when needed. The goal was to identify a seamless framework for developing hybrid system models in terms of combinations of models ranging from almost fully white-box to almost fully black-box. This approach was shown to be successful. It was shown that while entirely black-box optimization had computational speed far slower than white-box optimization, gray-box optimization, had speed somewhere between white and black box optimization. This speed was sometimes observed to be close to the speed of white-box optimization, indicating that the solver can take advantage of the white-box information it has, even if it lacks white-box information for parts of the system. Further, it was found that care must be taken to provide information to the solver about which output variables are functions of which input variables. For example, in a block-diagonal system, if the solver is told that all outputs variables are a function of all input variables, then the solver will compute the "always zero" upper right and lower left terms in the Jacobian and find that they're almost zero. On the

other had, if you tell the solver exactly which output variables are a function of exactly which input variables, then the solver knows which Jacobian terms will always be zero, and won't compute them...which is significantly faster.

To perform a system-level optimization based on device and flow models, automation of the assembly of system-level optimization models enables automated analysis of all feasible architectures produced by AEE. The approach is analogous to assembling a patchwork quilt of models (i.e., patches), interconnected by relevant flow equations for conservation of energy and mass (i.e., sewing). This method was prototyped as part of the July Engine Challenge Problem L0 Abstraction Layer demo.

3.16 Complexity and Adaptability Metrics in Design

Task 4 concentrates on the creation of metrics for use in two steps in the META II design flow, design space exploration and optimization. These two steps are related and may be interleaved, depending upon the nature and complexity of the constraints on the system. We are using a rule based tree pruning program that efficiently rejects infeasible architectures (architectures that fail design constraints or rules) to generate a list of feasible architectures. An optimization step may then follow that further constrains this list of feasible architectures. Finally the results are presented for use in a trade study. Choices are made and the process repeats again with more detailed subsystem decomposition, modeling, and analysis.

Per request from DARPA, the metrics are focused on system complexity, system adaptability and the connection between the metrics and first cost (NRE cost). Ideally, the goal is to create aggregate metrics that can be easily calculated and used during early design space exploration. Many such combinations might be considered. The trick of course, is to prove a solid correlation between NRE and the metric.

The terms “complexity” and “adaptability” have multiple meanings depending upon context, so there are multiple metrics that should be considered.

3.16.1 System Complexity

Simple “common sense” metrics for system complexity abound, and can be a part of this analysis. A count of the number of subsystems is perhaps the simplest metric. A related count is the interface density, (signals per module) both as an aggregate, and also as a count of all subsystems with an interface density greater than some value. These simple metrics miss the additional complexity of dealing with cycles or feedback in the system. Our analysis shows that it is desirable to apply an approach that does include the effect of cycles in an aggregate metric.

The essential elements of this analysis come from a consideration of physical flows that occur between subsystems. The flows can be information, momentum, energy, fluid, heat, gas, electrical current, etc. These flows can be represented in directed graphs, or equivalently using a Design Structure Matrix (DSM). For a system with N subsystems, its Design Structure Matrix is an N x N matrix, with a non zero element indicating the existence of a flow. The value of the element depends upon the kind of analysis being done. It can be a simple Boolean value or a coefficient with some other meaning.

Using DSM's we have concentrated on two metrics for complexity, focusing on “Structural Complexity”, and “Dynamic Complexity”.

3.16.2 Structural Complexity and Graph Energy

By structural complexity, we mean that we are interested in the impact of the connections between subsystems. The metric we propose for this is the “Graph Energy” associated with the DSM for the system.

We use a representation of product architecture using networks consisting of components as nodes of the network and connections between components as edges. Any such network is composed of set of n nodes or vertices and m edges or links. The size of the network is given by the number of nodes in the network. If there exists an edge from node i to node j , then the value of element (i,j) is unity, otherwise the value of the element is zero. In the binary matrix representation of a system, the diagonal elements of the matrix are marked as zero. The adjacency matrix of the product architecture and is known as component or product Design Structure Matrix (DSM) in the system design community. A lot of work has been done in the area of using the product DSMs for efficient partitioning of components into modules, reducing the possibilities of rework during the development cycle, etc. but there has not been many detailed analytical investigation of the product DSMs to understand the underlying fundamental characteristics of the architecture. These characteristics are inherent in the product architecture and are invariant to any re-arrangement of the DSM. These values are affected only when the underlying product architecture itself is modified by changing the interdependencies. The singular values of associated network form such a set of invariant characteristics [Katja et. al., 2007]. Here, we introduce the notion of *graph energy* as a matrix norm and as a plausible measure of *complexity*.

The adjacency matrix $A \in M_{n \times n}$ of a network is defined as follows:

$$A_{ij} = \begin{cases} 1 & \forall [(i, j) | (i \neq j) \text{ and } (i, j) \in \Lambda] \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where Λ represents the set of connected nodes. The diagonal elements of A are zero. For an undirected network, the adjacency matrix A is symmetric and we have, $\sigma_i = |\lambda_i| \forall i \in [1, n]$ where σ_i is the i^{th} singular value and λ_i is the i^{th} eigenvalue value of the adjacency matrix. The singular value decomposition (SVD) and associated *graph energy* of the network are as follows:

$$\left. \begin{array}{l} \text{SVD, } A = U\Sigma V^T \\ \text{Graph Energy, } E(A) = \sum_{i=1}^n \sigma_i \end{array} \right\} \quad (2)$$

where $\Sigma = \text{diag}(\sigma)$ and $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_n$; $U, V \in \Omega_{n \times n}$ where Ω represents the set of *unitary* matrices [10]. It is worth noting that this matrix-based abstract *energy* has found applications in many areas of theoretical and computational chemistry [Gutman 1978, 2008; Koolen 2001]. The singular values can be thought of as manifestation of *energy* associated with design structure or product architecture. Originally graph energy was developed and proposed in molecular and quantum chemistry as a way to measure the complexity and pi-electron energy of large hydrocarbon molecules. The related Hamiltonian matrix, H for the molecular system can be expressed as: $H = \alpha I + \beta A$ where A is the adjacency matrix of a molecular graph that corresponds to the carbon atom skeleton of the molecule. The energy levels of π -electrons are related to the singular values σ_k of the molecular graph by: $E_k = \alpha + \beta \sigma_k$. We propose that the structural complexity of electro-mechanical systems can be estimated by an equation of similar form. The complexity equation has a term that is essentially due to only the complexity of the individual components (containing the α 's), the second term brings in an extra contribution of complexity due to the number of interactions and the way the components are arranged w.r.t each other. This second term is the interesting one and it is the scaled product of the total number of interfaces and graph energy. This term captures the challenges associated with system integration (see Figure 37).

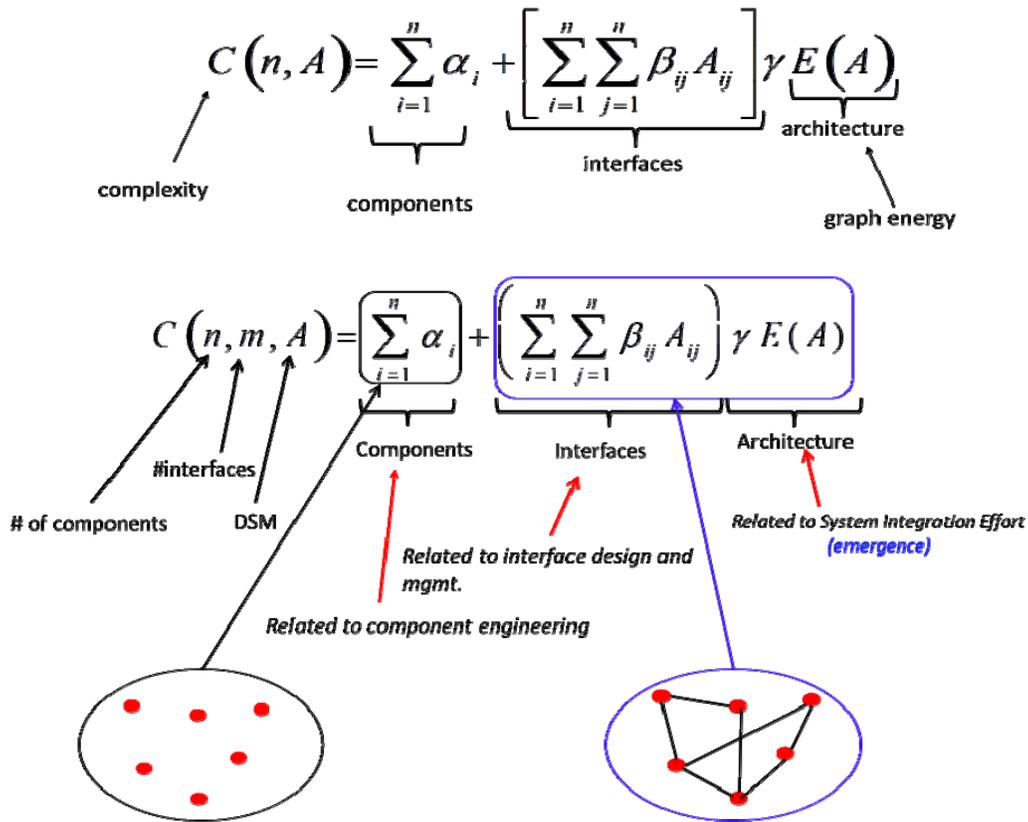


Figure 37: Graph Energy

Notionally, we can think graph energy as a measure of “emergence” in a structural sense and encapsulates the “intricateness” of structural dependency among components. This term help distinguish structural complexity of two very different connectivity structure with the same number of components and interactions as shown in Figure 38.

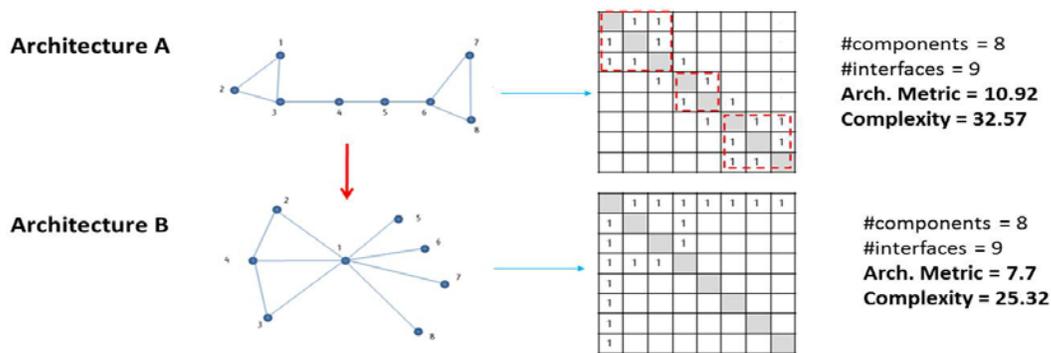


Figure 38: Structural Dependencies

In case of Architecture B above, knowledge of the central component and its connections could enable one to understand most aspects of the system, but the same cannot be said about the Architecture A. The singular value distribution and rank of a matrix has been used as a tool to extract essential aspects/parts of a system and thereby abstract out a reduced, representative system from a much larger system. A rank-dense system is described as a more distributed

system while a rank-deficient system represents a more centralized system. A more distributed system is one that cannot be condensed / reduced. Such a system indicates higher structural complexity but it might also help achieve higher performance levels with higher robustness and reliability. This represents the tension between increased performance and complexity since they tend to share a positive correlation. It raises a very interesting question about how does the structural complexity vs. performance trade space look like. The graph energy is a powerful and rigorous measure of structural complexity that can quantify and explain the inherent structural complexity in electro-mechanical, software-enabled systems as such systems go from being tree-like to much higher degrees of connectedness. This measure depends only on the level of abstraction chosen to represent the system and is expected to be a powerful predictor of required non-recurring engineering (NRE) effort or the development cost in product development. A data-centric validation plan, correlating NRE to structural complexity would consist of the following steps: (1) building of component connectivity network for selected electro-mechanical products; (2) get representative development cost numbers; (3) check for correlation between structural complexity and development cost. Preliminary validation against historical gas turbine engine data suggests a super-linear relation between them (see Figure 39).

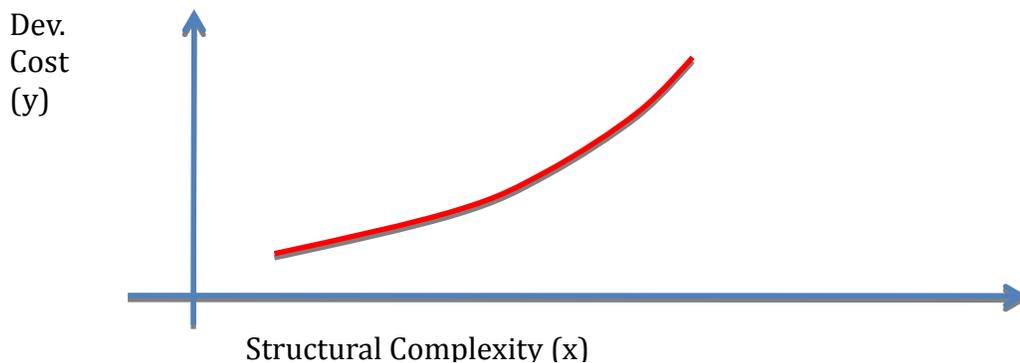


Figure 39: Superlinear Relationship Between Development Cost and Structural Complexity

One issue with the structural complexity metric as proposed is that it is clearly not invariant across abstraction layers or levels, and is therefore hard to apply during early stages of analysis. We attempt to address this by the insertion of the “complexity coefficient” of a subsystem or component, but those coefficients start out at best as estimates, and only become sharp as further analysis is done, or with significant domain specific historical data.

