Structuring Product Family Requirements for n-Dimensional and Hierarchical Product Lines*

Jeffrey M. Thompson and Mats P.E. Heimdahl
Department of Computer Science and Engineering
University of Minnesota
4-192 EE/CS; 200 Union Street S.E.
Minneapolis, MN 55455 USA
+1 (612) 625-1381
{thompson,heimdahl}@cs.umn.edu

Abstract

The software product-line approach (for software product families) is one of the success stories of software reuse. When applied, it can result in cost savings and increases in productivity. In addition, in safety-critical systems the approach has the potential for reuse of analysis and testing results, which can lead to a safer systems. Nevertheless, there are times when it seems like a product family approach *should* work when, in fact, there are difficulties in properly defining the boundaries of the product family.

In this paper, we draw on our experiences in applying the software product-line approach to a family of mobile robots, a family of flight guidance systems, and a family of cardiac pacemakers, as well as case studies done by others to (1) illustrate how domain structure can currently limit applicability of product-line approaches to certain domains and (2) demonstrate our progress towards a solution using a set-theoretic approach to reason about domains of what we call *n-dimensional* and *hierarchical* product families.

*This work as been partially supported by NSF grants CCR-9624324 and CCR-9615088, and by NASA grants NAG-1-2242 and NCC-01-001.

1 Introduction

Software product-line engineering has the potential to deliver great cost savings and productivity gains to organizations that provide families of products, as well as give those organizations a competitive edge in the market-place. For safety-critical systems, software-product line engineering has the potential to produce systems that are more safe than their serially produced counterparts while being cheaper and faster overall to build.

Although one of the main barriers to the use of product family techniques is one of process and organizational acceptance, technical issues have not been completely solved for product-line engineering. The techniques available work best for cohesive product families, where the variabilities do not have complex interdependencies. When this is not the case, it can be difficult to apply the product family approach even though there might be significant commonalities between the members of the family. As an alternative, we propose to view the families themselves in a multi-dimensional and hierarchical fashion. This helps us to deal with existing problems, for example, *near commonalities*, and also, helps to extend the approach to domains that, traditionally, would be difficult for product-line engineering.

The paper is organized as follows. The next section presents background on work in product-line engineering. Section 3 presents current issues with the product family approach. Then, we discuss the foundations of our approach in Section 4. Section 5 presents how our approach can be used to address existing issues in product-line engineering. The FGS case example is discussed in Section 6 and the mobile robotics case example is in Section 7. Finally, Section 8 presents a brief evaluation of the technique while Section 9 presents our conclusions and ideas about future directions for this work.

2 Product-line engineering background

The notion of a product family was introduced by David Parnas in [21]. According to Parnas, it is desirable to study a group of programs as a whole whenever the programs share extensive commonalities. Parnas observed that often programmers would create new programs by modifying existing programs. This process usually involved a reverse step where parts of the working program were discarded. The new program was sometimes crippled by design assumptions made for the original program that did not apply to the new program. Thus, Parnas postulated that it would better to start out by defining what was common about all such programs and successively refining the design until you had working programs as the leaves of a tree structure, with nodes within the tree representing the various design decisions.

Batory and O'Mally [3] discussed how to reuse large portions of a system based on breaking it into components and introduced a simple language for describing the components and their composition. Gomaa [13] discusses using domain modeling [22] to create a centralized library of components which are then used by a generation facility to produce the target application.

Weiss and Ardis [25, 2] developed the FAST (Family-oriented Abstraction, Specification and Translation) process that integrates the above with specialized languages [20, 6] for each domain. A similar process is mentioned by Campbell *et al.* in [10, 9] and also by Lam [16, 15]. The differences between these works are primarily in the sort of artifacts produced by the domain engineering effort.

The commonality analysis [24] is a central feature of product-line engineering. This is the document that notes all the commonalities, i.e., features which are present in all systems in the domain, and variabilities, i.e., features which distinguish the different members of the domain. The commonality analysis defines the requirements for the product line.

Current techniques for product-line engineering work well if the following conditions are met:

- The systems in the family share significant commonalities, and
- The variabilities which define each family member have a straightforward decision model, i.e., it does not require many complicated rules to describe how the variability values are assigned to produce each family member.

The first point describes the essential feature of product families that Parnas noticed in his work. However, the second point originates in the practical experience of many researchers who have labored to construct software product-lines. Robyn Lutz observed that the primary limitations of the product family approach stem from difficulties in handling "*near-commonalities* and relationships among the variabilities"[emphasis added] [18]. Thus, the more simple the relationships among the variabilities, the easier it is to construct the product family.

