Proceedings of the 2009 Design Engineering Technical Conference &
Computers and Information in Engineering Conference
IDETC/CIE 2009
August 30 - September 2, 2009, San Diego, USA

DETC2009-87057

A FORMAL REPRESENTATION OF SOFTWARE-HARDWARE SYSTEM DESIGN

Eric Walkingshaw, Paul Strauss, Martin Erwig
School of EECS
Oregon State University
Corvallis, OR 97331

Jonathan Mueller, Irem Tumer[*]
School of MIME
Oregon State University
Corvallis, OR 97331

ABSTRACT

The design of hardware-software systems is a complex and difficult task exacerbated by the very different tools used by designers in each field. Even in small projects, tracking the impact, motivation and context of individual design decisions between designers and over time quickly becomes intractable. In an attempt to bridge this gap, we present a general, low-level model of the system design process. We formally define the concept of a design decision, and provide a hierarchical representation of both the design space and the context in which decisions are made. This model can serve as a foundation for software-hardware system design tools which will help designers cooperate more efficiently and effectively. We provide a high-level example of the use of such a system in a design problem provided through collaboration with NASA.

INTRODUCTION

Software/hardware systems are becoming increasingly complex and prevalent, tremendously impacting our lives. Systems in areas ranging from security and safety to health care and personal devices consist of hardware and software components that must cooperate to perform accurately. Autonomous systems depend heavily on software for planning actions, monitoring and diagnosing general system health, and executing critical commands.

Unfortunately, the growing dependence on software-intensive designs also means vulnerability when systems fail to function in critical situations. There have been many unfortunate cases to demonstrate that even minor errors in designs can have a big impact on the functioning of the artifact, in some cases with disastrous consequences. Many examples of spectacular software failures have been reported that draw our attention to the importance of *good design*, including the high-cost and high-visibility mishaps of the Patriot Missile Defense System in 1991, the Ariane 5 rocket in 1996, and the loss of the Mars Climate Orbiter in 1999, to name only a few [9].

What makes one design successful and another one fail? How can we judge the impact of a particular set of decisions on a design? Why were the design decisions made? Being able to answer these kinds of questions is an important prerequisite to effective and systematic design of complex software-intensive systems. In this paper, we present a methodology that can support designers in working with such complex hardware/software systems, by helping them *capture when and why the decisions were made* and *understand the consequences of these design decisions*. Such support can be ultimately achieved only if the formal model provides general knowledge that explains hardware/software system design and the design process.

Concurrent Design of Software and Hardware

Ensuring that such software-intensive systems operate properly requires that the software and the hardware are compatible with each other, reinforcing the need for concurrent design. Concurrent design environments have become common in industries that deal with large scale complex systems as a means to tackle the problem of generating requirements and producing conceptual designs that consist of multidisciplinary subsystems (e.g., Team X at NASA Jet Propulsion Laboratory) [38, 58]. Unfortunately, a true concurrent design approach between the hardware and software developers is not easily realized. The two sides of development can often find themselves disconnected, resulting in

[*]Address all correspondence to this author: irem.tumer@oregonstate.edu

them being developed completely independent of each other and put together at the end. The problem with this approach is that, because there is a lack of understanding between the software and hardware sides of development, dependencies are not fully understood and incompatibilities may exist. Testing the system can discover incompatibilities and changes can be made to ensure proper system performance. However, this can be a difficult task, and become very expensive due to changes to the system being implemented late in the design process.

In this research, we draw direct parallels between engineering design and software design, with the purpose of bridging this gap. In particular, this paper presents a formal representation of the design process to help bridge the information gap between software and hardware system designers.

Observations about Design

In the world of engineering design, a "physical design" can have many representations during the different phases of design. A concept can be described as functions and their interactions semantically (for example, verbal or textual representation of a product) [64], using a graphical representation (for example, a functional decomposition) [49], preliminary sketches to generate concepts/ideas [48], models to analyze how the design works (stress analysis, failure analysis, vibration analysis, etc.) [15], and/or detailed drawings that show how the parts/components fit together to meet a need and/or a physical representation of the product (for example, a prototype) [48].

In contrast, "software design" is not very well defined. Very generally, a software design is a plan for implementing a software system, usually by representing the individual parts of the system and their arrangement [1, 11, 13].

The process of design is one of decision making, that is the progression from the initial need (the design problem) to a final product [48, 64]. From this point of view, software design is also understood as a problem-solving process [13]. In this paper, we explore existing commonalities between software and hardware system design. In particular, we make the following observations about design in general that are common to both software and hardware systems.

1. Any software or hardware artifact is an embodiment of a set of *design decisions*. Each of these decisions affects a particular aspect, or *attribute*, of the design.
2. Designs, design attributes, and design decisions, are abstract ideas that must be given in some *concrete representation* when they are to be communicated. In general, multiple representations, which might differ in usability, exist for particular (combinations of) design attributes.
3. The function of designed artifacts are always given within some *context*. The context represents requirements and constraints for the artifact.

For physical objects, the idea of design attributes is obvious: A physical object has a certain form, color, size, etc. In addition to these almost universal attributes, objects can have many more specialized attributes, such as number of wheels or doors, and also attributes that describe functionality, such as maximum speed of a car or resolution of a digital camera.

The fact that some attributes are derived from others, such as weight, indicates that attributes are generally not independent of one another but often related through constraints. Different attribute representations are also apparent. For example, attributes can be given as textual descriptions, such as "color:red" or "height=5in" or in graphical form. Contexts are typically represented as constraints to restrict the large space of design alternatives [19, 24, 70].

For software, attributes are of a more abstract nature, including the types of input and output or the employed algorithm. As in the case for physical objects, some attributes are derived, for example, the runtime efficiency of an algorithm. Different representations for software attributes have been investigated in the area of software visualization [6, 35, 55]. For example, module relationships can be displayed as graphs instead of text. Employing alternative representations for creating/modifying software has also been investigated, such as syntax trees [10, 53, 54], type derivations [68, 69], or abstract data types [21–23]. In particular, *context* in software is often neglected and is seldom explicitly modeled or represented. The approach of problem frames [31] makes a strong case for representing software design contexts.

Contributions

In this paper, we exploit these observations to develop a general representation of the design process. Our model captures *design decisions* on individual *attributes* of a larger *design object*. The structure of this object defines the entirety of the design space. This representation can be employed to represent and study a variety of design domains and to investigate the design process within these domains.

The design object is generated dynamically, as it is explored, through the use of a hierarchical library of *rules*. These rules, which describe both the local and cross-cutting impact of design decisions, collectively describe the context of the design. This modular definition of the design context allows the creation of a sharable repository for constructing design spaces, facilitating standardization and reuse.