We believe that this is not a significant issue to practical use of the metric. Boolean values for the coefficients suffice for many purposes.

3.16.3 Dynamic Complexity

While Structural Complexity gets at a kernel of interactions between subsystems it does not deal with the essential difficulty of modeling or controlling the physical system. Difficulties in either can seriously impact the cost of developing a cyber-physical system.

The DSM work was extended to produce a metric for design complexity based on Information Theoretical Entropy (Shannon Entropy). An aggregate metric of structural and dynamic complexity is proposed. Other approaches for dynamic complexity were surveyed and worked on, based on metrics on the differential geometry of the physical system. Finally a

measure of the impact of the propagation of uncertainty of the flows between components was considered. (It seems clear that this approach is related to the Shannon Entropy calculation, but this relationship needs to be further explored.)

3.16.4 Shannon Entropy and Dynamic Complexity

Structural complexity applies to the domain of physical elements while dynamic complexity applies to the functional/engineering attributes domain.

Dynamic complexity has two main components:

- (1) Dependency structure among *engineering attributes*
- (2) Inherent uncertainty in the dependency relationships among engineering attributes (Figure 40).

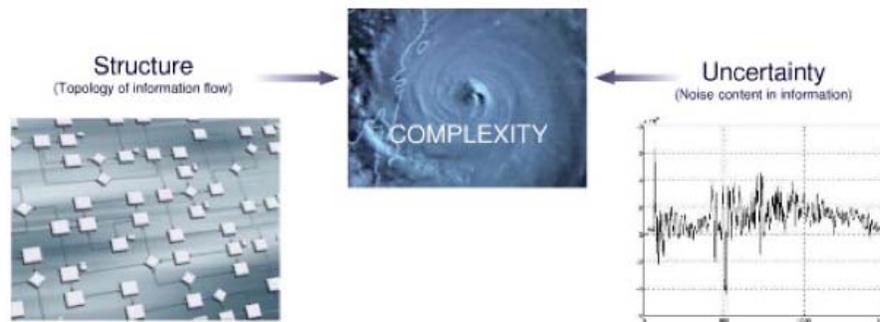


Figure 40: Dynamic Complexity as Uncertainty

The first step is the generation of the dependency structure matrix B ($m \times m$) of representative engineering attributes. This information can be authored from the analytical model if all relationships are understood to last detail (unlikely for a large complex system); or correlation structure “generated” from studying the system at different states (more applicable to large, complex system) or a combination of these two methods. Here we opt for data-driven approach using correlation structure, where data may be generated from simulation or observation (i.e., experiment). No explicit mathematical model is assumed in this method. The information gathered herein can be modified by ‘experts’ to generate the B matrix.

To generate the dependency structure from correlation structure amongst system responses -

1. Identify main functional responses (m) and compute them for (s) “states” of the system. The states could march over time or be points in the mission space. This is a ($s \times m$) data matrix.
2. Extract the correlation structure from the data in step 1.
3. We get a matrix of p - values for testing the hypothesis of no correlation. Each p -value is the probability of getting a correlation as large as the observed value by random chance, when the true correlation is zero. If $p(i,j)$ is small, say less than 0.05, then the correlation $r(i,j)$ is significant.
4. Populate the functional dependency matrix, B with 0’s and 1’s. If $r(i,j)$ is significant (from step 3), assign $B(i,j) = 1$, otherwise $B(i,j) = 0$.

Once the dependency structure B is generated, the next step is to estimate the uncertainty involved in the dependency structure. Uncertainty is estimated using the notion of information entropy (i.e., Shannon entropy). The concept of Shannon entropy, traditionally derived as the

only function satisfying certain criteria for a consistent measure of the "amount of uncertainty" in information theory. There are different ways for computing the entropic content of the dynamical system. The Shannon entropy is defined by:

$$H(X) = \mathbb{E}_X[I(x)] = - \sum_{x \in \mathcal{X}} p(x) \log p(x).$$

where the entropy, H , of a discrete random variable X is a measure of the amount of uncertainty associated with the value of X and $p(x)$ is the probability distribution of the random variable X . In the case of a large-scale, complex engineered system with multiple state variables, $p(x)$ represents a joint probability distribution of the state variables. In this case, the entropic content of the system is computed using the image entropy (Shannon entropy) of the 2D scatter plots of engineering attributes in pairs and finally summing them up. Plot x-y scatter plots for each pair of functional responses. In case of minimal scatter the relationship between the considered responses is "deterministic" indicating that dependency structure dominates, while a large scatter indicates uncertainty dominates. Higher entropy reveals larger uncertainty (Figure 41).

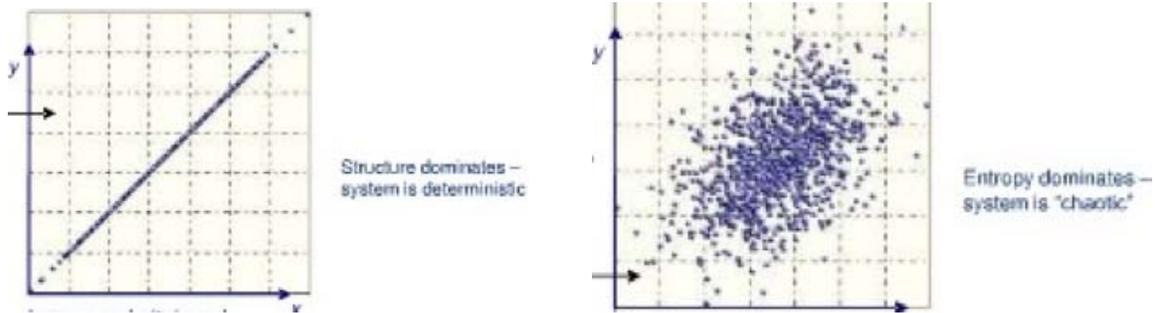


Figure 41: Entropy

Basic steps here include:

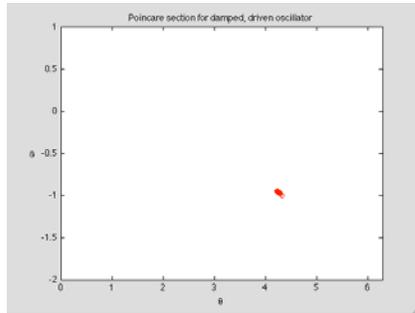
1. **Creation of a 2D image of the scatter plot.**
2. **Extract entropy content from this image.**

Consider each pair of functional responses and build an "entropy matrix" ($m \times m$). Each entry is a normalized entropy measure (≤ 1). Then total entropy is the sum of all elements of the entropic matrix. One can also use a normalized version by dividing the total entropy by the factor $m \cdot (m-1)$. We express the dynamic complexity as the product of a function of dependency structure) and function of uncertainty in functional relationships.

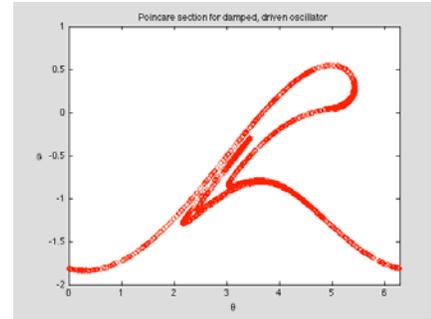
Here Dynamic Complexity = $E(B) \cdot \text{total entropy}$, where $E(B)$ = sum of singular values of B matrix.

3.16.5 Some Examples:

Consider a damped, driven pendulum in (non-chaotic region) and in a chaotic region (Figure 42). The information entropy increases by a factor of 2.8 as the same system with it moves from a non-chaotic to chaotic region.



(1)



(2)

Figure 42: Poincaré Section for (1) non-chaotic regime vs. (2) chaotic regime for damped, driven pendulum system

Similarly, when we consider a single spool turbojet and a single spool turbofan, turbofan has 3 times higher dynamic complexity dynamical complexity than a turbojet engine. Hence, a system can have high dynamical complexity due to: (a) rich dependency/correlation structure (i.e., higher coupling); (b) high relational uncertainty between functional responses (i.e., irreducible uncertainty/model uncertainty) or both of them.

Consolidation of different kinds of complexities to a single/unified measure of complexity:

In order to develop a scalar complexity metric for a large-scale system representing the complexity in physical domain (using structural complexity); complexity in the functional domain using dynamical complexity and the organizational complexity due to development team, we propose a multiplicative model. The multiplicative model has some useful mathematical properties that can be exploited and can be represented as:

System Complexity = structural complexity*dynamic complexity*organizational complexity.

This system complexity metric correlates to NRE can act as a measure of overall complexity that incorporates multiple domains that cuts across various areas that are involved in the development of large-scale, complex engineered system.

3.16.6 Survey of Other Dynamic Complexity Metrics

The following metrics were surveyed:

- Computational and storage complexity(C,S)
- Dimensional complexity (DC)
- Nonlinear dynamic complexity (NDC)
- Failure complexity (FC)
- Diagnostic complexity (DgC)
- Coupling complexity (CC)
- Time-scale complexity (TsC)

3.16.6.1 Computational and Storage Complexity

The Computational and Storage Complexity are simply the CPU resources and storage resources necessary to run the dynamic model of the system or subsystem. Tracking these directly makes sense, and is easy, given the model. Closely associated theoretical metrics are the

algorithmic complexity (N , $N\text{-Log}N$, $N\text{-Squared}$, etc) and a count of the storage elements required.

3.16.6.2 Dimensional Complexity

Dimensional complexity is the dimension of the underlying state space of the model of the physical system.

3.16.6.3 Nonlinear dynamic Complexity

Nonlinear Dynamic Complexity is a ratio of suitable norms of the non-linear / linear behavior of transfer functions associated with the system.

$$\text{NDC} = \frac{\|NL\|}{\|L\|}$$

The choice of the norm is specific to the mathematics of the system. It requires that the system be decomposed into a linear and non linear form, and that a L_p norm (Sobolev Norm) on that form or operator is understood to be appropriate for the system involved.

3.16.6.4 Failure Complexity

The Failure Complexity is the probability of a system or subsystem failure given a specified uncertainty of the system. This can be complicated to assess analytically, but a combination of the results detailed failure mode analysis of previous implementations, actual failure data, and details of the mission environment for the subsystem, it may be possible to create a usable metric. Our belief is that an early attempt to quantify this metric during an analysis of a product family should bear significant fruit because unexpected system failures and associated rework have a significant impact on the cost of large engineering projects

3.16.6.5 Diagnostic complexity (DgC)

This metric measures the difficulty in diagnosing a problem with a system or subsystem. Again, this can be complicated to assess analytically, although work is going on in this area – Prof. Joel Tropp from Caltech is developing randomized algorithm based techniques to approximately compute DgC in an efficient manner. DgC is also amenable to calculations based on historical data from related systems or subsystems.

3.16.6.6 Coupling Complexity

Coupling Complexity measures the level of coupling between elements of the dynamic system, viewing the system by its Jordan canonical form of the system or its linearization. We quantify the CC by looking at the Jordan canonical form of the system or its linearization. If the system has k distinct Jordan blocks, it means that if you look at it in the right coordinates, it has k decoupled subsystems.

3.16.6.7 Time Scale Complexity (TsC)

The main driver behind time-scale complexity is the fact that when the system dynamics has a mixture of oscillatory and slow modes, the analysis and simulation becomes nontrivial. A TsC metric is comprised of a norm on a vector of the time scale differences between system elements. (The Euclidean norm would work for this purpose).

3.16.6.8 Comparison with Dynamic Complexity based on Shannon Entropy

One issue with all the surveyed metrics is that they for the most part require significant knowledge of the mathematical environment of the model of the physical system involved. This knowledge may or may not be well known, and may not be accessible to engineering teams. The proposed Shannon Entropy metric can be calculated from repeated simulations of the models of

the physical system. This requires less analytical knowledge of the details of the model --- you just run it to find out how it behaves.

In all cases, we are measuring the behavior of a model. As such the correlation between the model and the actual physical system is critical, and knowing the limits of how to extrapolate behavior from the model is a key element for successful use.

3.16.7 Propagation of Uncertainty and Dynamic Complexity

The main thrust behind uncertainty assessment is to

“develop methods automatically identify sources and sinks of uncertainty in systems, as well as its effect on the complexity metrics”

The main drivers behind uncertainty assessment were

- Mapping of individual sub-components as emitters, propagators and absorbers of uncertainties, both controlled internal uncertainties and external influences
- Quantification of uncertainty as a user friendly scalar number (the EPA number) which acts as an enabler for architecture selection

3.16.7.1 Emitters, propagators and absorbers of uncertainty

In this section, we describe how we characterize each component in a specific architecture as emitter, propagator and absorber of uncertainty. To achieve this task, we model each component as an input-output map. For example, in a Turbofan model, the basic components consist of a diffuser, high pressure compressor, high pressure turbine, a burner and a nozzle. The inputs to each of these components include thermodynamic variables like temperature and pressure as well as other physical variables like the Mach number of the flow. Given such an input-output representation, one can now generate algorithms to classify these components as sources or sinks of uncertainties using smart sampling techniques.