Most of the current research and approaches to product family engineering focus on developing the assets (i.e., reference architectures or generation facilities) using the commonalities and variabilities as a requirements specification. The issue of how to structure the architecture to overcome difficulties in the family itself (such as near-commonalities) is often intermingled with solutions to general architecture problems.

Recording the structure of the product family at the requirements level before an architecture has been constructed may provide advantages in making an architecture that is more flexible in the face of changes to the domain. Therefore, we feel that it is important

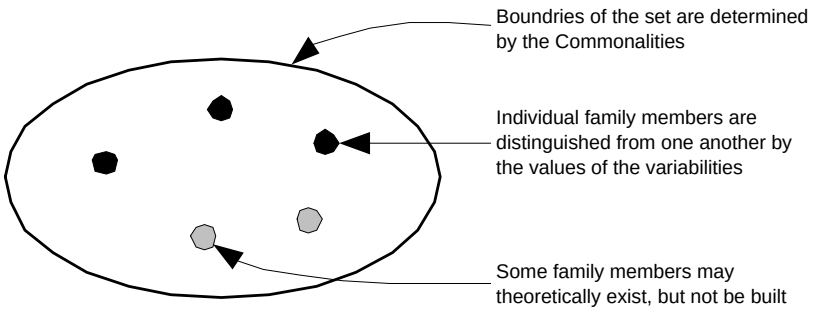

Figure 1: A simple product family

to develop a structuring technique for product families which is independent of architectural structuring and is designed to be utilized at the requirements level. This would allow the potential to develop analysis techniques for the product family requirements and could provide insight into the high-level structure of the architecture. In this paper, we focus on the structures that we have observed in the domains themselves, independent of the architectures that are used. This high-level structure can guide the later creation of the architecture and is therefore complementary to current work in the field.

3 Extending product families

One way to view a product family is as a set, where the boundaries of the set are determined by the commonalities, and the individual members of the set are distinguished by the values of their variabilities (Figure 1). As the figure demonstrates, it is entirely possible that some members of the family may theoretically exist but not yet be built (shown in gray). Furthermore, the family may be undefined at some points within the boundaries due to, for example, illegal or nonsensical combinations of variability values. We will use this view of a product family throughout the paper to demonstrate how current approaches to product-line engineering might be expanded to a greater class of systems.

3.1 n-Dimensional product families

Attempts have been made to organize the product family requirements in a hierarchical fashion [18, 21, 14, 15]. Lutz noted in her attempt to organize the variabilities into a tree that "there were several possible trees, with often no compelling reason to select one

possible tree over another" [18].

Brownsword and Clements present a shipboard command and control systems family which contained 3000-5000 parameters of variation for each ship [8]. They state that "the multitude of configuration parameters raises an issue which may well warrant serious attention." In addition, they present three different views of the architectural layering of the base system that "do not conflict with each other; rather they provide complementary explanations of the same ideas."

Both these examples, as well as our own experience in the domain of mobile robotics, illustrate the fact that often a product family is multi-dimensional; therefore, a hierarchical decomposition is not sufficient to capture the structure of the domain. As an example, mobile robots form families along the dimensions of hardware platform (common basic features, but different modes of locomotion, different environmental sensors, different manipulators, etc.) and behaviors (common basic behaviors, but they may also require wall following, obstacle avoidance, mapping, etc.). We call families that decompose naturally along such dimensions *n-dimensional product families.*

n-Dimensionality is common in software systems. Thus, software design and implementations already deal with problems associated having an n-dimensional space. Our approach is similar in structure to the notation of design spaces [17] and extended design spaces [5, 4]. In addition, there is much work in structuring that has been done in the object oriented community, and software architecture community that may be applicable to the requirement phase. The key, in our view, is to define a simple structuring mechanism which does not introduce unnecessary design or implementation detail but which is still able to capture the essence of the problem at hand.

3.2 Hierarchical product families

Suppose that a company wished to construct a flight guidance system (FGS) for both fixed-wing aircraft and helicopters[1]. The FGS is responsible for issuing commands that keep the aircraft level, cause it to climb or descend, and so forth. Furthermore, the FGS must interact with other airborne systems. Many of the tasks that the system has to perform might be common across these two radically different aircraft: interaction with other systems, deciding to level off at a particular altitude, mode transition logic related to when it is legal to switch between the various operating modes. Therefore, many requirements between these two systems will be the same, or very similar. Nevertheless, the actual control of the aircraft is very different. Therefore, developing a single set of commonalities and variabilities that span this entire domain is difficult.