Finally, design decisions generate and maintain associated *annotations*. Annotations contain arbitrarily complex information and provide an explanatory record of a decision—why it was made, and how it affected the design space.

These contributions can be employed in many ways, such as representing and studying design domains directly, measuring differences and similarities between design alternatives, creating reusable design repositories, and developing a variety of tools to support the design process.

In what follows, we first present related work that addresses formalisms in both hardware and software design. We then present our model of the design process, followed by an application to a research rover example at NASA that illustrates how the model could be utilized by designers of software-intensive systems.

RELATED WORK

Many in the engineering design research community have focused on developing formalisms that help model and hence improve the design process. Three areas of direct relevance are: (1) the use of examples and reuse to generate new designs; and the efforts in (2) modeling the decision making process to help generate alternatives for new designs, and, (3) modeling the design space to help narrow the search for "best" designs.

Formalisms in Reuse and Example-Based Design

The reliance on examples is often the starting point when designing a new artifact, be it software or a physical product. In fact, most design is done by "redesign", based on reusing previous concepts, models, ideas, exploring commonalities amongst products, functions, and failure mechanisms, and drawing analogies with other products and domains [2,47,61,62].

There have been numerous efforts in the engineering design research literature to formalize the process of designing by examples during conceptual design [48,49]. There are obvious reasons for this common practice. In many cases it is much faster, easier, and more cost-effective to modify an existing artifact than to build a completely new one. For commercial companies, huge savings can be accomplished by exploring commonalities among products, and building common platforms for reuse and mass customization. Product platform design can greatly improve response to new customer requests, reduce design and manufacturing costs, improve reliability and failure tolerance, and improve time to market [18,34,37,59].

The idea of designing by example has been explored in the realm of software as well. The research on *software reuse* dates back almost 40 years [41]. As a result, the software design literature is replete with efforts to provide software reuse guidelines and procedures for taking advantage of commonalities, modularity, etc. [14,25,26,32,71]. The idea of exploring *commonality* and *variability* amongst software has also been suggested [17]. Note that "programming by example" is a different idea that refers to approaches to infer abstract programs from concrete examples [45].

A particularly simple, yet widely used, approach to software reuse is *code scavenging* [36] (also called *copy-paste* or *copy-edit*), in which code fragments are sought, copied, and then modified. Several studies have shown that the implementations of Linux, FreeBSD, MySQL, PostgreSQL, and the X Windows System all consist of 20-30%, sometimes even of up to 60%, duplicated code [4,33,40,52,63]. Code reuse essentially involves the following two steps. (1) Finding a suitable code to start with, and (2) modifying the code. In many cases the first step is trivial or not even needed. For example, one aspect of the so-called *agile methods* is to deliver working versions of software products frequently [16,30]. The focus is getting a working program within a fixed time frame and to reiterate to add or modify functionality. Therefore, agile methods constantly apply code reuse to the last released prototype.

Despite its great promise and much research efforts, software reuse has had only limited success so far [8,42,43], in part because of the difficulties with finding suitable code as a basis for reuse and because of the lack of structured approaches to modify reused code.

In summary, the use of examples in design is prevalent, but it is not always well supported by methods and tools. In this paper, we present a general representation of the design process for hardware and software design alike, which we plan to use to devise a design-by-example methodology. In particular, the ability to track these decisions will allow designers to have a better understanding of the system and enable the reuse of hardware and software more effectively by being able to go back and determine what decisions were made and why.

Formalisms in Decision Making in Design

Decision making has been recognized as an integral part of the engineering design process in all phases of design and the role of decisions during the concept generation phase of early abstract design has been the focus of many research efforts. Formal research in the engineering design community has focused on improving this process through the application of rigorous mathematical principles from decision theory [39,65], which is largely based on concepts from game theory, utility theory, voting, and preference modeling, and has its roots in decision science, economics, and operations research [27,56]. In an effort to introduce this formalism into engineering design many design researchers have worked on finding utility functions and preferences that work for the engineering design process [5,28,50,67].

These mathematical approaches were explored to satisfy the need for establishing the "mathematics of design" [29]. They offer a sound way to handle vast numbers of decisions using statistical models to account for uncertainty, risk, information, and preferences. However, most research in decision modeling assumes that designers (decision-makers) have a pool of design alternatives ready to explore for examination, evaluation, comparison, and selection. Incorporating these decision theory principles into the actual alternative generation process is not addressed thoroughly. Decision making takes place throughout the typical design process for physical artifact design. In the early stages, there is much uncertainty about the design and the requirements to enable to selection of the "best" design. During this stage, decisions help guide the design by eliminating options that are not viable [72]. Generating these alternatives costs significant amounts of labor and money to companies, and as a result, methods to facilitate the automation of alternative generation are in great demand.

In summary, many in the engineering design community have questioned the formalization of such an abstract phase of design using these mathematical approaches, because they offered more of a black-box approach and did not help with the actual process of generating concepts [39]. Our fine-grained model based on attributes offers a new, non-black-box alternative.

Formalisms in Design Space Exploration

Since the design process essentially consists of repeatedly making design decisions, any designed system is a manifestation of a set of such design decisions. However, the decisions that lead to the "best" designs have the potential to result in a very large number of alternatives, which raises questions about how to search the large design space to find an optimal solution. One idea that has been explored heavily is set-based design. The idea behind *set-based design* is to represent design alternatives in sets during the design process. These sets are then gradually narrowed down to the "best" solution as the design evolves [60, 66]. A derivative of this concept is set-based concurrent engineering, where sets of designs are first developed for a given design problem, followed by an inter-team and intra-team communication by comparing the sets of alternatives, looking for regions in the design space that overlap in their design alternatives [7, 60].

While powerful as a concept, the existing research on set-based design does not bring us closer to our goal of modeling the fundamentals of making decisions from a set of design alternatives in the large design space, because these approaches store the whole design in sets and do not maintain sets of individual attributes. Formalisms that focus on design-space exploration and global optimization [46, 57] and representing imprecision during decision making have been introduced and pursued successfully based on the mathematics of set-based theory [3]. Formalisms that establish the mapping from performance to design space have also been introduced based on the concepts of information systems and decision tables using the rough set theory, in an effort to help with the optimization and trade space exploration [57]. Finally, set theory has been used in efforts to find the best designs for computer-aided design (CAD) in the form of set-based parametric design, which combines the set-based concurrent design practice with the parametric modeling technique widely used in most 3D-CAD systems [44].