3.16.7.2 Emitters of uncertainty

Emitters (or sources) of uncertainties are components which in general scale up the uncertainty in terms of output variance given a certain input variance. For example, if the input has variance one, the output variance will be a number greater than one to qualify to be called an emitter of uncertainty. A very good example of an emitter model is a simple Brownian motion model. In this case, the underlying dynamics is give by

$$dx = \sigma \sqrt{dt} \quad ; \quad x(0) = 0$$

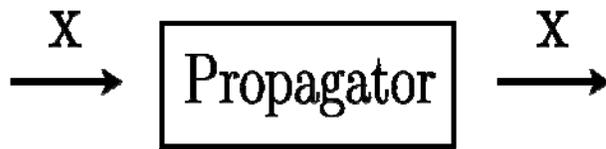
$$\underline{x(0)} \rightarrow \boxed{\text{Brownian}} \rightarrow \underline{x(T)}$$

As an input-output map, the input is zero (the initial state) and the output is the state of the system at some fixed time T. The Brownian motion can be considered as an ideal model for an emitter. The input has zero variance and the output has variance T. In other words, no matter how precisely you know your inputs, the output is always a random variable with variance T.

3.16.7.3 Propagators of uncertainty

Propagators of uncertainties are components which neither shrink nor scale up input uncertainties. They can be considered as marginal systems as far as uncertainty is concerned. For

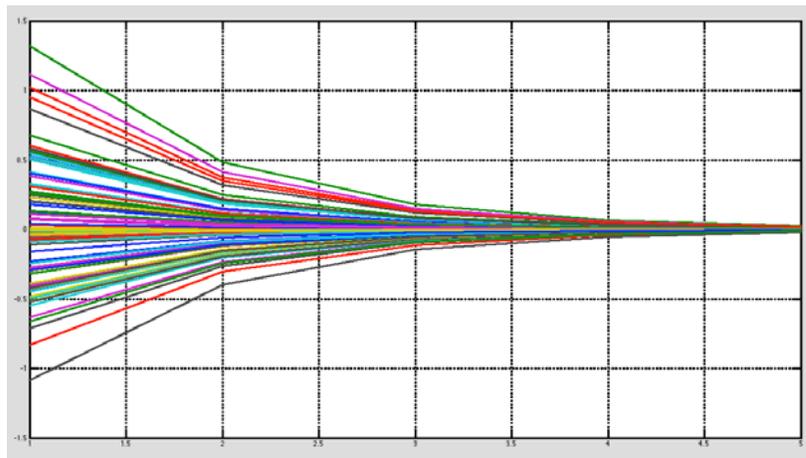
example, a harmonic oscillator without dissipation is a good model as a propagator of uncertainty because the variance of the output is the same as that of the input.



A very good example of an ideal propagator is depicted above where the input is the same as the output.

3.16.7.4 Absorbers of uncertainty

Absorbers of uncertainty are components which damp out the input uncertainties. If the input variance is one, the output variance has to be less than one to qualify to be called an absorber of uncertainty. See the figure below of an exponential damping system, i.e., an absorber. The input has variance zero and the output has exponentially decaying variance in time. And for any given finite time, the component is an absorber of uncertainty.



An ideal absorber of uncertainty is depicted below. In this case, no matter what the input is, the output is always a fixed constant. An infinite heat bath is a very good physical example of an ideal absorber. The heat bath maintains a constant temperature no matter how much heat we extract from it or add to it.



3.16.8 Quantifying EPA numbers

Quantifying EPA numbers for components is a challenging task and many times run the risk of being subjective. The main reasons which make this task daunting are

- Multiple inputs and multiple outputs
- Mismatch of physical dimensional units for the inputs and outputs which make the interpretation of results harder.
- Nonlinearity of the system which makes accurate computation of the output variance computationally harder.

We now state our definition of EPA number and then provide some background justification and motivation for this definition.

Definition: Consider an input-output system (IOS) with m inputs and n outputs given by (x_1, x_2, \dots, x_m) and (y_1, \dots, y_n) respectively. Let the standard deviations of the inputs and outputs be given by (S_1, \dots, S_m) and (T_1, \dots, T_n) respectively. Then we define

$$EPA = \frac{\frac{T_1}{|y_1|} + \dots + \frac{T_n}{|y_n|}}{\frac{S_1}{|x_1|} + \dots + \frac{S_m}{|x_m|}}$$

The EPA number defined as above take the following values for emitters, propagators and absorbers.

- Emitter: EPA = Infinity
- Propagator: EPA = n/m
- Absorber: EPA = 0

The main features of this definition are

- The EPA is a dimensionally consistent way to quantify the size of output variance to the input variance, i.e., one can combine temperatures, pressures, Mach number etc which show up
- The ratio of the standard deviation to the mean is called the coefficient of variation (CV) in statistics and this quantity has been used very successfully in renewal theory, queueing theory and reliability theory. A slightly related concept (ratio of variance to the mean), called the index of dispersion or the Fano factor is also a very commonly used quantity in studying dispersion of probability distributions.
- The EPA number can formally be considered as the inverse of signal-to-noise ratio. For input-output systems, the SNR quantifies how much the system scales up the information content in the input.

3.16.9 Issues with Correlation with First Cost

Historical data from multiple gas turbine engine projects were examined for correlation with complexity metrics. These studies for the Gas Turbine Engine and other Aerospace systems are problematic for several reasons. The first is the Intellectual Property and Export Control issues require that data be scrubbed, or normalized in some fashion to comply with company policy and U.S. Export Control law. The second is that the cost data is often not available in a work breakdown structure that matches subsystems of interest for the analysis.

Significant data mining, “financial forensics”, and conjectures are necessary to build a coherent picture. The difficulty was not anticipated when we started work but it puts this analysis beyond the scope of what can be accomplished on the 1 year DARPA META II project schedule. In any case, all the metrics were applied to real systems and applicability assessed, but the scope of the study must be expanded to be definitive.

It is suggested that for moving forward to META follow-on projects, metrics and cost be tracked explicitly, and analyzed for relevant correlations. Note that in the software industry, this

has been done with several complexity metrics, some of which were found to work only “in theory” but failed to be useful in actual practice. Other metrics, such as McCabe Cyclomatic Complexity, have stood the test of time and heuristics have been developed for their effective application on projects. It is expected that this must also hold true in the Model Based Systems Engineering space where META II’s methodologies live.

3.16.10 Adaptability Metrics

We want the “adaptability” metric associated with a particular architecture to show how easy or hard it is to modify or adapt the architecture to a change in specification. As such, “adaptability” requires that the system requirements or “mission space” be characterized in some modeling form, and that the enumerated architectures from design space exploration have some coefficients or metrics associated with them that reflect this goal.

Our focus is on the adaptability of a product platform -- a collection of designs “on the shelf” that can be combined in various ways to produce specific products. This differs from “robustness” which is the capability of a system to handle various situations outside of its primary design points.

As with the metrics for “complexity” there are some common sense notions that should not be lost in the focus on elegant mathematical ideas. The first common sense notion that is extremely important for the adaptability of a design in a product family is a scaling factor. If a design will function with a range of scale factors without changing the essentials of the design, then it is re-usable simply by adjusting the scaling factor. The aggregate scaling factor for a collection of subsystems is then just the smallest scaling factor of its components.

Another more difficult metric that should be considered is a degree of isolation that is possible for a subsystem. If by addition of a component (like a dust filter) or an easy change to the design such as perhaps making it a “sealed unit”, the component can be isolated from environmental effects, then the design naturally can be extended to include a more hostile mission environment than the original intent. This should result in a high “subsystem isolation” coefficient.

The other adaptability notion that we have explored is the switching cost to modify one enumerated design to handle the optimal design points of another design. This involves a pair-wise comparison of the switching cost of each pair of feasible designs that are output from the enumeration step during “Design Space Exploration” in response to a particular mission constraint.

3.16.10.1 A Switching Cost Metric applied to early abstraction levels on the Gas Turbine Engine

We believe that the differences in structural complexity between two designs can be calculated directly from the DSM’s for those designs, and that this will have a high correlation with switching cost. This conjecture needs further analysis from historical project data in multiple domains before it can be asserted with confidence.

We have investigated other forms of metrics associated with switching cost that can be applied in early analysis before the DSM decompositions have enough content to show essential system complexity.

Our definition of this adaptability metric for particular system-architecture is a utility function of a system-architecture and all the mission elements in the mission space. In another word, it is $U_A(A, M)$ where A is the architecture being discussed and M is the mission set. The adaptability of a system can be modeled as the total benefits of all the missions to perform subtracted with the cost of turning this architecture to support these missions. Alternatively, it

can be just the total cost of turning this architecture to support these missions. The latter definition reflects how hard it is for a certain design to perform all needed tasks. Adding the mission benefits in this utility function is optional. It implies that the benefit can compensate the cost given the benefit is big enough. In reality the benefit can be financial as well as strategic. For simplicity, here we just use financial profit as benefit in our example.

To understand how adaptable a system-architecture is, the first step is to obtain the mission information, and this can be done through mission space analysis, which typically includes two steps. First, a system level model as a black box model is used to illustrate the system inputs, outputs, and major internal variables; Second, all the possible mission segments associated with this system.

A mission generally consists of several mission segments. For example: a flight mission may contain mission segments such as taxing, take-off, climb, cruise, attack, climb, cruise and land. The following example illustrates the basic set of mission segments for three types of aircrafts: baseline aircraft that performs city to city short distance flights, commercial jet that flies within a continent, and transatlantic jet that flies across oceans (Table 8).

Table 8: Flight mission segments

	1. Baseline aircraft	2. Transatlantic jet	3. Commercial jet
Engine Failure Takeoff	False	True	True
Climb Rate	1000ft/min	2500ft/min	2500ft/min
Cruise Distance	700 nautical miles	4000 nautical mile	2000 nautical mile
Cruise M_{in}	0.5	0.8	0.8
Cruise altitude (ft)	7000m	11000m	11000m

Some mission segments are required to be satisfied by all architectures, and some others are optional that are only satisfied by a subset of architectures. Given a set of optional mission segments M , we must identify all their valid combinations (Mission Segment Combinations, MSC). Note that this is not a simple $|M|$ dimensional Cartesian product, as one mission segment can contain another one. This stage of analysis requires proper breaking down of the mission segments to form the MSC set. This is achievable with Use Case Diagram analysis in software engineering.

Once MSC is available, the next step involves with traversing through all the generated architectures by AEE, and calculating the adaptability metrics for them. Each architecture is assumed to be assigned with parameters or ranges of its parameters for each its supported $MSC_i \in MSC$. If not, an optimal sizing process is needed, such that the parameters are assigned under any objective functions that user specify, for example, minimized cost. After this step, each architecture is associated with all the MSC elements that it supports. For all these supported MSC elements, the cost of adapting to them is 0. Given an architecture $arch_j$, each unsupported MSC element MSC_i , we must compute the cost of adapting $C(arch_j, arch_k)$ for all $k \in S(i)$ where $S(i)$ contains the indexes of all architectures that support MSC_i . The adapting cost of $arch_j$ for MSC_i is then defined as $\min C(arch_j, arch_k)$ for all $k \in S(i)$.

To compute $C(arch_j, arch_k)$, several techniques can be used. One simple way is to compare all the components, flows between $arch_j$ and $arch_k$. Each new component in $arch_k$ incurs additional cost that includes the cost of such a component and the cost of adding such a component. Same method applies to flows as well in which cost of connecting all the components on the flow are included. A component cost can be obtained by either historical

data if available, or a recursive algorithm that treating the component as a system and further breaks down into sub-components and sub-flows until the cost can be easily estimated. Using the latter method can usually connects with other methodology such as DSM in which both components and flows/dependencies are captured. Cost of removing components/flows does not include the cost of the component or flow itself but cost must be included when re-design is needed, in which case added component/flow in the redesign are treated as new components/flows in archk. When comparing the same components with different parameters, a scaling effect needs to be taken into account. If the component in arch-k is within the scaling allowance of compared component arch-j, the cost should be 0. This reflects the reality that sometimes a component can be built easily by just scaling it to a larger or smaller size, with negligible efforts.

Figure 43 illustrates the algorithm of computing the adaptability metrics.

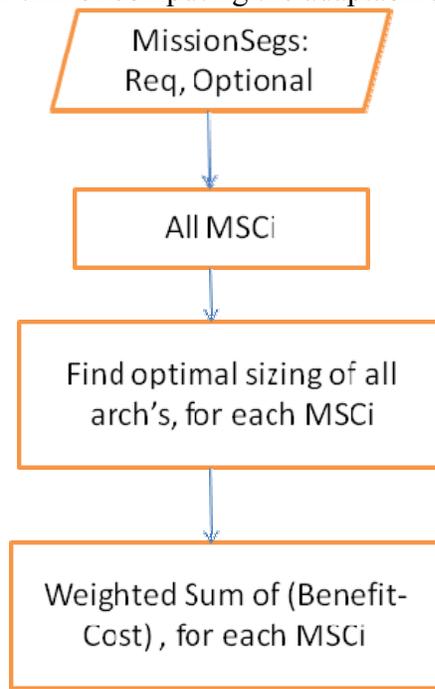


Figure 43: Computing adaptability metric

Below we used an example to show how this can be applied to an EPS case, in which three mission segments are defined: baseline (basic needs for a plane, such as supporting the electrical and electronic devices functioning), entertainment and cargo. Two important facts decide these abilities, capacity and electrical network. In order to support cargo, the capacity needs to be enough to support the loading equipment. For entertainment support, the electrical network must route to all the seats, while in the cargo case, it needs to be designed to route to the back of the plane (Figure 3). In this case, the optional mission segments are entertainment, and cargo. The $|\text{MSC}|=4$.