[1]We would like to thank Steven P. Miller of Rockwell-Collins Inc. for this example.

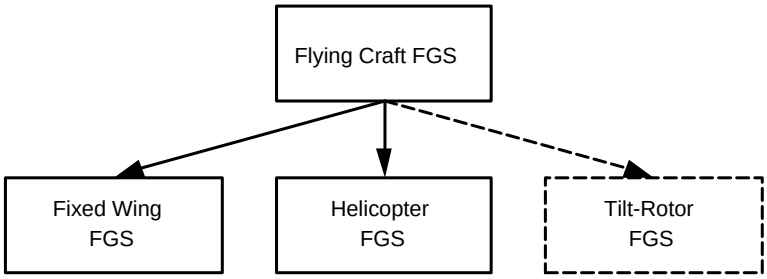

Figure 2: FGS product family covering flying craft

Some would argue that this difficulty stems from the fact that the family is simply too diverse to be considered a product line. However, it is clear that these systems share much in common, which was the original, and in our view the most important, criterion for being a family. Thus, we propose the concept of a *hierarchical product family*.

Most previous attempts at product family structuring have focused on hierarchically grouping the *variabilities* while the *commonalities* remain the same for all family members [18, 15]. Notable exceptions are Parnas [21] and Brownsword and Clements who noted in their case study at CelciusTech [8] that sometimes product-lines exist within the main product line. However, Parnas' work is rooted in design and coding choices. Brownsword and Clements mention this phenomenon in passing and apply it in a more limited way than what we advocate.

In our approach, *additional commonalities* which are *unrelated* to the commonalities of the parent product family can be added in the sub-families. Of course, these additional commonalities cannot conflict with those in the parent family. The hierarchical decomposition of the FGS family is shown in Figure 2. Thus, the helicopter sub-family can have significantly different requirements than for fixed-wing aircraft, yet share many things in common as well.

This will eventually effect the architecture and structure of the systems. For example, the product of the domain engineering for the parent family, Flying Craft FGS, might be a set of reusable components, whereas the product of domain engineering for the children might be a reference architecture or generation facility. The architectures for the fixed-wing aircraft and the helicopters could differ significantly and use the components from the parent family in different ways.

By structuring the requirements in this way, we have avoided imposing restrictive design constraints on the family members and instead focus on the structure of the domain itself. Furthermore, should the company wish to start building FGS systems for an entirely new set of aircraft, for example, tilt-rotor aircraft, this could be done while reusing many

aspects of the FGS systems already implemented. This is also shown in Figure 2.

3.3 Constraints on the solution

When starting to develop a structuring technique for product family requirements that would be able to deal with n-dimensional and hierarchical product families, we determined that any such structuring technique must:

- Be based on structures that are present in the *domain* itself, not on implementation or design concerns,
- Be simple, allowing the analysts to capture the structure of the domain without introducing complex notations or concepts,
- Be amenable to the types of structures observed in product family analysis, and
- Produce a readable and usable artifact that facilitates reasoning about the structure of the domain.

We chose to explore a structuring technique based on a set-theoretic view of product families. The notion of sets proves surprisingly useful for thinking about the structure of a software product line, yet is simple and based upon well understood principles.

4 Structuring technique

As mentioned previously, a product family can be described in terms of a set, where the boundaries of the set are defined by the commonalities and the members of the set are distinguished by the values of the variabilities. This section describes how set theory can be used to think about structuring product families.

The most basic structure that can be represented with the set theoretic approach is the subset. Figure 3 shows a product family, **A**, which has been divided into two subsets, **B** and **C**. Furthermore, **C** has been further divided into subsets **D** and **E**. This corresponds to a hierarchical decomposition of the family.

Consider a member of family **E**, e_1. The member e_1 must have all the commonalities defined for **E** as well as have some value for all the variabilities in **E**. Furthermore, because **E** is a subset of **C** and **A**, e_1 is also a member of families **C** and **A**. The general definition for any family **E** which is a subset of another family **C** is as follows:

- **E** must include all of the commonalities in **C**.