In summary, these formalisms lay the groundwork for developing a model of how decisions are made when exploring the large sets of alternatives in the design space, and as such, contribute to our general understanding of the process of making design decisions for hardware-software systems.

A FORMAL MODEL OF THE DESIGN PROCESS

In order to facilitate the development of better design systems we define a model of the design process. Using this model we can define the structure and content of the design space, formalize what it means to make a decision, and describe the evolution of the design space through the constraints and decisions which refine it. In the following subsections, we introduce the model in a roughly bottom-up fashion. We begin with the observation that an artifact is an embodiment of individual design decisions. We first define decision making and introduce attributes, the fundamental element on which a decision is made. Building on this, we next develop a hierarchical representation of design objects, and we define the design environment through a hierarchical library of rules. Finally, we extend the model to include annotations, a means of relating decisions to the environment and outside world.

Attributes and Decisions

Conceptually, the *design space* of an artifact is the range of all possible designs of that artifact, and is thus of both arbitrary breadth and arbitrary density. In order to provide structure to this space, we need some fundamental design unit, which we call an *attribute*. Each attribute A has an *identifier* i and an associated *domain* D, which we write as $A = i : D$, where i is a string and D is a (possibly infinite) set. For example, the design of a car might include an attribute *exterior-paint-color* $: C$, where the set C represents all possible colors. Elements of an attribute's domain, such as individual colors in the example, are referred to as *values*.

A *decision*, in the most basic sense, is the constraining of an attribute's domain. More formally, a decision $i : D \succ D'$ is an operation on attribute $A = i : D$ that changes the domain of A to D', where $D' \subset D$. An attribute whose domain still contains multiple elements is an *unresolved attribute*, and elements in its domain are called *potential values* or, where linguistically appropriate, *choices*. An attribute which has been fully constrained, such that its domain contains only one value, is said to be a *resolved attribute*. The remaining value of a resolved attribute is simply called its value except in cases where we must distinguish from values associated with unresolved attributes, in which case we call the values of resolved attributes *selected values*. Similarly, a decision which converts an unresolved attribute into a resolved attribute is said to *select* the remaining value, regardless of whether this selection is performed explicitly or as a result of eliminating all other potential values.

Note that it is not possible for a decision to convert a resolved value into an unresolved value. This reflects the notion that decision making is a directed process, with the goal of reaching a concrete design within the design space; a decision always makes progress towards this goal. In practice, designers may decide to undo or re-evaluate decisions, resulting in the conversion of resolved attributes to unresolved attributes. Such decisions can be classified as *meta-decisions*, higher-level decisions which operate on decisions rather than attributes. These meta-decisions are not considered here, but are supported by the introduction of annotations.

Attributes are the building blocks of design space modeling, while decisions provide a fundamental means of navigation. In the next section we show how attributes can be combined to form higher-level design objects which capture the structure of both the artifact to be designed and the implicit structure of the decision process itself.

Structure of Design Objects

A *design object* O refers to the formal characterization of a specific design space, and can be defined, very simply, by a set of attributes $\{A_1,...,A_n\}$. However, the structure of O is not flat, since attribute values (potential or selected) may be design objects in their own right. Below is a simple formal definition of design objects. The domain of an attribute is given by a set of values V and no syntactic distinction is made between resolved and unresolved attributes; the lower-case v refers to a terminal value, that is, a value which is not a design object and thus represents a leaf of the resulting *design tree*.

$$\begin{aligned} O &::= \{A_1,...,A_n\} \\ A &::= i : \{V_1,...,V_n\} \\ V &::= v \mid O \end{aligned}$$

The terms "design object" and "design tree" both refer to this data structure, but are used in different contexts where convenient and to emphasize the structure's role in these contexts.

Note that two different types of branching occur within a design tree. *Object branching* refers to branches in the O production rule, where each branch corresponds to an attribute within the design object. Returning to the car design example from the previous subsection, the top level of a car design might contain the attributes "transmission" and "interior" (among others). Potential values of the transmission attribute would be design objects in their own right, corresponding to the design of a particular transmission. Similarly, values of the interior attribute would be design objects corresponding to the design of car interiors. Thus, object branching captures structure that exists within the artifact itself.

Decision branching, on the other hand, refers to branches in the A production rule, where each branch represents a different potential attribute value. Our transmission attribute from above, for example, would likely contain at least two choices, corresponding to designs for both manual and automatic transmissions. While object-branching captures the structure of the artifact, decision branching captures the implicit structure of the decision space—the structure of the decisions which must be made to attain a fully resolved design object. The hierarchical structure of the decision space implies that certain decisions (those lower in the tree) are contingent upon other decisions (their ancestors higher in the tree). Although it may be possible in some cases to make a decision in a subtree before making a decision in its unresolved parent attribute, that decision is only conditionally defined on its subtree being selected in the parent attribute.

Until now we have made a few assumptions about the nature of the design space and the scope of impact of a particular decision. In the next subsection we challenge those assumptions and provide a solution through extensible sets of rules which refine the decision space by performing validation and dynamically altering the design tree.

Rules and Actions

Representing the design space as a tree-like structure seems to imply a couple of important but ultimately invalid assumptions: First, that it is possible to statically model the entire design space. Second, that the impact of a decision is confined to the attribute on which it was made.

The first assumption is clearly unrealistic. The design space is rarely completely known at the outset of the design process, and even if it is, relatively small examples can produce design trees which are prohibitively large to represent, and may recurse infinitely. Therefore, we must be able to generate parts of the design tree lazily, on demand. Additionally, different parts of the design tree often have subtrees in common. We must be able to abstract out these commonalities to avoid duplication.

The second assumption, that decisions only have local effects, does not allow for the existence of cross-cutting constraints. Returning to the car design example, suppose the car design used has budgetary restrictions; the decision to use an expensive part in one part of the car limits our choices in other parts of the car since less money remains in the budget. Thus, the decision space is not strictly modularized and hierarchical as we have modeled it. We need a way to represent the non-local implications of some decisions.

As a solution to all of these problems, we introduce the concept of design-guiding *rules*. Rules exist independently of the design object, as part of the environment within which the design object exists. They are triggered by decisions within the design object, and affect change on that design object. More precisely, a rule consists of two parts which address the following questions: Given a decision and the current design tree, does this rule apply? If this rule applies, what *action* should be carried out?

More formally, a rule R is a pair of functions $(apply, action)$. *apply* addresses the first question, of whether a rule applies to a particular decision, and has the corresponding type $(i : D \succ D', O) \rightarrow Bool$. *action* is a tree transformation with type $O \rightarrow O$ and describes changes to the design tree as a result of applying this rule. Additionally, the action component can potentially have side-effects, for example, providing a notification that the rule was triggered, or writing to a log file.