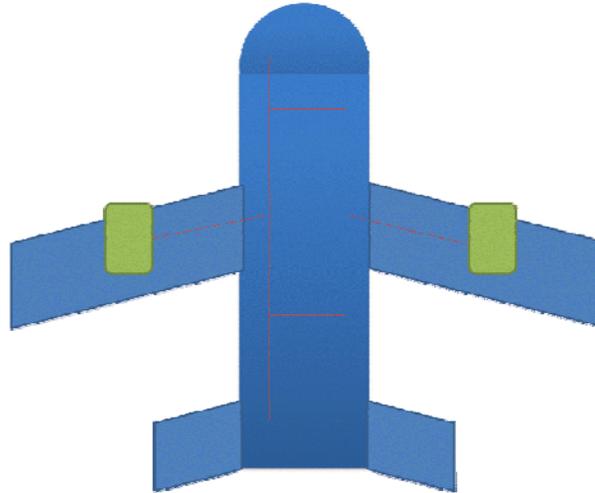


Figure 44: Simplified illustration of EPS wiring

There are three components defined in a level 1 of an EPS system: generator, load and connection. Different architectures are formed using a combination of these components. Following the process described earlier, each architecture's adapting cost is 0 under the supported mission segments, and the switching cost to closest architecture (in terms of cost) A_k for the unsupported mission segments. And the weighted sum of these costs, or the MSC benefits B subtracted costs against an architecture (A) are calculated.

$$U_A(A) = \sum_{k=1}^{MSC} w_k C(A, A_k)$$

or

$$U_A(A) = \sum_{k=1}^{MSC} w_k [B(k) - C(A, A_k)]$$

3.16.11 Design Flow Modeling

The main goal of META is to achieve a 5x improvement in product development for cyber-physical systems compared to current practice. The Complex Systems Design and Analysis (CODA) process being developed by the UTRC team aims to achieve this kind of speedup by a combination of three main mechanisms:

1. The deliberate use of *layers of abstraction*. High-level functional requirements are used to explore architectures immediately rather than waiting for downstream level 2,3,4 ... requirements to be defined.
2. The development and use of an extensive and trusted C2M2L *model library*. Rather than designing all components from scratch, the CODA process will allow importing component models directly from the library in order to quickly compose functional designs.
3. The ability to find emergent behaviors and problems ahead of time during *virtual V&V* allows a more streamlined design process and avoids costly design iterations and “turnbacks” that often lead to expensive design changes.

In order to quantify the impact of these main META mechanisms we have developed a System Dynamics (SD) simulation that permits simulation projects ahead of time. The key to this is the ability to estimate the schedule and non-recurring engineering (NRE) cost profile of design projects, both traditional and META-inspired. Figure 45 shows a causal-loop diagram of the current version of the model.

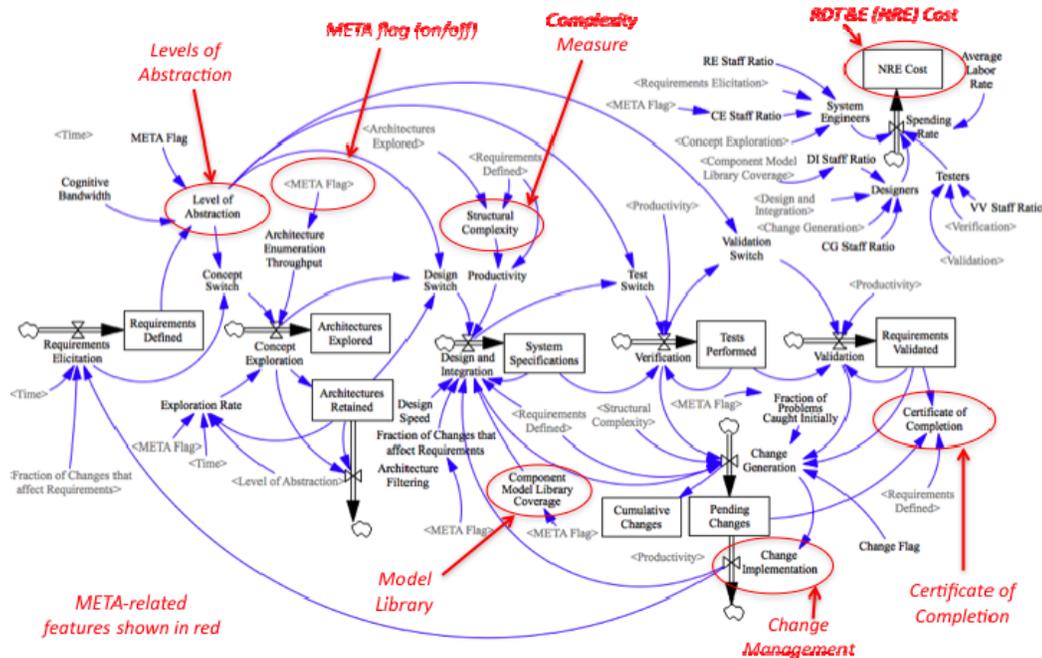


Figure 45: Vensim System Dynamics Model of Product Development

The model is generally read from left to right and begins with requirements elicitation, followed by concept exploration, design and integration, verification and validation. Discrete “switches” represent the milestones that turn on or off the ability to move to the next phase. The META-enabling features in Figure 45 are shown in red and include the use of levels of

abstraction, the existence of a model library and the issuance of a certificate of completion when all the defined requirements have been validated *and* no more changes are pending.

The model can be used to simulate traditional product development where all phases are done sequentially in a stage-gate manner. When the Change Flag is turned off we simulate an idealized project where no errors are ever made and therefore no design changes are ever generated. Figure 46 shows what such an idealized project might look like in terms of the rate of work during requirements definition, architecting, design and integration as well as verification and validation.

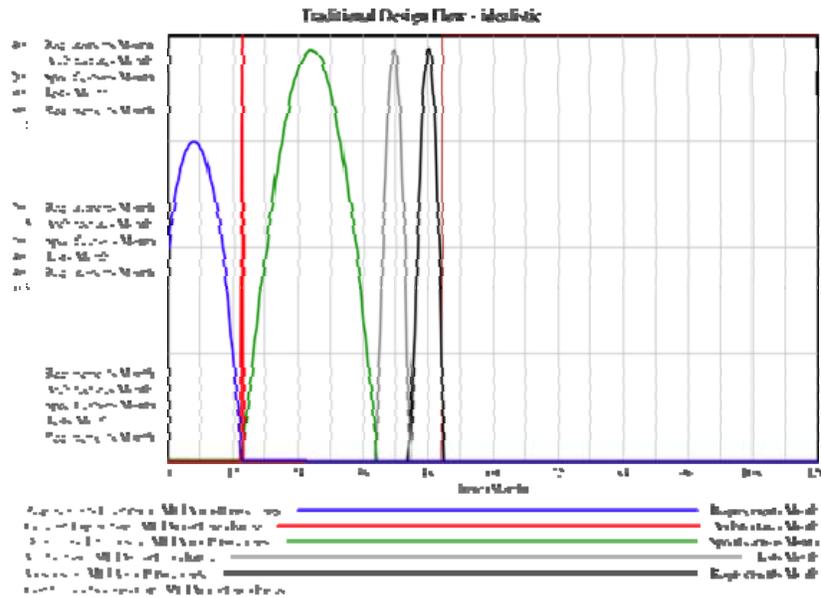


Figure 46: Idealized Sequential Project (Current Practice) – Completion Time 51 Months

Acknowledging the existence of engineering changes makes a big difference. Turn on the change flag in the model allows the generation of changes in design & integration, validation and verification. These changes have to be processed and may add to the work to be done during design & integration but can also lead to requirements changes (both changing existing requirements and adding new ones). This leads to a dramatic impact on the project as shown in Fig. 3. Rather than finishing “on time” the project experiences a number of subsequent waves of change. These waves of change are also referred to as “turn-backs” and represent costly design iterations that are needed to incorporate new information and fix problems before the design can be converged.

According to our simulation the project would complete after 87 months, which represents a 70% schedule increase compared to the optimistic, idealized schedule. Such overruns are often observed in practice for projects that are scheduled in an overly optimistic fashion.

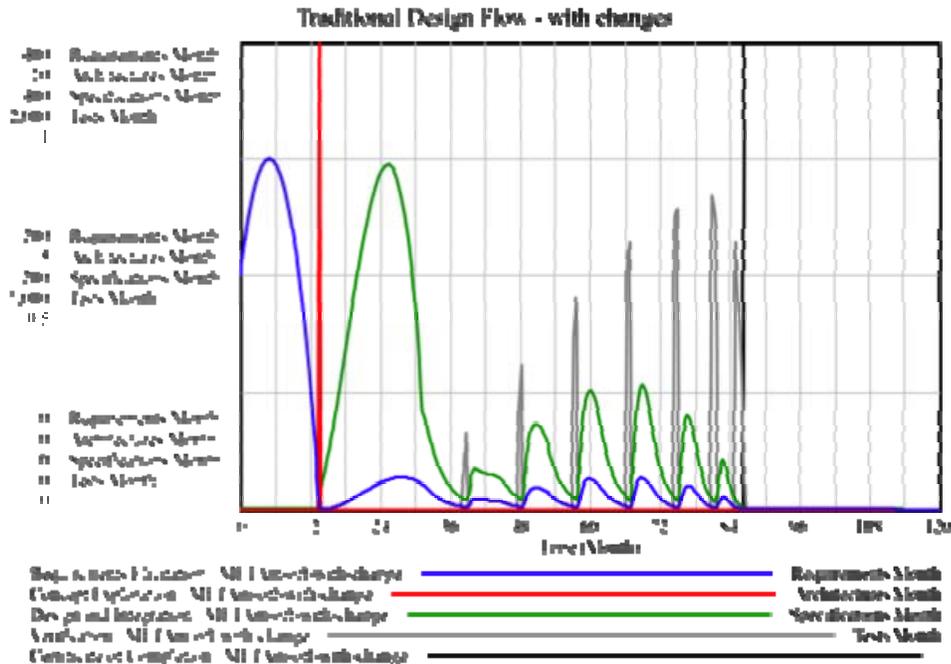


Figure 47: Realistic Project with Changes (Current Practice): Completion Time 87 Months

Enabling the META process yields a very different dynamic. First, design and integration can start as soon as high level requirements are available. Second the rate at which design & integration can progress is accelerated due to the existence of a model library. Third, the ability to predict problems ahead of time leads to a dampening of - but not complete elimination – of design changes. Figure 4 shows a simulated META project with a projected completion time of 17 months.

This represents a 5.2x speedup compared to current practice where projects often struggle to complete due to the waves of change as shown in Figure 3. The assumptions that yielded the behavior shown in Fig. 4 are as follows:

- META = on
- Model Library coverage: 80%
- Cognitive Bandwidth: 7 (leading to the use of 4 layers of abstraction)
- Ability to detect problems early to avoid design changes: 75%

We are currently conducting a series of model refinements and sensitivity analyses to show how sensitive the speedup factor is to the elements of the META CODA process. Work so far indicates that the model library (C2M2L) and the use of layers of abstraction are the two most important mechanisms to create speedup.