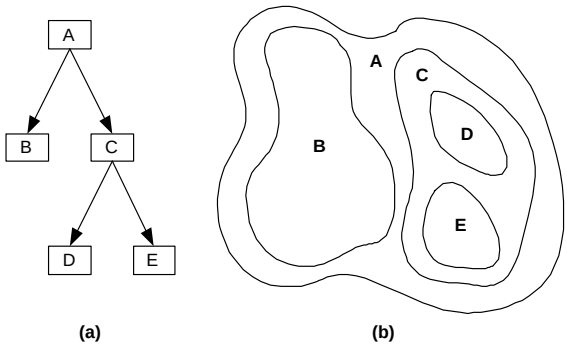

Figure 3: Hierarchical decomposition and subset structure

- **E** must include all of the variabilities in **C**; however, **E** may restrict the range or options available in the variabilities.
- **E** can add additional commonalities which are not present in **C** as long as the additional commonalities do not conflict with the commonalities or variabilities in **C**. These new commonalities might come from a refinement of variabilities in **C** or might be completely unrelated.
- **E** can define additional variabilities which are not present in **C** as long as those variabilities do not conflict with the above.

The first criterion is straightforward and necessary for the subset **E** to be completely contained within **C**. The second criterion defines the fact that **E** may wish to refine or restrict the values of the variabilities of **C**. For example, in the mobile robotics domain, a variability across the entire domain might be that the maximum speed of the mobile robot can vary from one to five miles per hour. However, subsets might define a lesser maximum speed depending on the hardware involved. It is possible for this refinement to result in an additional commonality, for example, suppose that we have aircraft that can use either radio altimeters, barometric altimeters, or GPS altimeters to gain altitude (a variability); then, a subfamily of these aircraft could state as a commonality that all aircraft in that subfamily have only barometric altimeters. Additional commonalities can also be added which are unrelated to the parent family. For example, it is likely that the family of helicopters will need different commonalities than the family of fixed-wing aircraft. Finally, it is possible to add additional variabilities.

The two cases of hierarchical decomposition are shown in Figure 4. Part (a) of the figure demonstrates that the family **R** need not have any members that only exist in **R**. In a sense, **R** is an *abstract family*, because any member of **R** must be either a member

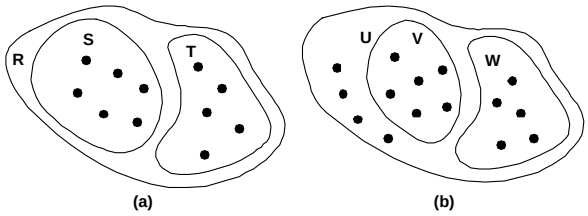

Figure 4: Abstract verses non-abstract families

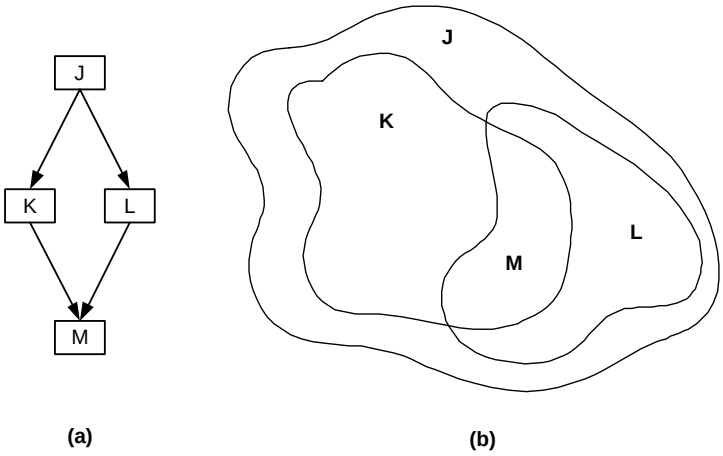

Figure 5: Set intersection and non-hierarchical structure

of **S** or a member of **T**. This is similar to our FGS example from earlier, where all family members are either helicopters or fixed-wing aircraft and it does not make sense to talk about member which are only of the parent family. However, this need not be the case, as Figure 4(b) demonstrates. In the mobile robotics domain (see Section 7), we will have a basic robotic platform which will form the outer family member. This outer family will *not* be abstract because there are some robots which only conform to the minimum specification.

Another structure that can be represented using a set-theoretic approach is that of set intersection. The ability to represent a set intersection distinguishes this approach from the purely hierarchical structures which have been applied by others. This is shown in Figure 5.