Actions can take many forms. One of the most basic actions is to simply extend the decision tree by replacing a particular leaf value with a decision object. This allows the designers to generate the decision tree as particular subtrees become relevant. It also provides a means of abstraction, by moving the definition of recurring design objects into rules. This allows arbitrarily large game trees to be defined lazily, as well as providing a way to abstract common subtrees into a single rule.

Other rules address the need to define constraints on the design. This could include cross-cutting constraints like budget or weight limitations, or more specialized constraints—for example, if a manual transmission is selected, we must also select an interior which is compatible with the inclusion of a clutch and gear stick. Rules which express these sorts of constraints may include actions which transform the tree accordingly (e.g. by removing all interiors which violate the constraint) or if such a transformation is not possible (e.g. if the designer has already chosen an incompatible interior) an action may notify the designer of the constraint violation. In the next section, we discuss the proposed design system from a user's perspective and includes a more detailed look at some rule actions which require user interaction.

After any decision is made, all of the currently included rules are checked (by applying the *apply* function of each to the new decision), and the actions of any matching rules carried out. Thus, rather than being determined by a statically defined tree, the design space is determined by a combination of the initial design tree and a set of rules which define how that tree will change as decisions are made. Since the number of rules can grow quickly, they can be organized into hierarchical libraries, easing rule management and promoting rule reuse between designs. The next section provides an example hierarchy of rules in the context of the design of a program for a NASA rover.

Additionally, we would like to be able to dynamically add and remove rules from the model. This would allow designers to begin work on a design without complete knowledge of the design space, and it would provide a way for designers to respond to external change, such as changing requirement specifications or resources. However, added or removed rules may have had an impact on past decisions and subsequent transformations—how do we take this into consideration? We could simply consider the new rule set to affect only subsequent decisions. This may be desirable if we want to modify a rule only temporarily (e.g. to circumvent a constraint in a special case) or have rules which apply only at certain points in the design process. It seems like both of these scenarios, however, could be avoided with careful rule definitions. It seems far more likely that we will want to reconsider all past decisions involving changing rules. For this, and other reasons, each decision also maintains a list of associated annotations.

Annotations

Previously, we defined a decision as simply the process of constraining an attribute's domain. The ultimate goal of the model, however, is not to simply transform a decision tree into a concrete artifact through a sequence of decisions, but also to provide a record, or an explanation, of why decisions were made and how the decision tree was affected. Thus, each decision also maintains a corresponding list of annotations, each of which contains information about the decision.

At the lowest level, all rules which were applied when a particular decision was made are saved as an annotation to that decision. This allows us to easily go through and re-evaluate decisions that correspond to changing rules, as discussed above. On a much higher level, a user might want to supply a free text comment about why a particular decision was made. The Related Work section discussed various ways of formally modeling design requirements; structured annotations could relate decisions to these more traditional engineering models as well.

This approach is similar to that found in [51], which associates rigidly defined metadata with each decision in a design tree. This metadata contains information providing, e.g. justifications for a decision and/or design alternatives considered. By allowing arbitrary annotations our approach is somewhat more general, but could incorporate this previous work as just one of many different types of annotations.

Annotations provide a link from within the model to the outside world. At the lowest level, a decision provides links to the rule environment it is working in. At higher levels annotations provide links to design documents and the designers themselves. These links are critical to understanding the design process.

APPLICATION: A NASA RESEARCH ROVER

The United States and the world have gained a vast amount of knowledge from the work done at NASA. NASA continues advancements in developing and applying new technologies through research into developing new rovers for future space exploration. Spirit and Opportunity, two current rovers from NASA, have done a great job of collecting data and providing us with information about Mars. They have performed beyond expectations but they do have some limitations. The rovers have limited intelligence that constricts the actions of the rovers. It takes three Martian days to be able to get the rover to an inspection point and to get an instrument correctly positioned on the spot to collect data.

To address this limitation and other problems, NASA has a research rover program to integrate and test new technologies that will help them meet the goals of future space missions, such as the Mars Space Laboratory (MSL), which is scheduled to launch in the Fall of 2009. The goal of the program is to build a reliable and autonomous rover. It is necessary to have autonomous placement of instruments to acquire samples, determine mineralogy, obtain microscopic images and other operations to understand the geology of a planet. By implementing advancements in technology and creating a more autonomous rover, NASA hopes to complete sampling and testing more efficiently. To help with this goal, two research rovers have been created for testing of new technology: the K9 and the K10 rovers, discussed briefly next, followed by a discussion of the software designed to help with the autonomous operation of these rovers.

The K9 and K10 Rovers

The K9 rover was developed jointly by NASA Ames and NASA's Jet Propulsion Laboratory (JPL). The K9 is a six-wheeled, solar-powered rover that was modeled after the Field Integrated Design and Operation (FIDO) rover that was designed by JPL in the late 1990's. The rover features several hardware components including a compass, an inertial measurement unit, three pairs of monochromatic cameras, solar tracker, and pair of stereo cameras. There are several other components and possible attachments for the rover [12].

The K10 rover was developed as a successor to K9 and was designed to be a less expensive and easily upgradeable rover. The K9 features several specialty components and is expensive to maintain whereas the K10 was designed using as many commercially available off the shelf components as possible to address reliability and cost concerns. The K10 has a four wheel steer and drive chassis, digital 3-axis compass, GPS unit, 2D laser scanner, stereo cameras, panorama camera, laptop computer, and several other components. The K10 reused several pieces from the K9 to take advantage of proven technology and to also reduce costs in development. The electrical system and several hardware components including the cameras were carried over from the K9.

The rovers are controlled by means of autonomy software developed at NASA Ames. To enable autonomous operation, the rovers need to be able to know their position. The K9 accomplishes this by using a compass with a sun tracker. The K10 rover upgraded to a 3-axis compass to be able to capture all the measurements necessary so that a sun tracker does not need to be used. However, this upgrade resulted in a failure. During testing, the 3-axis compass proved to be a problem and designers later changed to a compass and sun tracker setup similar to what the K9 used. The rover program at NASA is a relatively small project compared to a NASA mission and features a relatively small staff. Even with a relatively small development team, contact between the hardware developers and software developers is limited. Changes to the hardware and software are generally done independently of each other and problems are identified and corrected through testing.