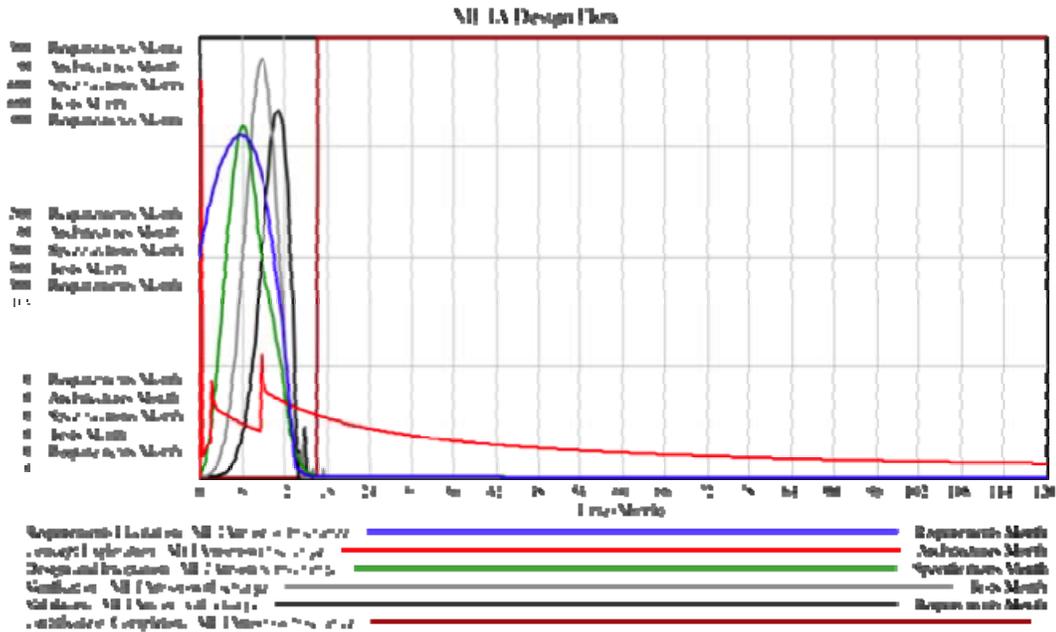


Fig.4: Simulated META project: Completion time: 17 months (5.2x speedup)

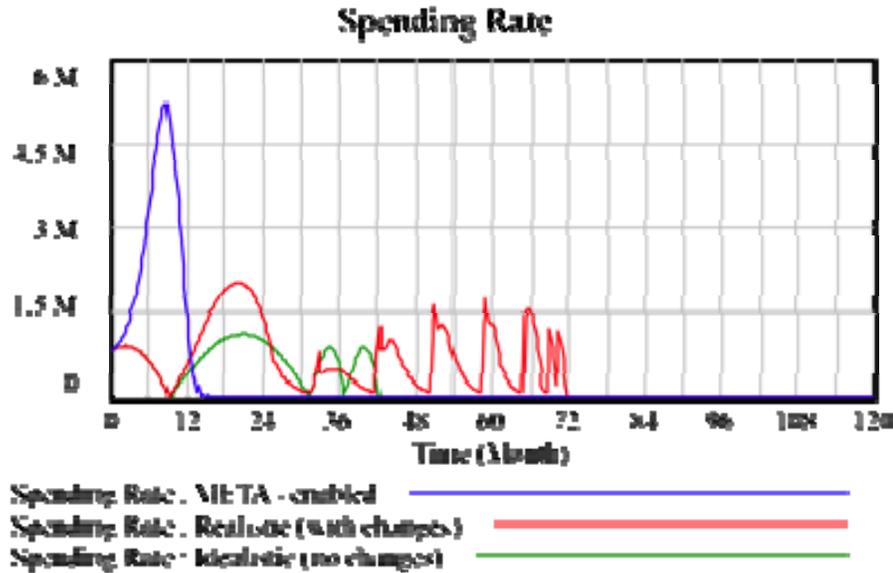
The main features of the META process shown in Fig.4 are as follows:

- Overlapping project phases
- Short duration to complete (~ 17 months)
- Architecture Exploration with “spikes” when transitioning to next layer of abstraction (see red curve in Fig.4)
- Rapid design thanks to model library
- Virtual and early V&V with help of META toolset
- Some minor rework in the last 2 months

Sensitivity Analysis

The table below shows the results from running our standard reference case. This is a comparison of an idealistically planned project (green) where no changes or error are assumed versus a realistic project with changes (“turn-backs”) shown in red versus a META enabled project in blue. This project assumes a baseline of about 3,000 individual requirements. We also show the simulation assumptions on the chart.

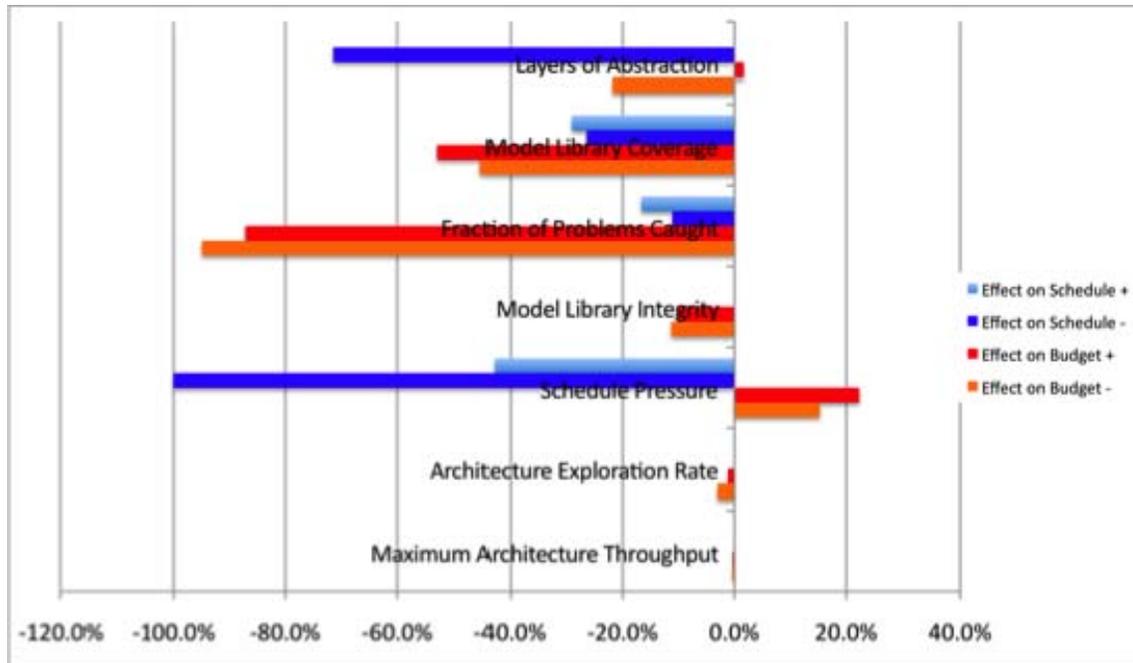
Simulation Case	Schedule to complete	NRE \$ to complete
Idealistic Project	42.25 months	\$27.9M
Realistic Project w/changes	70 months	\$51.9M
META-enabled project	15.75 months	\$31.5M



The results show a dramatic improvement for project completion schedule with a demonstrated speedup factor of 4.4 (70 months versus about 16 months). Here the schedule pressure was set to 1.5 and in META we use 3 layers of abstraction, 50% novelty and a library integrity of 80% and 70% of problems are caught early. **THE IMPORTANT MESSAGE IS THAT THE META TOOL CHAIN DOES NOT HAVE TO BE 100% PERFECT TO YIELD SUBSTANTIAL BENEFITS.**

The simulation of Non-Recurring Engineering Effort (\$) shows that the improvement relative to the realistic project with changes is only 1.5. This is because work is done at a higher rate and more people are needed, despite the automation. It is also important to note that the META project is more expensive than the idealistically planned project where no changes (turn-backs) are accounted for. Thus META's main strength is speeding up the design process, not saving development money.

We took the baseline model and varied each of the 7 META factors up (+) and down (-). For each one-at-a-time perturbation of each factor we simulated the project and recorded the impact on project schedule and cost. Recall that the baseline META project took 15.75 months to complete at a NRE \$ cost of \$31.5 million, while the traditional project (w/o) META took 70 months to complete at and NRE cost of \$51.9 million.



The sensitivity results shown above are normalized, so for example a value of -40% means that if the factor is increased by 100% (doubled) then the impact on cost or schedule will be a 40% reduction. First we notice that most sensitivities are negative. Increasing layers of abstraction from 2 to 3 leads to a significant reduction in schedule, as does applying schedule pressure. Also increasing model library coverage and fractions of problems caught early by the META tool-chain leads to speed-up. Increasing the architecture exploration rate and maximum number of architectures that can be carried does not lead to a speedup (but improvement of technical performance and slight budget reduction).

This sensitivity analysis confirms and adds nuance to some of our key findings to date:

- Working in layers of abstraction is extremely important for speedup. There seems to be a sweet-spot around 3 layers of abstraction (for a system with about 1,500 requirements) , adding more layers does not yield much more benefit.
- The C2M2L model library is very important for both schedule and cost. The completeness of the library (“coverage”) seems to be a more important factor than ensuring that it is completely error free. In other words for META it appears to be better to have a near complete library even if it has bugs in it. We suspect that the reasons that model library errors are not that important is because the rest of the META tool-chain is able to find and resolve such errors quickly.
- The ability of the META tool-chain to catch emergent problems early is very important, especially for the NRE budget. The reason is that problems that are caught during virtual V&V can be discovered quickly, but they do cost effort for their resolution.

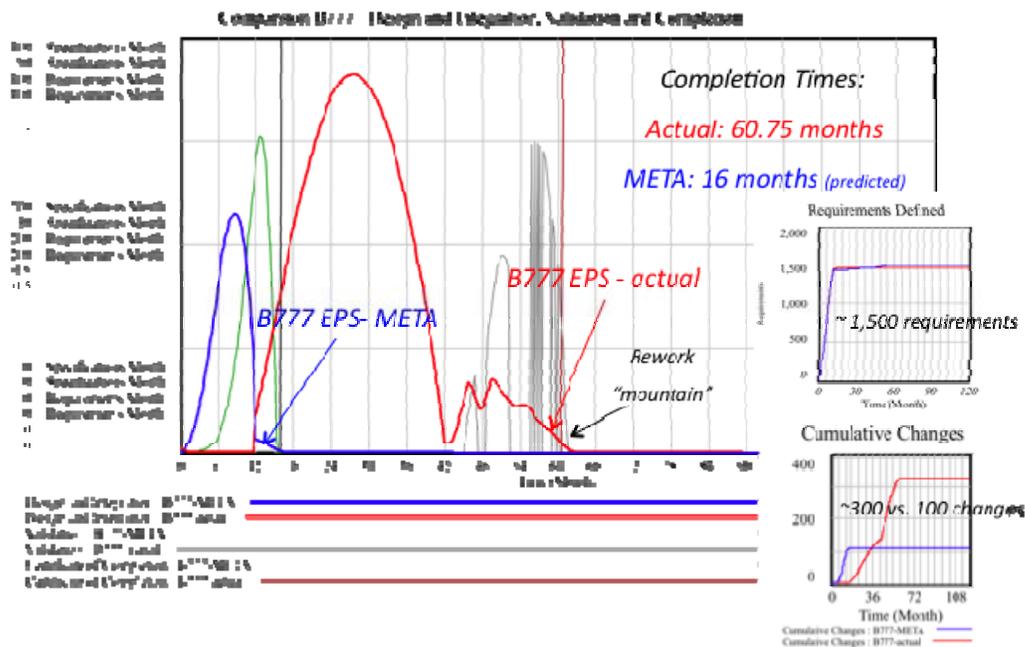
Traditional Schedule Pressure by management does also lead to a speedup in META, but it has limits.

Simulation of B777 EPS Product Development Process

We applied the META System Dynamics Model to the Boeing B777 Electric Power System (EPS) Design. Hamilton Sundstrand (HS), a UTC company performed the design of the power system with/for Boeing from 1990-1995. The B777 is generally viewed as a success story and was one of the first programs where CAD/CAM/CAE were used at a large scale. We worked with the HS team to obtain some key program data that is shown on the chart.

We then tuned or calibrated the SD Vensim model to replicate as closely as possible the actual behavior of the B777 EPS project. Some important features was the general timeline from start to completion of Design and Integration and the source control drawing, the fact that about 300 ECRs (engineering change requests) were generated and roughly the ratio of systems engineers to test personnel.

We then turned on the META “flag=1” in the model to see how the project might have unfolded, had META been available at that time.



11

The chart above compares the design and integration effort, validation effort and certificate of completion (straight vertical lines) for the B777 actual project (red) versus what it might have looked like with the META tool-chain (blue). Very noticeable is the “rework mountain” for the B777-actual between month 42 and 60 towards the end of the program as problems were surfaced during V&V. The META program might have completed after 16 months, with only a small amount to rework at the end (see small blue “wedge” between month 12 and 16). The small inset figures on the right show that both simulations assume that about 1,500 customer

requirements have to be satisfied. While the actual B777 project generated about 300 changes, it is predicted that META would generate about 100 changes.

The speedup factor here is 3.8 x

4.0 CONCLUSIONS

The Design Flow Modeling done by the UTRC CODA team demonstrates that a 5x speedup in the design process is possible if certain assumptions are met and if a process similar to the one proposed here is used. This process is characterized by design space exploration through multiple levels of abstraction using design metrics including complexity and adaptability as well as more traditional objective functions such as performance and cost. At each level of abstraction, the components used to construct architectures (model library) together with structural and functional constraints called contracts collectively form a “platform.” This is the essence of platform-based design which has been successfully used for many years in VLSI design to control complexity. Applying this approach in cyber-physical design is a new approach, but appears promising based on the work here.

Some of the key requirements to successfully use the META CODA process to reduce design time include: model library coverage, the appropriate use of 2 or more levels of abstraction and contracts that capture essential design constraints that would lead downstream to rework if not captured early in the design process. A comprehensive approach to design space exploration is the glue that brings these elements together.

The CODA project has established basic elements to implement and deploy the META design process.

5.0 REFERENCES

1. Moir, I., & Seabridge, A. (2008). Aircraft Systems (3rd ed.). Wiley.
2. Eismín, T. K. (2002). Aircraft Electricity & Electronics (5th ed.). McGraw Hill.
3. Saadat, H. (2002). Power System Analysis (2nd ed.). McGraw Hill.
4. L. Benvenuti, A. Ferrari, E. Mazzi, and A. L. Sangiovanni Vincentelli, "Contract-based design for Computation and Verification of a Closed-loop Hybrid System," In Hybrid Systems: Computation and Control, vol. 4981 of Lecture notes in Computer Science, pp. 58--71, 2008.
5. Andrade, L., & Tenning, C. (1992). Design of Boeing 777 Electric System. Aerospace and Electronic Systems Magazine, IEEE , vol. 7, pp. 4-11.
6. Balabanian, N., & Bickart, T. A. (1969). Electrical Network Theory. John Wiley & Sons.
7. Chen, C.-T. (1999). Linear System Theory and Design (3rd ed.). New York, NY: Oxford University Press.
8. Franklin, G. F., Powell, J. D., & Emami-Naeini, A. (2002). Feedback Control of Dynamic Systems (4th ed.). Upper Saddle River, NJ: Prentice-Hall.