Consider a member, m_1, of **M**. By definition, m_1 is also a member of families **K**, **L**, and **J**. Thus, m_1 must have all the commonalities of both **K** and **L**. In addition, **M** is a

subfamily of both families **K** and **L** (this is shown in the figure). The constraints on any family **M** which is a subset of families **K** and **L** are as follows:

- **M** must include all the commonalities of both **K** and **L**.
- **M** must include all the variabilities of both **K** and **L**; however, it may restrict those variabilities as above for subsets.
- **M** may introduce additional commonalities which are not present in either **K** or **L** so long as those commonalities do not conflict with the commonalities or variabilities in **K** or **L**.
- **M** may introduce additional variabilities which are not present in either **K** or **L** so long as those variabilities do not conflict with the above.

These structures can be used to describe product families which are both *n-dimensional* and *hierarchical*. Representing hierarchy is done by primarily by using the subset concept. Representing a dimension requires a bit more thought.

Dimensions represent alternate views of the product family based on some particular aspect of the commonalities and variabilities. For example, commonalities and variabilities in the hardware platforms may be viewed as one dimension, and the functionality of the family members viewed as another dimension. Our notion of dimensions is similar to the notion of dimensions in extended design spaces [5, 4]; however, we would advocate the choice of several primary dimensions defined over cohesive aspects of the system and not make every variability a dimension. Possible dimensions may be hardware platform, required behavior, fault tolerance capabilities, etc. Some examples of dimensions are provided in Sections 6 and 7.

When a family has been decomposed into several dimensions, we expect to have to make a choice in each one of those dimensions in order to have a valid family member. That is, we will have to instantiate the variabilities and select, for example, both a hardware platform as well as the desired functions for a family member.

5 Addressing existing issues

This section describes how our approach can assist with well-known documented issues in product-line engineering. We describe how our structuring method can deal with near-commonalities as well as variability dependencies.

Near-commonalities: A near-commonality (NC) is a commonality which is true for almost all (e.g., all except one) member of the product family. Lutz states that in her

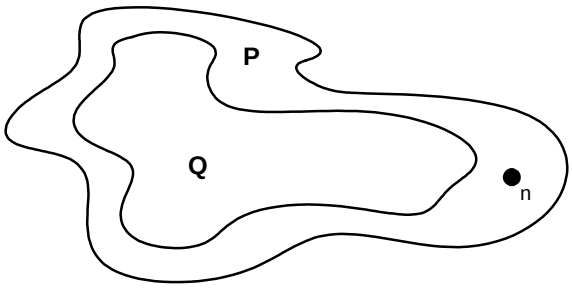

Figure 6: Set representation of a near-commonality

experience near commonalities "frequently had to be modeled" [18]. One solution for near commonalities is to model them as variabilities; however, this is, in some sense, a misrepresentation of their basic properties. The solution that Lutz advises is to model it as a constrained commonality of the form "If not member **n** then NC_1." However, a complex domain might contain numerous constrained commonalities with conditions significantly more complex than the example just mentioned.

Figure 6 shows how a near-commonality is represented in our approach. The near commonality, NC_1, would simply be a property of family **Q** (and not of **P**). Thus, the commonality naturally does not apply to **n**, a member of only **P** but does apply to any member of **Q**. This has several advantages. First, NC_1 is now a pure commonality of **Q**. Second, if another member of the family is introduced with reduced functionality [18] it need only be added as a member of **P** and **Q** may remain untouched. Finally, the subset structure can act as a guide in determining that certain components in the eventual application engineering environment will not be needed for **n**.

Dependencies among options: In [18], Lutz cites modeling dependencies among options as one issues that must be addressed in product family engineering effort. A dependency is typically a constraint among the variabilities, for example, if variability V_1 has value B then variability V_2 must have option C. Ardis recommends treating this constraint as a commonality. However, in our experience, without some additional structuring, the domain could become littered with such commonalities; in addition, it may not be clear given a set of constraints whether or not a particular variability is viable.

In our approach, we can also represent constraints like these as commonalities. However, we isolate them into logical groups by forming different subfamilies so that their numbers do not become overwhelming. In the abstract example given above, a set would be defined where "V_1 has option B" and "V_2 has option C" are both commonalities.

6 Flight Guidance System

In this section, we describe a hypothetical Flight Guidance System (FGS) family. This example is loosely based on some of the FGS systems built by Rockwell-Collins, but does not represent actual products of the company. This FGS example is essentially the same one that has been specified in numerous other publications, including [19, 12].

The purpose of the FGS is to compare the measured state of the aircraft to the desired state of the aircraft and then generate pitch and roll commands that attempt to maintain the measured state as close as possible to the desired state. The FGS can be partitioned into two pieces: (1) the continuous control laws that govern the performance of the various control surfaces of the aircraft, and (2) the complicated mode logic that defines how the FGS switches between these control laws.