PLEXIL and The Universal Executive

One of the field demonstrations for the K9 rovers accomplishes the autonomous control of the rover with the help of PLEXIL, which is an execution planning language designed at NASA [20]. The main building block of a PLEXIL program is called a *node*. Each node contains a single action, which can be one of the following seven types:

Command: A request to be executed on the external hardware system
Function: A complicated calculation to be calculated outside of PLEXIL
Assignment: Sets an internal variable within a PLEXIL program
List: A collection of sub nodes for grouping similar nodes and allowing multiple actions
Update: Sends output data to the external system, like status information for users
Plan Request: Sends a request to design support asking for a new plan
Empty: No other defined action

Nodes are inherently executed in parallel. A programmer can affect this by using *node conditions*. These conditions specify when the node's action executes, and, in the case of list nodes, when its children's actions should be started and completed. PLEXIL programs are executed through a system known as the Universal Executive (UE). UE is a lightweight interpreter designed to run PLEXIL programs on a variety of automated hardware systems at NASA. PLEXIL programs are designed to schedule commands, and the UE uses these programs to send the commands to the hardware system.

SYSTEM DESIGN USING THE FORMAL MODEL

In this paper, we want to view the design decisions made in software-hardware systems as they are being developed. Our example will focus on the creation of PLEXIL programs for the K9 and K10 rover design. We will show that keeping track of decisions made during the design phase can prevent issues when connecting software designs to hardware components.

Use Scenario

We envision a programmer sitting and programming within a *design system*. Working within this system would be similar to working within a modern IDE (integrated development environment), and in fact, would likely be implemented as an extension to an existing IDE. The programmer enters PLEXIL code into a text editor which is automatically compiled and analyzed at certain intervals, for example, when the user saves the file they are working on. When we refer to the design system (e.g., "the system alerts the user..."), we mean the application that the user is working in, but more specifically, the extensions related to the theory of design decisions.

Crucial to the scenario here is that the *design space* of the system at any given time is specified by a hierarchical library of *rules*. Each rule is composed of two parts which answer the following questions: Given the current state of the *design tree*, does this rule apply? If this rule applies, what *action* should be carried out?

A design tree corresponds nearly exactly with the PLEXIL syntax tree. An action can take one of the following forms:

1. Alert the user that a rule was matched by providing a relevant message, usually indicating a problem with the design.
2. Alert the user to the problem and suggest a *solution*.

3. Automatically apply a solution and possibly alert the user.

A solution takes the form of a tree transformation which resolves the problem identified by the rule by modifying the PLEXIL code. For example, a solution could specify adding a predefined node to an existing node's list of children.

When considering the implementation of these actions, alerts can be either *active* (e.g., a popup window) or *passive* (e.g., a line in a specific pane or log file). It may be best to follow the precedent set by other tools common in IDEs like compilers and style checkers and implement only passive alerts, allowing the user to triage and address these at their discretion.

It is also worth noting that the information required to specify a rule utilizing either the second or third type of action is the same. In both cases the design system must be able to detect the problem and provide a solution—the only difference is whether or not the system automatically applies that solution. The first type of action only requires the design system to be able to detect the problem. It will probably be the case that all three types of actions will be utilized in different places in the rule hierarchy depending on the severity and complexity of the problems addressed by each rule.

Notes on colors and fonts

In each figure that follows, nodes added or modified since the last figure are colored green if they were added or modified by the user, red if they were added or modified by the design system. The rule hierarchy is shown to the right or below any graph containing a red node. The level containing the rule that generated the red node is also colored in red.

In the text, low-level commands available to PLEXIL are indicated in `monospaced` font. PLEXIL node and variable names are indicated by *italics*.

Scenario Overview

A NASA programmer has the simple task of writing a PLEXIL program to drive a K9 rover forward some amount. The programmer creates a very simple program to do this, which the design system helps augment to handle the case where a wheel is stuck.

The programmer then adds some code to maintain a measure of the distance traveled. The system again helps to improve the robustness of the system, this time detecting and providing a handler for a potential error with the Sun Tracker unit.

Later, a new rule is added to the system which ensures that any time the rover drives, it handles the case where a wheel is slipping (i.e., spinning, but not getting traction). The code is analyzed and the system alerts and suggests solutions for newfound potential holes in the design.

This latest change triggers a code readability-related rule which suggests some simple code maintenance changes.

Finally, the K9 rules will be switched out for the K10 rover as the hardware is updated. The new rules will trigger a change in the code to conform to the new hardware.

The rule hierarchy in use is given in Fig. 1. At the base are rules concerned with PLEXIL syntax and style. Next in the hierarchy are general NASA-wide rules that apply for any vehicle or mission. Next are rules specific to the K9 rover. At the top are rules that apply specifically to the mission at hand—in this case, driving forward.

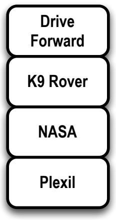

Figure 1. The rule hierarchy for this mission

Scenario Implementation

First, the user adds a single PLEXIL node named *Drive* as shown in Fig. 2. The *Drive* node contains a call to the lower-level `Drive` command. A rule in the K9 level of the rule hierarchy specifies that any call to `Drive` must ensure that the wheels are not stuck by including an appropriate monitor as a sibling of the calling node. The system proposes a resolution to this rule by asking the user if they would like to include a predefined *Stuck Monitor* node. The user accepts, resulting in the graph shown in Fig. 3.

Figure 2. User adds drive node

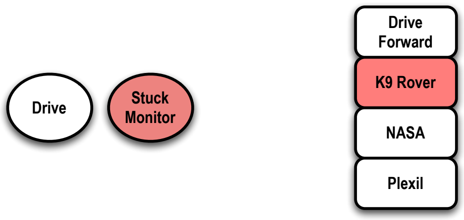

Figure 3. System adds stuck monitor

The result of this addition, however, is a syntactically invalid PLEXIL program. The rules are rechecked automatically and a rule in the PLEXIL level matches the current tree. The rule specifies that a program must have a single root node. *Parent Node* is automatically created and the two existing nodes are added as children (see Fig. 4). The user gives this newly created node a slightly more meaningful name, *Drive Program* (Fig. 5).

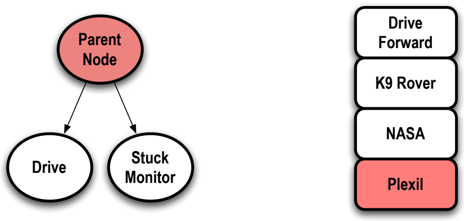

Figure 4. System adds parent node

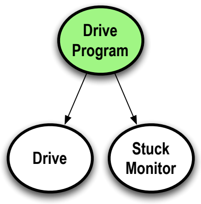

Figure 5. User renames parent node

The specification indicates that the program must also maintain a total distance traveled. The programmer accomplishes this by adding three new variables and two children nodes to *Drive Program* (see Fig. 6, variables not shown). The new variables are *startPosition* indicating the location of the rover before moving, *endPosition* indicating the location of the rover after moving, and *distance* indicating the total distance traveled by the rover.