APPENDICES

APPENDIX
Architecture Enumeration and Evaluation (AEE):
Design-Rule Specification Language

By

Lawrence E. Zeidner
United Technologies Research Center, East Hartford, CT 06108

1.0 INTRODUCTION

This document describes the Design-Rule Specification Language (DRSL) used within AEE to express Design rules. Design rules are used by AEE to rapidly explore a design space by rapidly ruling out large logical sections of the design space as necessarily infeasible. Rather than constructing every complete system architecture in the combinatorically large design space, and then using “evaluative” rules to check them for feasibility, “generative rules” are used to build and test partial architectures progressively, checking them as each new technology option is added. These rules are called “generative” because the majority of the full-system architectures they build are feasible, the infeasible ones having been ruled out, without explicitly visiting most of them, as the partial architectures were being built. Ultimately, as the last technology option is assembled, the partial architecture becomes a full-system architecture.

AEE first finds all of the combinations of devices that satisfy the device rules. Then, for each feasible combination of devices, it finds all of the combinations of flows between these devices that satisfy the flow rules. The result of this second design-space exploration (DSE) is the list of ALL feasible full-system architectures, including devices and flows. The second DSE is usually significantly more computationally intensive than the first.

AEE does not actually “enumerate” each potential architecture in the combinatorically large design space. It is fast primarily because it is able to rule out large sections of the design space, based on particular partial architectures failing specific design rules. AEE is also fast because it doesn’t check every design rule for every partial architecture that it constructs. AEE determines, before it starts exploring the design space, for each design rule, and for each type of device or flow that is being assembled on to the previously feasible partial architecture checked last, whether the design rule could possibly fail. During design space exploration, AEE consults this information to check only a small fraction of the design rules for each partial architecture. Finally, AEE achieves additional speed by only evaluating each value described in the rules, at most, once per partial architecture.

2.0 CONSTRUCTS

Design rules describe both the mission to be performed, and compatibility among the technology options. These two classes of design rules are often segregated because rules about compatibility among the technology options are reusable and can be selected automatically based on the selection of relevant technology options, while each design problem includes its own new mission. In this description, we will not segregate based on this distinction.

Design rules are expressed in terms of statements that specify logical relationships and comparisons between real and Boolean values. In DRSL, three types of statements are defined. Real-valued “quantities” can be defined, logical “premises” can be defined, and “rules” can be expressed in terms of comparisons between quantities or between premises. Further, quantities and premises can contain either scalar values, vector values, or a tree-like nested vector of values. These will all be called vectors of different dimensionality, i.e., scalar: $V(0)$, vector: $V(1)$, vectors of vectors: $V(2)$, $V(3)$, etc. For example, some simple quantity values:

- Scalar quantity (quantity vector of dimension 0, i.e., $QV(0)$): 3.75
- Quantity vector of dimension 1 (i.e., $QV(1)$): (2.5, 3.14, 0.2, 100.6)
- Quantity vector of dimension 2 (i.e., $QV(2)$): (6.1, (8.7, 9.2, 124.0), (-45.2, 15.0), -19.7)
- Quantity vector of dimension 3 (i.e., $QV(3)$): (8.6, -72.2, (81.0, -25.3, (101.2, -14.2), -99.1), 0.0, -8.1)

Examples of simple premise values:

- Scalar premise (quantity vector of dimension 0): 1 [Note: 1 represents True here; 0 represents False]
- $PV(1)$: (0, 1, 1, 0, 1)
- $PV(2)$: (1, 0, (1, 1, 0, 1), 0, 1, (1, 1), 0, 1)
- $PV(3)$: (1, 1, 1, 0, 1, (0, 0, (1, 1, 0), 0), 0, 1, 1)

In DRSL, each statement can specify a rule, a premise, or a quantity. A rule is simply a premise that is enforced to be true. For vector values of premises, rules are enforced to be completely true at the deepest level only, so the following examples show how premise values would be evaluated as rules:

- True $PV(0)$: 1

- False PV(0): 0
- True PV(1): (1, 1, 1, 1)
- False PV(1): (1, 0, 0, 1)
- True PV(2): (1, 1, (1, 1, 1), 0, 1, (1, 1, 1), 0, 0, 0)
- False PV(2): (1, 1, (1, 1), 1, 1, (1, 0), 1)

3.0 DEVICE-RULE STATEMENT SYNTAX

Since premises and rules use the same syntax, there are really only two types of device-rule statements:

`<device-rule-statement> ::= <dr-define-premise-or-rule> | <dr-define-quantity>`

Defining Premises or Rules

There are multiple types of device-rule statements to define premises or rules.

`<dr-define-premise-or-rule> ::= <dr-premise-rule-simple> | <dr-premise-rule-comparison>
| <dr-premise-rule-mutual-excl> | <dr-premise-rule-logical-fn>`

Defining a premise based on a constant

A premise can be set to a constant value.

`<dr-premise-rule-simple> ::= <dr-id-act-defpr-name> "Simple" "Constant" <t-f>`

where the following non-terminals are defined:

`<dr-id-act-defpr-name> ::= <stmt-id> <y-n> <dr-def-premrule-name>`

All device-rule statements begin with a statement id, which is a sequence number

`<stmt-id> ::= <positive-integer-sequence-number-for-rules>`

Next is a term to indicate whether or not the statement is active. This is a practical consideration, since it allows the user to turn statements on and off to test and use the design rules for AEE.

`<y-n> ::= "Y" | "N"`

Next are two terms to indicate whether this statement is defining a premise or a rule. If it is a rule, then its truth will be enforced. If it is a premise, then it must have a name to be referenced in subsequent statements. If it is a rule, this name is optional.

`<dr-def-premrule-name> ::= <dr-def-rule> | <dr-def-premise>`

`<dr-def-rule> ::= "DefineRule" <pname> | "Define Rule"`

`<dr-def-premise> ::= "DefinePremise" <pname>`

`<pname> ::= <name>`

`<name> ::= <text-without-blanks>`

`<t-f> ::= "True" | "False"`

For example, a device-rule statement to define a premise based on a constant would be:

1, Y, DefinePremise, Embedded, Simple, Constant, True

This statement would define the premise Embedded to be true.

Defining a premise based on a comparison

A premise can be set to a constant value.

`<dr-premise-rule-comparison> ::= <dr-id-act-defpr-name> "Comparison" <quantity> <ineq> <quantity>`

where the following non-terminals are defined:

`<quantity> ::= "Constant" <realnumber-value>`

`| "Quantity" <qname>`

`| "#DeviceInstances" <device-name>`

```

<qname> ::= | "#BlockInstances" <device-blockname>
           <name>
<ineq> ::= "<" | "<=" | "<>" | "=" | ">=" | ">"
<device-name> ::= "Device_Name" <text-name-from-device-table>
<device-blockname> ::= "Device_BlockName" <text-blockname-from-device-table>

```

For example, a device-rule statement to define a premise based on a comparison between two quantities would be:

2, Y, DefinePremise, Requirement_Satisfied, Comparison, Actual, >=, Required

This statement would define the premise Requirement_Satisfied to be true if the quantity Actual is >= the quantity Required.

Defining a premise based on Mutual Exclusion

A premise can be set to be true if a set of only one of a set of other premises is true, i.e., the set of premises is mutually exclusive.

```

<dr-premise-rule-mutual_excl> ::= <dr-id-act-defpr-name> "MutualExcl" <pname> <pname>
                                | <dr-id-act-defpr-name> "MutualExcl" <pname> <pname> <pname>

```

For example, a device-rule statement to enforce mutual exclusion among three premises would be:

3, Y, DefineRule, MutualExcl, TurboProp, TurboFan, TurboJet

This statement would ensure that a jet engine architecture can't have propeller(s) together with fan(s), but can have either propeller(s) or fan(s), or neither (turbojet).

Defining a premise based on a Logical Function

A premise can be defined based on a logical function applied to 1-3 premises:

```

<dr-premise-rule-logical_fn> ::= <dr-id-act-defpr-name> <logic1> <pname>
                                | <dr-id-act-defpr-name> "BoolFn" <pname> <logic2> <pname>
                                | <dr-id-act-defpr-name> "IF" <pname> "THEN" <pname>
                                | <dr-id-act-defpr-name> "IF" <pname> "THEN" <pname> "ELSE" <pname>
<logic1> ::= "NOT"
<logic2> ::= "AND" | "OR" | "NAND" | "NOR" | "IF/THEN"

```

For example, a device-rule statement to define a premise based on a logical function applied to 2 premises would be:

4, Y, DefinePremise, Multiple_Fans, BoolFn, Multiple_Fans_Ducted_to_Core, OR,

Multiple_Fans_Not_Ducted_to_Core

This statement would define a premise that included multiple fans whether or not they were ducted to the core.

Defining Quantities

There are multiple types of statements to define device-rule quantities.

```

<dr-define-quantity> ::= <dr-quantity-simple> | <dr-quantity-math-fn>
                        | <dr-quantity-simple-#dev> | <dr-quantity-simple-#devblk>
                        | <dr-quantity-expression>

```

Defining a quantity based on a constant

A device-rule quantity can be set to a constant value.

```

<dr-quantity-simple-constant> ::= <dr-id-act-defpr-name> "Simple" "Constant" <real-number>

```

For example, a device-rule statement to define a quantity based on a constant would be:

5, Y, DefineQuantity, Required_TSFC, Simple, Constant, 0.8

This statement would define the required thrust-specific fuel consumption of a jet engine to be 0.8.

Defining a quantity based on the number of instances of a device type

A device-rule quantity can be set to the number of instances of a specific device type in a partial architecture.

```
<dr-quantity-simple-#dev> ::= <dr-id-act-defpr-name> "Simple" "#Device_Instances" <device-name>  
<device-name> ::= "Device_Name" <text-name-from-device-table>
```

For example, a device-rule statement to define a quantity based on the number of generator instances would be:
6, Y, DefineQuantity, nGenerators, Simple, #Device_Instances, Generator

Defining a Quantity to be the Number of Device Instances in a Device Block

A device-rule quantity can be set to the number of device instances that are members of a specific device block in a partial architecture. Device blocks are contiguous collections of device types that have something in common. For example, if PrimGen90 is a 90 kW primary generator, and PrimGen120 is a 120kW primary generator, then both of these devices could be in the block PrimGen, which includes all primary generators.

```
<dr-quantity-simple-#devblk> ::= <dr-id-act-defpr-name> "Simple" "#Block_Instances" <device-blockname>  
<device-blockname> ::= "Device_BlockName" <text-blockname-from-device-table>
```

For example, a device-rule statement to define a quantity based on the number of load device instances that are AC vs DC would be:

7, Y, DefineQuantity, nAC_Loads, Simple, #Block_Instances, AC_Load

Defining a Quantity Based on a Device Parameters or Variables

A device-rule quantity can be set to the value of a specific device-type parameter or variable, or a function of the value of a specific device parameter across all device instances. Device types are defined in the device table, along with parameters and variables that may apply to each device type. Device parameters are specified in device-table columns. The parameter values exist within that column, and correspond to the row's device type.

```
<dr-quantity-expression> ::= <dr-id-act-defpr-name> "Function" "Expression" <dev-expression-mathfn2>  
| <dr-id-act-defpr-name> "Function" "Expression" <dev-expression-devparvar>  
<dev-expression-mathfn2> ::= <math-fn2>".Devices." <dr-parvar>  
<dev-expression-devparvar> ::= <dr-dev-name> "." <dr-parvar>  
<dr-parvar> ::= <text-parameter-name-from-device-table>  
| <text-variable-name-from-device-table>  
<dr-dev-name> ::= <text-name-from-device-table>
```

For example, a device-rule statement to define a quantity based on an a mathematical function applied to a specific device parameter of all of the device instances in the partial architecture would be:

8, Y, DefineQuantity, Max_Individual_Cruise_Load, Function, Expression, MAX.Devices.power

A device-rule statement to define a quantity based on a particular device type's parameter value would be:

9, Y, DefineQuantity, PrimGen90_Power, Function, Expression, PrimGen90.power

Example Device-Rule Statements

The following is a set of device-rule statements used as input to AEE to enumerate Layer 0 device-only architectures for an aircraft Electric Power System (EPS).