The following commonalities describe the overall structure of the FGS.

C1 Every FGS has a lateral axis and a vertical axis, each of which has one or more modes.

C2 On every FGS, exactly one mode shall be active at one time along each axis

C3 Every FGS designates one mode for each axis (lateral and vertical) that shall be made active in the event that no other mode on that axis is active. This is called the *default* mode for that axis.

The ways in which each FGS differs is primarily in the number and type of modes that are present on the various aircraft. The engineers think of the modes as pluggable features; however, in reality there are dependencies among the choices of which modes the aircraft contains as well as subfamilies of the aircraft that contain the same collection of modes.

V1 The set of modes along each axis varies from aircraft to aircraft

V2 The mode which each FGS designates as the default varies from aircraft to aircraft

V3 The FGS may or may not select the default mode for a particular axis upon transfer of flight guidance computations

Each mode, for example, Roll Mode can be viewed as defining a sub-family of FGSs that contain that mode. Thus, an FGS that contained Roll mode, Pitch Mode, and Heading mode would exist at the intersection of these three sub-families.

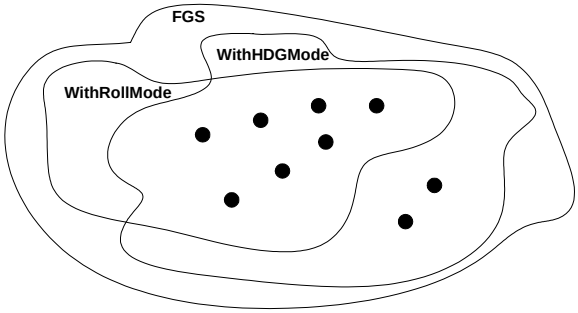

Figure 7: Example of sub-families of FGS

$C_{Roll}1$ Every FGS with a Roll mode uses a roll reference

$C_{Roll}2$ The roll reference is synchronized with the SYNC switch is pressed with the Flight Director on

$V_{Roll}1$ There may or may not be a roll knob to adjust the roll reference.

$V_{Roll}1.1$ The roll knob may have a detent angle of 0, 5, or 6 degrees.

$V_{Roll}2$ The Roll/Heading transition angle can assume values of 5 or 6 degrees

In this way, each mode can be specified. Figure 7 shows a simple example of the intersection between the Roll and Heading modes. Although this example is purely hypothetical, we can see from the figure that most FGSs support both Roll and Heading mode while two FGSs do not support Roll mode. Notice, that if only one FGS did not support Roll mode this would be a *near commonality*; in our structuring technique a near commonality is simply a special case of one sub-family having many less members than another sub-family.

If two modes must occur together in every FGS, then they can be specified in the same sub-family and that constraint can be noted as a commonality. This occurs between several lateral and vertical modes that must synchronize with each other. One example is the lateral and vertical go around modes.

$C_{GA}1$ Every GA-FGS has both a lateral and a vertical NAV mode

$C_{GA}2$ The lateral and vertical go around modes are always either both active or both cleared

$V_{GA}1$ The number and type of cockpit-located switched used to select go around mode varies from aircraft to aircraft.

In all, the FGS family maps into current product family techniques reasonably well and therefore does not showcase as much the power of our approach. The next section presents a more complex example of how this technique can be used in the mobile robotics domain.

7 Mobile robotics: An n-dimensional and hierarchical example

The domain of mobile robotics that we have examined encompasses small robots ranging in size from approximately 6 inches long to up to about 2 feet long. Also, the robots can support many different behaviors – scouting an area, constructing a map, working collectively, and so forth. We wished to model the mobile robotics domain as a product family taking into consideration both the different hardware platforms that could be supported and the many different behaviors we wished to specify. This proved to be difficult using conventional product-family techniques because the mobile robot family is both *n-dimensional* and *hierarchical* [11].

The mobile robotics domain breaks down along two clear dimensions: the hardware platform and the desired behavior. Each hardware platform conforms to a basic specification: it can move forward and backward, turn left and right, sense whether or not an object is in front of it, and so forth. In addition, the hardware platform may or may not be equipped with some sort of vision system or infra-red camera; the various sensors used to monitor the environment differ greatly in the speed and accuracy with which they provide information. On the behavior side, we can imagine that a basic behavior might be a random exploration of the robot's environment where the primary goal of the robot is collision avoidance and recovery. Furthermore, more complex behaviors can be added, for example, wall following, going through doors, and finding particular objects. Therefore, along both the hardware and behavior *dimensions*, the mobile robot family can be viewed *hierarchically*.