The first child of *Drive Program* is *Pre Drive*, a node which will be executed before each execution of *Drive*. It has one child, *Start Position*, which checks the Sun Tracker and stores the current position to the *startPosition* variable.

Post Drive is executed after each execution of *Drive*. It has two children. The first, *End Position*, checks the Sun Tracker and stores the new current position to *endPosition*. *Update Distance* increments *distance* by the difference between *endPosition* and *startPosition*.

In this example, the program reads the starting position from the Sun Tracker while simultaneously sending a command to start the rover driving. When the `Drive` command is complete, the Sun Tracker again reads the rover's position, which can later be used to calculate the distance travelled in a later node *Update Distance*. This code will continue to be executed until another node sets *keepDriving* to false, forcing a repeat condition (not shown) to evaluate to false and ending program execution.

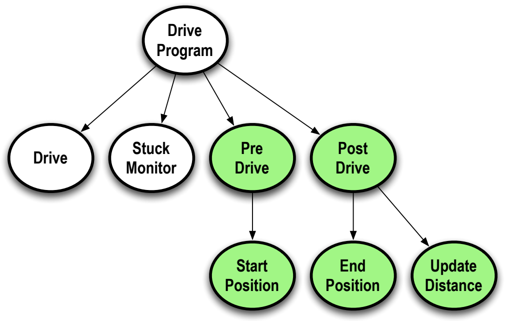

Figure 6. User adds nodes that use Sun Tracker

After these changes are saved the design system automatically analyzes the new program. A rule in the NASA level of the hierarchy recognizes that we are using a Sun Tracker and suggests the addition of a new node, *Sun Tracker Monitor*, to our program. The programmer accepts. With the addition of *Sun Tracker Monitor*, the program will only let the program run as long as the Sun Tracker is on. Since the Sun Tracker is used both in *Start Position* and *End Position*, the monitor is added to the first common ancestor of these two nodes (see Fig. 7).

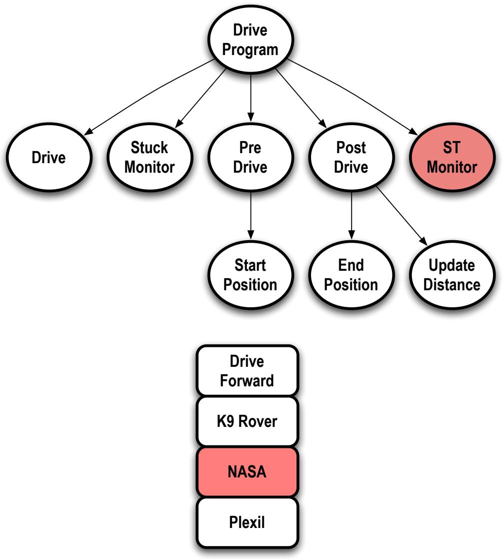

Figure 7. System adds Sun Tracker Monitor

The programmer submits the completed program and moves on to another project. Sometime later, field-tests reveals that sometimes K9 wheels will spin without getting traction, a problem that all existing programs for K9 rovers that call `Drive` should handle.

A new rule is added to the K9 level of the rule hierarchy and the design tool is run against all existing K9 programs, including the one the programmer submitted long ago. The rule detects the `Drive` call in this existing program and no existing slip monitor, so the *Slip Monitor* node is automatically created and added to the program (see Fig. 8) and a notice sent to the programmer requesting that they verify the modification.

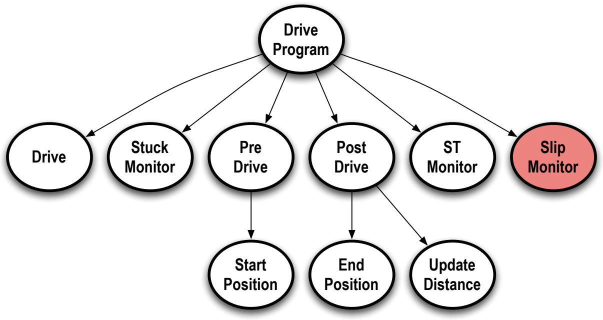

Figure 8. System adds slip monitor

When the programmer opens the program in their IDE to verify the change, a rule in the PLEXIL ruleset concerned with code style detects that the node *Drive Program* has three or more monitors attached to it. In an effort to declutter *Drive Program*, it suggests adding a node called *Monitor Set*, and placing all monitors beneath it. The programmer verifies the previous change and accepts this one as well resulting in the final program tree (Fig. 9).

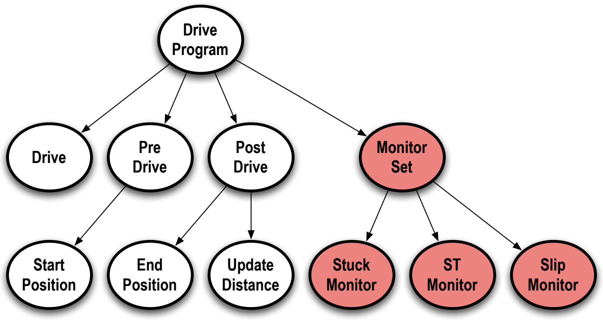

Figure 9. System organizes monitors

Later on, we could imagine this program getting transferred from the K9 rovers to the K10 model. The rover designers can scan through the rules and update them to meet the hardware requirements. The Sun Tracker was removed during the hardware upgrade, so the designers can add a rule to state that the rover position value can only be read from the new 3-axis compass. When the program is later checked for consistency with the new ruleset, the system can point out the hardware issues.

The system would find three places where the Sun Tracker was used within the code. Two are the *Pre Drive* and *End Position* nodes, where the code reads the position value. The last case is in *Slip Monitor* which checks that the Sun Tracker is on. With user confirmation, the system changes the code to read position data from the compass instead.

CONCLUSIONS AND FUTURE DIRECTIONS

The overall objective of this research is to support designers in working with complex hardware/software systems, by helping them *capture when and why the decisions were made* and *understand the consequences of these design decisions*. Our hypothesis is that every system is a compilation of design decisions. The ability to track these decisions will allow designers to have a better understanding of the system and enable the reuse of hardware and software more effectively by being able to go back and determine what decisions were made and why. This paper specifically presents a formal model of the design process, focusing on being able to track the decisions made during the design process. An initial model that accomplishes these goals is presented in this paper, and illustrated using a rover design example at NASA.