Rule #	Active	Type	Name						Comments	
1	Y	DefineQuantity	PrimGen90_cap	Function	Expression	PrimGen90.power			Power capacity of each PrimGen90	
2	Y	DefineQuantity	PrimGen120_cap	Function	Expression	PrimGen120.power			Power capacity of each PrimGen120	
3	Y	DefineQuantity	M_power_sum	Function	Expression	Sum.Devices.M_power			Total power required in Maintenance mode	
4	Y	DefineQuantity	IO_power_sum	Function	Expression	Sum.Devices.IO_power			Total power required in Take-Off mode	
5	Y	DefineQuantity	C_power_sum	Function	Expression	Sum.Devices.C_power			Total power required in Cruise mode	
6	Y	DefineQuantity	E_power_sum	Function	Expression	Sum.Devices.E_power			Total power required in Emergency mode	
7	Y	DefineRule		Comparison	#BlockInstances	PrimGen	<=	Constant	10	Max 10 PrimGen
8	Y	DefineRule		Comparison	#BlockInstances	PrimGen	>=	Constant	2	Min 2 PrimGen
9	Y	DefineQuantity	minPrimGen_cap	Function	Quantity	C_power_sum	/	Constant	0.8	Max PrimGen capacity is C_power_sum/0.8
10	Y	DefineQuantity	maxPrimGen_cap	Function	Quantity	C_power_sum	/	Constant	0.85	Min PrimGen capacity is C_power_sum/0.85
11	Y	DefinePremise	TwoPrimGen90	Comparison	#DeviceInstances	PrimGen90	=	Constant	2	Two PrimGen 90 exists
12	Y	DefinePremise	FourPrimGen90	Comparison	#DeviceInstances	PrimGen90	=	Constant	4	Four PrimGen 90 exists
13	Y	DefinePremise	SixPrimGen90	Comparison	#DeviceInstances	PrimGen90	=	Constant	6	Six PrimGen 90 exists
14	Y	DefinePremise	EightPrimGen90	Comparison	#DeviceInstances	PrimGen90	=	Constant	8	Eight PrimGen 90 exists
15	Y	DefinePremise	TenPrimGen90	Comparison	#DeviceInstances	PrimGen90	=	Constant	10	Ten PrimGen 90 exists
16	Y	DefinePremise	PrimGen90Muts1	BoolFh	TwoPrimGen90	OR	FourPrimGen90			
17	Y	DefinePremise	PrimGen90Muts2	BoolFh	SixPrimGen90	OR	EightPrimGen90			
18	Y	DefinePremise	PrimGen90Muts3	BoolFh	TenPrimGen90	OR	PrimGen90Muts1			2, 4, 6, 8, OR 10 PrimGen90 allowed
19	Y	DefinePremise	PrimGen90Muts	BoolFh	PrimGen90Muts2	OR	PrimGen90Muts3			
20	Y	DefinePremise	SomePrimGen90	Comparison	#DeviceInstances	PrimGen90	>=	Constant	1	some PrimGen90
21	Y	DefineRule		IF	SomePrimGen90	THEN	PrimGen90Muts			PrimGen90 must exist in multiples of 2
22	Y	DefineQuantity	sumPrimGen90_cap	Function	Quantity	PrimGen90_cap	*	#DeviceInstances	PrimGen90	total power capacity from PrimGen90
23	Y	DefinePremise	TwoPrimGen120	Comparison	#DeviceInstances	PrimGen120	=	Constant	2	Two PrimGen 120 exists
24	Y	DefinePremise	FourPrimGen120	Comparison	#DeviceInstances	PrimGen120	=	Constant	4	Four PrimGen 120 exists
25	Y	DefinePremise	SixPrimGen120	Comparison	#DeviceInstances	PrimGen120	=	Constant	6	Six PrimGen 120 exists
26	Y	DefinePremise	EightPrimGen120	Comparison	#DeviceInstances	PrimGen120	=	Constant	8	Eight PrimGen 120 exists
27	Y	DefinePremise	TenPrimGen120	Comparison	#DeviceInstances	PrimGen120	=	Constant	10	Ten PrimGen 120 exists
28	Y	DefinePremise	PrimGen120Muts1	BoolFh	TwoPrimGen120	OR	FourPrimGen120			
29	Y	DefinePremise	PrimGen120Muts2	BoolFh	SixPrimGen120	OR	EightPrimGen120			
30	Y	DefinePremise	PrimGen120Muts3	BoolFh	TenPrimGen120	OR	PrimGen120Muts1			2, 4, 6, 8, or 10 PrimGen120 allowed
31	Y	DefinePremise	PrimGen120Muts	BoolFh	PrimGen120Muts2	OR	PrimGen120Muts3			
32	Y	DefinePremise	SomePrimGen120	Comparison	#DeviceInstances	PrimGen120	>=	Constant	1	some PrimGen120
33	Y	DefineRule		IF	SomePrimGen120	THEN	PrimGen120Muts			PrimGen120 must exist in multiples of 2
34	Y	DefineQuantity	sumPrimGen120_cap	Function	Quantity	PrimGen120_cap	*	#DeviceInstances	PrimGen120	total power capacity from PrimGen120
35	Y	DefineQuantity	TotalPrimGen_cap	Function	Quantity	sumPrimGen90_cap	+	Quantity	sumPrimGen120_cap	intermediate sum of PrimGen capacity
36	Y	DefineRule		Comparison	Quantity	TotalPrimGen_cap	<=	Quantity	maxPrimGen_cap	Total PrimGen capacity <= maxPrimGen capacity
37	Y	DefineRule		Comparison	Quantity	TotalPrimGen_cap	>=	Quantity	minPrimGen_cap	Total PrimGen capacity >= minPrimGen capacity

4.0 FLOW-RULE STATEMENT SYNTAX

Since premises and rules use the same syntax, there are really only two types of statements:

<flow-rule-statement> ::= <define-premise-or-rule> | <define-quantity>

4.1 Defining Premises or Rules

There are multiple types of statements to define premises or rules.

<define-premise-or-rule> ::= <premise-rule-reachout> | <premise-rule-reduction>
| <premise-rule-filters> | <premise-rule-comparison> | <premise-rule-pairing>

4.1.1 Increasing Dimensionality of Nested Vectors

Nested premise vectors are created by a mechanism called “reach out.” In AEE, if we start with a premise vector corresponding to the set of all device instances in a partial architecture, we can reach out from those devices to the set of all flows emanating from them. For example, if we have 3 generator instances in the partial architecture: PG1, PG2, APU1, and various flows of electrical power leading from them to various loads: DCLoad1, DCLoad2, ACLoad1, ACLoad2, as shown in **Figure 48**,

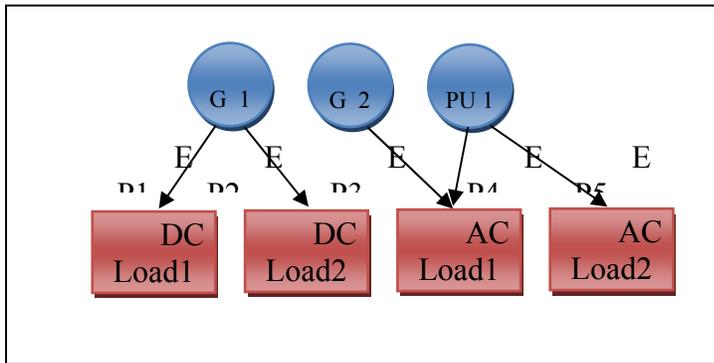


Figure 48: A simple Electric Power System (EPS) example

There are 7 device instances of multiple device types, and 5 flow instances. In this example, a premise vector describing the generators would be a PV(1) called Generators:

(1,1,1,0,0,0,0),

where the three 1s represent the 3 generators and the four 0s represent the 4 loads. If we use the reachout mechanism to reach out from the generators, we can produce a PV(2) called Gen_Outflows:

((1,1,0,0,0),(0,0,1,0,0),(0,0,0,1,1),0,0,0,0).

This PV(2) has 7 terms at the 1st level, which correspond to the terms in Generators. For each 1 in Generators, there is a nested vector of 5 terms, corresponding to the 5 flow instances. For PG1, this nested vector is:

(1,1,0,0,0)

Since PG1 has EP flows out to DCLoad1 and DCLoad2. Likewise, PG2 and APU1 each have their own nested vectors:

(0,0,1,0,0) and (0,0,0,1,1)

Corresponding to PG2’s EP flow out to ACLoad1, and APU1’s EP flows out to ACLoad1 and ACLoad2.

Reachout can extend both forward to consider flows leaving a device, or backward to consider flows entering a device. Reachout can also start from flows. For example, in figure 1, we can start with a PV(1) called EP_Flows:

(1,1,1,1,1),

Where the 1s correspond to the flows EP1, EP2, ..., EP5. If we use the reachout mechanism to reach out from these EP flows, considering the devices on their input ends, we can produce a PV(2) called EP_Flows_InDevs:

((1,0,0),(1,0,0),(0,1,0),(0,0,1),(0,0,1)),

where each EP flow has one input device, so there are 5 nested vectors, each with only one 1. The first and second nested vectors correspond to EP1 and EP2, and they both are:

(1,0,0),

since their input devices are PG1.

Reachout can extend multiple levels, for example starting with Generators, reaching out to create the nested vector of their outflows, Gen_Outflows, and then reaching out to the devices on those outflows output ends, to create the PV(3): Gen_Outflows_Loads:

((0,0,0,1,0,0,0),(0,0,0,0,1,0,0),0,0,0),(0,0,(0,0,0,0,0,1,0),0,0),(0,0,0,(0,0,0,0,0,1,0),(0,0,0,0,0,0,1)),0,0,0,0).

At the deepest level in this PV(3), there are 5 nested vectors, corresponding to the 1s in the PV(2) Gen_Outflows. Each of these 5 nested vectors has 7 terms, corresponding to the 7 device instances. For example the first one is:

(0,0,0,1,0,0,0),

referring to the fact this output device is ALoad1, which is the output of flow EP1, out from PG1.

The syntax for statements to define premise vectors via reachout is:

```
<premise-rule-reachout> ::= <pr-reachout-flows-6> | <pr-reachout-devs-7>
<pr-reachout-flows-6> ::= <id-act-defpr-name> <pv-pvname> <io-flows>
<pr-reachout-devs-7> ::= <id-act-defpr-name> <pv-pvname> <io-devices>
<io-flows> ::= "Input_Flows" | "Output_Flows"
<io-devices> ::= "Input_Devices" | "Output_Devices"
```

where the following non-terminals are defined:

```
<id-act-defpr-name> ::= <stmt-id> <y-n> <def-premrule-pvname>
```

All statements begin with a statement id, which is a sequence number

```
<stmt-id> ::= <positive-integer-sequence-number-for-rules>
```

Next is a term to indicate whether or not the statement is active. This is a practical consideration, since it allows the user to turn statements on and off to test and use the design rules for AEE.

```
<y-n> ::= "Y" | "N"
```

Next are two terms to indicate whether this statement is defining a premise or a rule. If it is a rule, then its truth at its deepest level will be enforced. If it is a premise, then it must have a name to be referenced in subsequent statements. If it is a rule, this name is optional.

```
<def-premrule-pvname> ::= <def-rule> | <def-premise>
```

```
<def-rule> ::= "DefineRule" <pvname> | "Define Rule"
```

```
<def-premise> ::= "DefinePremise" <pvname>
```

```
<pvname> ::= <name>
```

```
<name> ::= <text-without-blanks>
```

Next are two terms to reference the premise vector argument for reachout. The first is simply a keyword "PV" signaling that the next term is the name of a premise vector. The next term can either be a premise vector defined in another statement, or it can be one of two keywords. If it is "Partial_Devices" then the premise vector will be a PV(1) consisting of all 1s, with one term per device instance in the partial architecture. If it is "Partial_Flows" then the premise vector will have one term per flow instance.

```
<pv-pvname> ::= "PV" <prior-pv>
```

```
<prior-pv> ::= <pvname> | "Partial_Devices" | "Partial_Flows"
```

4.1.2 Reducing dimensionality of nested vectors

A nested premise vector can be reduced by applying a dyadic logical or mathematical function along the elements of its deepest dimension. For example, if we begin with the vector describing the generators in the example above, it is a PV(1) called Generators:

(1,1,1,0,0,0,0),

where the three 1s represent the 3 generators and the four 0s represent the 4 loads. If we use reduction with the AND logical function along its one and only dimension, we will get:

(1 AND 1 AND 1 AND 0 AND 0 AND 0 AND 0),

which results in a False. Note that 1s represent True and 0s represent False here. The meaning of this reduction in this example is to ask the question, “Are all of the devices generators?” In this case there are 3 generators and 4 loads, so the answer is No.

The syntax for PV reduction is

```
<premise-rule-reduction> ::= <pr-reduce-logic-8>  
<pr-reduce-logic-8> ::= <id-act-defpr-name> <pv-pvname> "Reduce" <logic2> <target-depth>
```

where the following non-terminals are defined:

```
<logic2> ::= "AND" | "OR" | "NAND" | "NOR" | "IF/THEN"  
<target-depth> ::= <non-negative-integer>
```

Target depth provides a degree of control to the user, since reachout doesn’t always add a new dimension. For example, if we had constructed the PV(1): Interim_Result, it might have the following content for certain but not all partial architectures:

(0,0,0,0,0,0,0)

and if we applied reachout to it, we would get the same exact PV(1) as the result, though in general, we’d expect reachout from a PV(1) to produce a PV(2). This situation-dependent uncertainty is why the target depth construct is provided, to introduce certainty back into the reduction operation. The user specifies the target depth that he intends to reduce down to. If the PV is already at that dimensionality, no action is taken (as in the case of the reachout from Interim_Result, above). If the PV is at a higher dimensionality, the reduction is applied as many times as is necessary, each time at the deepest level, until the target depth is reached.