Of course, there are constraints between the two dimensions and not all behaviors can run on all hardware platforms. For example, a behavior that requires the robot to find all red objects in a room will not work unless the robot has a sensor capable of distinguishing red objects from non-red objects.

As a specific example for this paper, we will consider a mobile robotics domain consisting of three classes each of robots and behaviors. Although this is a *significant* simplifica-

tion of the actual domain, it will be sufficient to illustrate the benefit of the structuring techniques that we are proposing. In addition, due to space constraints we will not be able to go into detail on the particular parameters of variation (i.e., the particular values that each variability can assume) for each of the variabilities involved.

A family member will consist of a pairing of the desired behavior with the robotic platform. We will first explain each dimension in some detail and then look at an overview of the family composed of these two dimensions. As a notational convention in the following two sections, commonalities and variabilities which deal with the hardware are noted [C_H] and [V_H] respectively and those which deal with the behavior are noted [C_B] and [V_B].

7.1 Hardware dimension

Along the hardware dimension, we will consider a limited subset of the robot domain containing three families of hardware. The actual domain is much more complex; it includes many more types of sensors and different actuators, for example, a gripper or robotic arm that can be used to pickup and move objects in the environment. We will consider the following three classes of mobile robotic hardware in this paper.

1. A basic robot with forward and backward motion capabilities, a range sensor that give distance to the nearest obstacle and whether or not the obstacle is on the right or on the left, and a forward collision detection mechanism.
2. The basic robot with the ability to distinguish between obstacles that are straight ahead versus only the right or left (i.e., better granularity in the estimation of the obstacle's position).
3. The basic robot with the ability to distinguish the color of objects in its environment.

The following paragraphs describe the properties of the various hardware platforms that we will consider.

Basic platform: A basic feature of our robotic platform will be that it can move around its environment in some fashion. Thus a common feature of the robots is the following:

C_H1.1 Each platform will provide a basic means of locomotion; it will have the ability to turn a specified number of degrees from the initial heading, move forward, move backward, and stop.

Nevertheless, the robotic platforms that we will consider differ greatly. Some platforms are commercially built whereas others are built in-house, for example, out of Lego building blocks and small motors. The following variabilities capture these ideas:

V_H1.3 The hardware comprising the robotic platform varies

- V_H1.3a The means of locomotion may vary (e.g. treads, wheels, legs, etc.)
- V_H1.3b The maximum speed of the robot varies.
- V_H1.3c The control of locomotion varies. The locomotion system may provide simple on/off values or real or digital valued representation of speed and direction.
- V_H1.3d The type of input expected by the locomotion system varies. It may expect boolean, real, or digital values indicating speed and direction of the platform.
- V_H1.3e The size of the platform varies. This will dictate the amount of room needed to turn or avoid an obstacle.

In order to avoid running into obstacles in the environment, the robot must have some kind of range finder. The platform must also be able to tell whether or not the obstacle is on the right or left so that it can take actions to avoid hitting the obstacle. However, range finders vary significantly in the type and quality of information they provide. For example, a sonar sensor provides a wide field of detection but is noisy and inaccurate. A laser range finder, on the other hand, will provide distance with high accuracy and can detect even small obstacles.

C_H1.2 All platforms will have at least one range finder that will provide input to the system regarding the detection of an obstacle.

- C_H1.2a The range finder will provide an indication of the distance to the obstacle.
- C_H1.2b The range finder will provide an indication of the location (right or left) of the obstacle in relation to the robot.

V_H1.1 The number and type of devices used for range finding is likely to vary. The type of output generated by the range finder varies. Different range finders may provide output as a real-valued estimate, a digital estimate, or a boolean indication of obstacle detection.

Finally, because the mobile robots operate with such noisy and inaccurate sensors it is a certainty that they will occasionally have collisions. Thus, platforms must have a method of detecting collisions so that they can perform recovery actions in the behaviors. This could be implemented in a variety of different ways, for example, buy installing bumpers on the robot or by detecting that the motors that drive the wheels have stalled.

C_H1.3 All platforms will have at least one mechanism for detecting collisions.

V_H1.2 The number and type of collision sensors(s) varies and the type of output generated by the collision sensor varies.

Enhanced obstacle detection: Some platforms may have more advanced sensors to detect obstacles. For example, a robot with an array of sonar sensors arranged in an arc can get much more information about potential obstacles than merely whether they are on the right or on the left. For enhanced obstacle detection, the robotic platform should be able to detect whether or not it has an obstacle in front of it in addition to obstacles on the right and left.