The overall goal of the research presented in this paper is to create a body of systematized knowledge about example-based design, which helps to better understand how example-based design works, under which conditions it succeeds and when it fails, and what is needed to support a methodology for designing by example. The model derived to represent the design and decision making process will have a variety of benefits.

1. The theory can be employed to *represent and study* a variety of *design domains* and to investigate the design process in these domains. For example, the design-attribute-oriented viewpoint helps to separate concerns in design domains and supports the identification of methods and techniques for a more effective design process.
2. Since in the model every design is expressed based on the basic entities of design attributes and design decisions, we can *identify measurements* that are based on these elements. In particular, differences between designs can be expressed in terms of these elementary units of design. Based on such difference measures, a concept of *design similarity* can be defined that can be exploited in tools supporting designers, for example, in searching for similar design examples.
3. The modular definition of the concepts of design domain and design space allows the creation of a sharable *repository* for *design spaces*, *attribute representations*, and *design contexts* Design domains and design contexts can then be defined by *reusing* definitions from the repository.

One of the goals in this research is to test the decision model on a real-life application. Future work will enable the simulation of real-world applications, using a simulator, the LUV Viewer, currently under development. The LUV Viewer reads in a PLEXIL file and displays the contained nodes. Then, it runs the program through the Universal Executive and displays the node status changes to the user. In order to handle communication with the external system, the programmer must also provide a simple script that contains a list of preprogrammed responses. However, it does not include any state information about the conditions in the surrounding world.

ACKNOWLEDGMENT

The authors would like to thank Dr. Guillaume Brat at CMU-Silicon Valley and Dr. Ann Patterson-Hine at NASA Ames research center for their valuable input to the project as part of the advisory board. This research is supported by the National Science Foundation's Science of Design program under Grant Number CCF-0741584. Any opinions or findings of this work are the responsibility of the authors, and do not necessarily reflect the views of the sponsors or collaborators.

REFERENCES

[1] UML. `http://www.uml.org`.

[2] E. Antonsson and J. Cagan. *Formal Engineering Design Synthesis*. Cambridge University Press, 2001.

[3] E. Antonsson and K. Otto. Imprecision in engineering design. *Journal of Mechanical Design*, 117(B):25–32, 1995.

[4] B. S. Baker. On finding duplication and near-duplication in large software systems. In *2nd Working Conf. on Reverse Engineering*, page 86, Washington, DC, USA, 1995. IEEE Computer Society.

[5] J. Barzilai. Measurement and preference function modeling. *International Transactions in Operational Research*, 12:173–183, 2001.

[6] S. Basil and R. K. Keller. Software visualization tools: Survey and analysis. In *9th Int. Workshop on Program Comprehension*, pages 7–17, Washington, DC, USA, 2001. IEEE Computer Society.

[7] Joshua I. Bernstein. *Design methods in the aerospace industry: looking for evidence of set-based practices*. Dissertation, Massachusetts Institute of Technology, 1998.

[8] Ted J. Biggerstaff and Alan J. Perlis, editors. *Software reusability: vol. 2, applications and experience*. ACM Press, New York, NY, USA, 1989.

[9] JPL Review Board. Report on the loss of the mars polar lander and deep space 2 missions. Technical Report JPL D-18709, NASA JPL, 2000.

[10] P. Borras, D. Clèment, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The System. In *3rd ACM SIGSOFT Symp. on Software Development Environments*, pages 14–24, 1988.

[11] E. J. Braude. *Software Design: From Programming to Architecture*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2004.

[12] M. "Bualat, L. Edwards, T. Fong, M. Broxton, L. Flueckiger, S. Y. Lee, E. Park, V. To, H. Utz, V. Verma, C. Kunz, and M." MacMahon. Autonomous Robotic Inspection for Lunar Surface Operations. In *Proceedings of the 6th International Conference on Field and Service Robotics*, 2007.

[13] D. Budgen. *Software Design (2nd edition)*. Addison Wesley Publishing Company, Harlow, England, 2003.

[14] M. Burgin, H. K. Lee, and N. Debnath. Software technological roles, usability, and reusability. In *IEEE International Conference on Information Reuse and Integration*, 2004.

[15] W. L. Chapman, A. T. Bahill, and A. W. Wymore. *Engineering modeling and design*. CRC Press, London, UK, 1996.

[16] Alistair Cockburn. *Agile software development*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[17] J. Coplien, D. Hoffman, and D. Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37 – 45, 1998.

[18] B. Corbett and D. W. Rosen. A configuration design based method for platform commonization for product families. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 18(1):21– 39, 2004.

[19] Y.-M. Deng, G.A. Britton, and S.B. Tor. Constraint-based functional design verification for conceptual design. *CAD Computer Aided Design*, 32(14):889–899, 2000.

[20] G. Dowek, C. Muñoz, and C. Păsăreanu. A Formal Analysis Framework for PLEXIL. In *Proceedings of 3rd Workshop on Planning and Plan Execution for Real-World Systems*, September 2007.

[21] M. Erwig. Programs are Abstract Data Types. In *16th IEEE Int. Conf. on Automated Software Engineering*, pages 400–403, 2001.

[22] M. Erwig and D. Ren. Type-Safe Update Programming. In *12th European Symp. on Programming*, LNCS 2618, pages 269–283, 2003.

[23] M. Erwig and D. Ren. An Update Calculus for Expressing Type-Safe Program Updates. *Science of Computer Programming*, 55, 2007. to appear.

[24] C. Feng and A. Kusiak. Constraint-based design of parts. *CAD Computer Aided Design*, 27(5):343–352, 1995.

[25] M. Fujita, S. Sasaki, and K. Matsui. Object oriented analysis and design of hardware software codesigns with dependence analysis for design reuse. In *IEEE International Conference on Information Reuse and Integration*, 2005.

[26] P. Gomes and C. Bento. A case similarity metric for software reuse and design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 15(1):21–35, 2001.

[27] G. A. Hazelrigg. *System Engineering: An approach to information-based design*. Prentice Hall, 1996.

[28] G. A. Hazelrigg. A framework for decision based engineering design. *Journal of Mechanical Design*, 120:653–658, 1998.

[29] G. A. Hazelrigg. An axiomatic framework for engineering design. *Journal of Mechanical Design*, 121:342–347, 1999.

[30] John Hunt. *Agile Software Construction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[31] Michael Jackson. *Problem frames: analyzing and structuring software development problems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[32] JM Jobe, TA Johnson, and CJJ Paredis. Multi-Aspect Component Models: A Framework for Model Reuse in SysML. In *ASME 2008 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (IDETC/CIE 2008)*, 2008.