4.1.3 Filtering a premise based on corresponding information

A premise vector can be filtered, at its deepest level, by checking a specific condition.

```
<premise-rule-filters> ::= <pr-filter-dfname-1> | <pr-filter-dfblkname-2>  
| <pr-filter-dfgrpname-3a> | <pr-filter-dfparvar-ineq-q-3>  
| <pr-filter-dfparvars-ineq-4> | <pr-filter-logic-5>
```

4.1.3.1 Filtering by Device/Flow Type Name

A premise vector representing, at its deepest level, the device/flow instances in a partial architecture can be filtered by specifying a **device/flow type name**; the resulting premise vector’s deepest-level terms will be unchanged from the original premise vector’s terms for terms that correspond to device/flow instances with that name, and set to false (0) otherwise.

```
<pr-filter-dfname-1> ::= <id-act-defpr-name> <pv-pvname> <df-name>  
<df-name> ::= "Device_Name" <text-name-from-device-table>  
| "Flow_Name" <text-name-from-device-table>
```

So, in the example above, filtering based on a device name would enable creation of a PV(1): Primary_Generators, simply by starting from “Partial_Devices”:

(1,1,0,0,0,0,0).

The statement would be:

1, Y, DefinePremise, Primary_Generators, PV, Partial_Devices, Device_Name, PrimaryGenerator

Filtering by Device/Flow Block Name

Similarly, a premise vector can be filtered by specifying a **device/flow block name**; the resulting PV’s deepest level will be unchanged from the original premise vector’s terms that correspond to device/flow instances with that block name, and set to false (0) otherwise. A block is a contiguous collection of device types listed in the Device

table, all listed there with the same block name. Device block names are listed in one column within the Device table.

```
<pr-filter-dfblkname-2> ::= <id-act-defpr-name> <pv-pvname> <df-blockname>
<df-blockname> ::= "Device_BlockName" <text-blockname-from-device-table>
| "Flow_BlockName" <text-blockname-from-flow-table>
```

For example, to create the PV(1): Generators, explained above, the statement would be:
2, Y, DefinePremise, Generators, PV, Partial_Devices, Device_BlockName, Generator

4.1.3.2 Filtering by Device/Flow Group Name

Similarly, a premise vector can be filtered by specifying a **device/flow group name**; the resulting PV's deepest level will be unchanged from the original premise vector's terms that correspond to device/flow instances that are members of that group, and set to false (0) otherwise. A group is similar to a block, except that its member device types are not necessarily listed as a contiguous collection in the Device/Flow table. In addition, it is specified differently within the Device/Flow table. While one specific column in the Device/Flow table is always devoted to the list of all block names, each group is specified by an individual parameter column, with 1s as parameter values, vs. blanks or zeros for non-member device/flow types.

```
<pr-filter-dfgrpname-3a> ::= <id-act-defpr-name> <pv-pvname> <df-groupname>
<df-groupname> ::= "Device_Group" <text-groupname-from-device-table>
| "Flow_Group" <text-groupname-from-flow-table>
```

For example, to create a PV(1): Essential_Loads,
(0,0,0,1,0,0,1),
based on a device parameter "Essential", in the example above, the statement would be:
3, Y, DefinePremise, Essential_Loads, PV, Partial_Devices, Device_Group, Essential

4.1.3.3 Filtering by Inequalities

4.1.3.3.1 FILTERING BY INEQUALITIES BETWEEN PVs

A premise vector can be filtered by specifying an **inequality** between it and another PV. This syntax can also be used to enforce a design rule.

```
<pr-comp-pv-pv-17> ::= <id-act-defpr-name> <pv-specificpv> <ineq> <pv-specificpv>
| <id-act-defpr-name> "Comparison" <pv-specificpv> <ineq> <pv-specificpv>
<pv-specificpv> ::= "PV" <pvname>
```

For example, we could ensure that all load with a single supply are powered by generators that supply only one load, by comparing the PV(1): Single_Source_Loads,

(0,0,0,1,1,0,1),

with the PV(1): Lds_Powd_by_1Load_Gens,

(0,0,0,0,0,1,0).

In the example above, the result would be the PV(1): Single_Supply_Consume_Lds,

(0,0,0,0,0,0,0),

since the only generator supplying only one load is PG2, and it powers DCLoad1, which is powered by both PG2 and APU1.

The statement to define this PV(1) would be:

4, Y, DefinePremise, Single_Supply_Consume_Lds, PV, Single_Source_Loads, =,
Lds_Powd_by_1Load_Gens

The statement to use this comparison to enforce a rule would be:

5, Y, DefineRule, , PV, Single_Source_Loads, =, Lds_Powd_by_1Load_Gens

FILTERING BY INEQUALITIES BETWEEN DEVICE/FLOW PARAMETERS/VARIABLES AND A QUANTITY

A premise vector can also be filtered by specifying an **inequality** between a device/flow parameter or variable, and a specific quantity.

```
<pr-filter-dfparvar-ineq-q-3> ::= <id-act-defpr-name> <pv-pvname> <df-parvar> <ineq> <quantity>
<df-parvar> ::= "Device_Parameter" <text-parameter-name-from-device-table>
| "Device_Variable" <text-variable-name-from-device-table>
| "Flow_Parameter" <text-parameter-name-from-flow-table>
| "Flow_Variable" <text-variable-name-from-flow-table>
<quantity> ::= "Constant" <realnumber-value>
| "Quantity" <qvname>
<qvname> ::= <name>
<ineq> ::= "<" | "<=" | "<>" | "=" | ">=" | ">"
```

For example, we could filter the PV(1): Essential_Loads above by specifying an inequality that the device's Availability must be ≥ 0.999999 . This could produce a PV(1): Hi-Avail_Essential_Loads: (0,0,0,1,0,0,0).

The statement to accomplish this filtering would be:

```
6, Y, DefinePremise, Hi_Avail_Essential_Loads, PV, Essential_Loads, Device_Parameter, >=, 0.999999
```

4.1.3.3.2 FILTERING BY INEQUALITIES BETWEEN TWO DEVICE/FLOW PARAMETERS/VARIABLES

A premise vector can also be filtered by specifying an **inequality** between two device/flow parameters or variables.

```
<pr-filter-dfparvars-ineq-4> ::= <id-act-defpr-name> <pv-pvname> <df-parvar> <ineq> <df-parvar>
```

For example, we could filter the PV(1): Essential_Loads above by specifying an inequality that the device's Actual_Availability (a device variable computed by examining the availability of sources and transmission paths) must be \geq its Required_Availability (a device parameter based on the role of the load in this particular mission). This could produce a PV(1): Avail_Loads: (0,0,0,0,0,0,1).

The statement to accomplish this filtering would be:

```
7, Y, DefinePremise, Avail_Loads, PV, Essential_Loads, Actual_Availability, >=, Required_Availability
```

Filtering by Logical Functions

One, two or three premise vectors can also be filtered by applying a **logical (Boolean) function** (monadic, dyadic or triadic) between them.

```
<pr-filter-logic-5> ::= <id-act-defpr-name> <pv-specificpv> "Function" <logic1>
| <id-act-defpr-name> <pv-specificpv> "Function" <logic2> <pv-specificpv>
| <id-act-defpr-name> <pv-specificpv> "Function" <logic3> <pv-specificpv>
<pvname>
<logic1> ::= "NOT"
<logic2> ::= "AND" | "OR" | "NAND" | "NOR" | "IF/THEN"
<logic3> ::= "IF/THEN/ELSE"
```

For example, we could find the PV(1): Non-Essential_Loads above by using the monadic NOT function. The statement to accomplish this filtering would be:

```
8, Y, DefinePremise, Non_Essential_Loads, PV, Essential_Loads, Function, NOT
```

4.1.4 Comparisons between Quantities

Most rules are created as comparisons between quantities. These comparisons could also be used to define premise vectors:

<premise-rule-comparison> ::= <pr-qvq-ineq-19> | <pr-qvqv-ineq -18>

Comparing a QV with a Quantity

A quantity vector can be compared with a quantity:

<pr-qvq-ineq-q-19> ::= <id-act-defpr-name> <qv-specificqv> <ineq> <quantity>
| <id-act-defpr-name> "Comparison" <qv-specificqv> <ineq> <quantity>

For example, we could find the generators with only one essential load, defining a PV(1): Gens_w/1_Ess_Load, by comparing the QV(1): nEss_Loads_per_Gen, (2,0,1,0,0,0,0),

with the constant: 1. The statement to accomplish this filtering would be:

9, Y, DefinePremise, Gens_w/1_Ess_Load, QV, nEss_Loads_per_Gen, =, Constant, 1

Comparing a QV with a QV

A quantity vector can also be compared with another QV:

<pr-qv2pv-ineq-qv-18> ::= <id-act-defpr-name> <qv-specificqv> <ineq> <qv-specificqv>
| <id-act-defpr-name> "Comparison" <qv-specificqv> <ineq> <qv-specificqv>
<qv-specificqv> ::= "QV" <qvname>

For example, we could ensure that all generators' available capacity meets or exceeds the power draw of their essential loads, defining a PV(1): Gens_w/_Capacity, by comparing the QV(1): Gen_Capacity, (90.0,90.0,65.5,0,0,0,0),

with the QV(1): Gen_Ess_Ld_Power,

(84.2,82.0,45.5,0,0,0,0).

The statement to create a PV(1) from this filtering would be:

10, Y, DefinePremise, Gens_w/_Capacity, Comparison, QV, Gen_Capacity, =, QV, Gen_Ess_Ld_Power

The statement to enforce this rule would be:

11, Y, DefineRule, , Comparison, QV, Gen_Capacity, =, QV, Gen_Ess_Ld_Power

Flow Pairing

A premise vector can also be created, or a rule enforced, by checking whether the collection of relevant flows have certain attributes, called their "pairing":

<premise-rule-pairing> ::= <pr-pv2yn-pairing-10.5>
<pr-pv2yn-pairing-10.5> ::= <id-act-defpr-name> <pv-specificpv> "Pairing"
<pairing-direction> <pairing-pattern>
<pairing-direction> ::= "Low_to_High" | "High_to_Low"
<pairing-pattern> ::= "Sequential" | "Concentric"

For example, for a jet-engine application, we could enforce a rule that the shaft-power flows from turbines to compressors go in a specific direction and are concentric (nested pairing).

The statement to enforce this rule would be:

12, Y, DefineRule, , PV, SP_Flows_from_T_to_C, Pairing, High_to_Low, Concentric

User-Defined Filtering

A premise vector can also be created, or a rule enforced, by checking whether the collection of relevant flows satisfy certain criteria, according to an external function call:

<premise-rule-user-defined> ::= <id-act-defpr-name> <pv-specificpv> <user-defined-function><udf-arg-list>
<user-defined-function> ::= <text-name-of-an-external-function-call>
<udf-arg-list> ::= <comma-separated-text-list-of-arguments-to-user-defined-function>

For example, for an electrical circuit partial architecture, we could enforce a rule that prohibits electrically “shorted” components (those with a direct very low-impedance path across their leads).

The statement to enforce this rule would be:

13, Y, DefineRule, , PV, Electrical_Circuit_Flows, No_Shorts, ,

4.2 Defining Quantities

There are also multiple types of statements to define quantities.

<define-quantity> ::= <quantityv-defn> | <quantityv-reduction>

Mathematical Functions between Quantities

A quantity vector can be created using monadic or dyadic mathematical functions:

<quantityv-defn> ::= <q-parvar2q-11> | <q-qv-math-12> | <q-qvq-math-13>
 <q-qv-math-12> ::= <id-act-defq-name> <qv-specificqv> "Function" <math1>
 | <id-act-defq-name> <qv-specificqv> "Function" <math2> <qv-specificqv>
 <id-act-defr-name> ::= <stmt-id> <y-n> <def-rule> <pvname>
 <id-act-defq-name> ::= <stmt-id> <y-n> <def-quantity> <qvname>
 <id-act-defq-vname> ::= <stmt-id> <y-n> <def-quantity> <text-variable-name-from-flow-table>
 <q-qvq-math-13> ::= <id-act-defq-name> <qv-specificqv> "Function" <math2> <quantity>
 <def-quantity> ::= "DefineQuantity"
 <math1> ::= "SIN" | "COS"
 <math2> ::= "+" | "-" | "*" | "/" | "MIN" | "MAX"

Device or Flow Parameters and Variable Quantities

A device/flow parameter or variable quantity can be used to define a QV.

<q-parvar2q-11> ::= <id-act-defq-name> <pv-pvname> <df-parvar>

Device or Flow Parameters and Variable Quantities

A quantity vector can be reduced in dimensionality to create another lower-dimensionality QV by reduction using a mathematical function.

<quantityv-reduction> ::= <q-reduce-math-14>
 <q-reduce-math-14> ::= <id-act-defq-name> <qv-specificqv> "Reduce" <math2> <target-depth>

Assigning Quantities to Variables

A quantity vector’s or premise vector’s terms can be assigned to the corresponding device/flow instances’ variable values.

<quantity-to-parvar> ::= <q-qv2var-15> | <q-pv2var-16>
 <q-qv2var-15> ::= <id-act-defq-vname> <qv-specificqv> "QV_to_Var"
 <q-pv2var-16> ::= <id-act-defq-vname> <pv-specificpv> "PV_to_Var"

Example Flow-Rule Statements

The following is a set of flow-rule statements used as input to AEE to enumerate Layer 1 full-system architectures for an aircraft engine.

List of Acronyms, Abbreviations, and Symbols

Acronym	Description
AEE	Architecture Enumeration and Evaluation
CBD	Contract-Based Design
CFD	Computational Fluid Dynamics
CLP	Constraint Logic Programming
DSM	Design Structure Matrix
FEM	Finite Element Modeling
NRE	Non-Recurring Engineering
PBD	Platform-Based Design
UAV	Unmanned Aerial Vehicle
UTRC	United Technologies Research Center