[33] C. Kapser and M. Godfrey. Toward a taxonomy of clones in source code: A case study. In *Evolution of Large-scale Industrial Software Applications*, Amsterdam, The Netherlands, 2003.

[34] M. Kokkolaras, R. Fellini, H. M. Kim, N. F. Michelena, and P. Y. Papalambros. Extension of the target cascading formulation to the design of product families. *Structural and Multidisciplinary Optimization*, 24(4):293 – 301, 2002.

[35] Rainer Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance*, 15(2):87–109, 2003.
[36] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.
[37] T. Kurtoglu and I. Y. Tumer. A graph-based fault identification and propagation framework for functional design of complex systems. *Journal of Mechanical Design*, 2006. In review.
[38] E. S. Lamassoure, S. D. Wall, and R. W.. Easter. Model-based engineering design for trade space exploration throughout the design cycle. In *Proceedings of the Space 2004 Conference and Exhibit*, number AIAA 2004-5855, 2004.
[39] K. E. Lewis, W. Chen, and L. C. Schmidt. *Decision making in engineering design*. ASME Press, 2006.
[40] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.
[41] M. D. Mcilroy. Mass produced software components. In P. Naur and B. Randell, editors, *Proceedings of the NATO Conference on Software Engineering*, pages 138–155, Garmish, Germany, October 1968. NATO Science Committee.
[42] Tim Menzies and Justin S. Di Stefano. More success and failure factors in software reuse. *IEEE Transactions on Software Engineering*, 29(5):474–477, 2003.
[43] M. Morisio, M. Ezran, and C. Tully. Success and failure factors in software reuse. *IEEE Transactions on Software Engineering*, 28(4):340–357, 2002.
[44] Y.-E. Nahm and H. Ishikawa. A new 3d-cad system for set-based parametric design. *Journal of Advanced Manufacturing Technology*, 29(1-2):137–150, 2006.
[45] Bonnie A. Nardi. *A small matter of programming: perspectives on end user computing*. MIT Press, 1993.
[46] K. Otto and E. Antonsson. Trade-off strategies in engineering design. *Research in Engineering Design*, 3(2):87–104, 1991.
[47] K. N. Otto and K. L. Wood. Product evolution: A reverse engineering and redesign methodology. *Research in Engineering Design*, 10(4):226–243, 1999.
[48] K. N. Otto and K. L. Wood. *Product Design: Techniques in reverse engineering and new product development*. Prentice Hall, 2001.
[49] G. Pahl and W. Beitz. *Engineering Design: A Systematic Approach*. Springer-Verlag, London, UK, 1996.
[50] Panchal, J. and Fernandez, M. G. and Allen, J. K. and Paredis, C. J. and Mistree, F. An interval-based focalization method for decision-making in decentralized, multi-functional design. In *Proceedings of the ASME Design Engineering Technical Conferences*, 2005.
[51] C. Potts and G. Bruns. Recording the reasons for design decisions. In *Proceedings of the 10th international conference on Software engineering*, pages 418–427. IEEE Computer Society Press Los Alamitos, CA, USA, 1988.
[52] Damith C. Rajapakse and Stan Jarzabek. An investigation of cloning in web applications. In *14th Int. Conf. on the World Wide Web*, pages 924–925, New York, NY, USA, 2005. ACM Press.
[53] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York, 1989.
[54] L. Rideau and L. Thèry. An Interactive Programming Environment for ML. Rapport de Recherche 3139, INRIA, Sophia Antipolis, 1997.
[55] Gruia-Catalin Roman and Kenneth C. Cox. Program visualization: the art of mapping programs to pictures. In *14th Int. Conf. on Software Engineering*, pages 412–420, New York, NY, USA, 1992. ACM Press.
[56] D. G. Saari. *Decisions and elections: explaining the unexpected*. Cambridge University Press, New York, NY, 2001.
[57] S. Shan and G.G. Wang. Space exploration and global optimization for computationally intensive design problems: A rough set based approach. *Structural and Multidisciplinary Optimization*, 28(6):427–441, 2004.
[58] T. Shumann, O. L. de Weck, and J. Sobieski. Integrated system-level optimization for concurrent engineering with parametric subsystem modeling. In *Proceedings of the 46th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, number AIAA 2005-2199, 2005.
[59] T. W. Simpson. Product Platform Design and Customization: Status and Promise. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 18(1):3–20, 2004.
[60] II Sobek, Duward K. *Principles that Shape Product Development Systems: A Toyota-Chrysler Comparison*. Dissertation, University of Michigan, 1997.
[61] R. B. Stone, I. Y. Tumer, and M. E. Stock. Linking product functionality to historical failures to improve failure analysis in design. *Research in Engineering Design*, 16(2):96–108, 2006.
[62] R. B. Stone and K. L. Wood. Development of a functional basis for design. *Journal of Mechanical Design*, 122(4):359–370, 2000.
[63] Michael Toomim, Andrew Begel, and Susan L. Graham. Managing duplicated code with linked editing. In *IEEE Symp. on Visual Languages and Human-Centric Computing*, pages 173–180, Washington, DC, USA, 2004. IEEE Computer Society.
[64] D. G. Ullman. *The mechanical design process*. McGraw-Hill, 2003.
[65] D. G. Ullman. *Making robust decisions*. Trafford Publishing, 2006.
[66] A. Ward, J. K. Liker, J.J. Cristiano, and D. K. Sobek. The second toyota paradox: How delaying decisions can make better cars faster. *Sloan Management Review*, 36(3):43–61, 1995.
[67] H. J Wassenaar, W. Chen, J. Cheng, and A. Sudjianto. Enhancing discrete choice demand modeling for decision-based design. *Journal of Mechanical Design*, 127(4):514–523, 2005.
[68] J. Whittle, A. Bundy, R. Boulton, and H. Lowe. An ML Editor Based on Proof-as-Programs. In *9th Int. Symp. on Programming Language Implementation and Logic Programming*, LNCS 1292, pages 389–405, 1997.
[69] J. Whittle, A. Bundy, and H. Lowe. An Editor for Helping Novices to Learn Standard ML. In *14th Int. Conf. on Automated Software Engineering*, 1999.
[70] S. Wilhelms. Function- and constraint-based conceptual design support using easily exchangeable, reusable principle solution elements. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 19(3):201–219, 2005.
[71] P. M. Wognum and I. F. C. Smith. Reuse of designs. *Knowledge-Based Systems*, 9(2):79–81, 1996.
[72] W. H. Wood and A. M. Agogino. Decision based conceptual design: Modeling and navigating heterogeneous design spaces. *Journal of Mechanical Design*, 127(1):2–11, 2005.