C_H2.1 Platforms will have the ability to distinguish whether an obstacle exists directly in front of them as well as whether it is on the right or on the left. See related [C_H1.2b]

V_H2.1 The granularity of obstacle position detected will vary. For example, some platforms may provide an enumerated indication of left, right, or front for the obstacle whereas some may provide an estimated degrees to the obstacle.

This sensing capability allows the robot to perform more complex behaviors, for example, maneuvering closer to obstacles or going through doors.

Environmental vision: Some robots may be equipped with a camera or other sensing device that can give them information about the color objects in their environment. The type and quality of robotic vision systems varies greatly; however, most can distinguish between primary colors.

C_H3.1 Platforms will have a sensor capable of determining the color of objects in their environment; for example, the sensor should be able to distinguish between red objects and blue objects.

7.2 Behavioral dimension

The behavioral dimension defines *what* the robot does. Of course, the behavior of the robot is highly related to the hardware dimension, which constrains what the robot *can* do and what information about the environment is available. Nevertheless, to a large extent the behaviors can and should be reused across different hardware platforms. The spectrum of behaviors possible, even with the limited hardware classes that we have defined, is large. For the purposes of this report, we only have space to discuss a few of them. Thus, along the behavioral dimension, we will consider the following classes of behavior.

1. Random exploration, where the robot moves around its environment attempting to avoid obstacles.
2. Random exploration with the ability to negotiate doors.
3. Random exploration with the ability to signal when it encounters objects of a particular color.

The following paragraphs discuss these behaviors in more detail.

Random exploration: Rodney Brooks [7] recommends a layered architecture of robotic behaviors with a simple reactive behavior being on the lowest level and higher-level behaviors built on top of this. When the robot encounters a problem, for example, a collision, in a higher-level behavior, then the higher-level behavior is suspended by a lower-level behavior designed to correct the problem. Our approach to modeling the behavioral dimension is similar in that our basic behavior is a random environmental exploration and more complex behaviors are built on top of it.

Our basic behavior is a random exploration; while exploring, the robot should attempt to avoid obstacles in the environment.

C_B1.1 The robot shall attempt to avoid colliding with obstacles in its environment using its sensors to detect obstacle(s) and changing its course or speed to avoid the obstacle.

V_B1.1 Although detected by the robot's sensors, an object may or may not be considered an obstacle depending on the robot's mode of operation. See, for example, [C_B2.1a]

As mentioned previously, because of the robot's noisy and inaccurate sensors it is likely that the robot will sometimes collide with an obstacle. When this occurs, the robot should attempt to recover from the collision and continue exploration.

C_B1.2 If the robot collides with an obstacle, it shall attempt to recover from the collision.

V1.2 Successive collisions (i.e., a collision during the recovery from a previous collision) may result in the robot shutting down all activity and declaring failure. The number collisions in a chain that the robot can tolerate varies.

The random exploration behavior coexists with all the other possible behaviors that we might define. In the absence of any obstacle or collision, the robot will potentially be performing some other functions which are defined by a subfamily. However, this family is *not* abstract; thus, if no other behaviors are specified the robot will move forward at full speed.

V_B1.3 In the absence of an obstacle or collision, the behavior of the robot may be further specified by a sub-family

C_B1.3 In the absence of an obstacle, collision, or any other specified behavior, the robot will move forward at maximum speed.

Door navigation: Maneuvering through a doorway is difficult for a mobile robot. Often, obstacle detection sensors provide little information about the environment; thus, doorways are often not seen as viable passageways. Furthermore, it is difficult for the robot to find doorways in the first place given the noisy sensor data it receives.

C_B2.1 The robot shall attempt to locate doors in its environment

C_B2.1a Once the robot has found what it believes to be a door, it shall not consider the sides of the door to be obstacles as the door is navigated. See [C_B1.1], [V_B1.1].

V_B2.1 The width of the door which can be navigated by the robot will vary according to the width of the robotic platform and the quality of the on-board sensors.

Environmental interpretation: This behavior allows the robot to signal when it encounters a particular object in the environment. That object or objects will be identified by a particular color.

C_B3.1 The robot will signal when it has detected an object in its environment of the desired color.

V_B3.1 The color of the object(s) to be detected will vary and may be configurable at run time.

Figure 8 shows a photograph of two of the mobile robots used as an example in this paper.

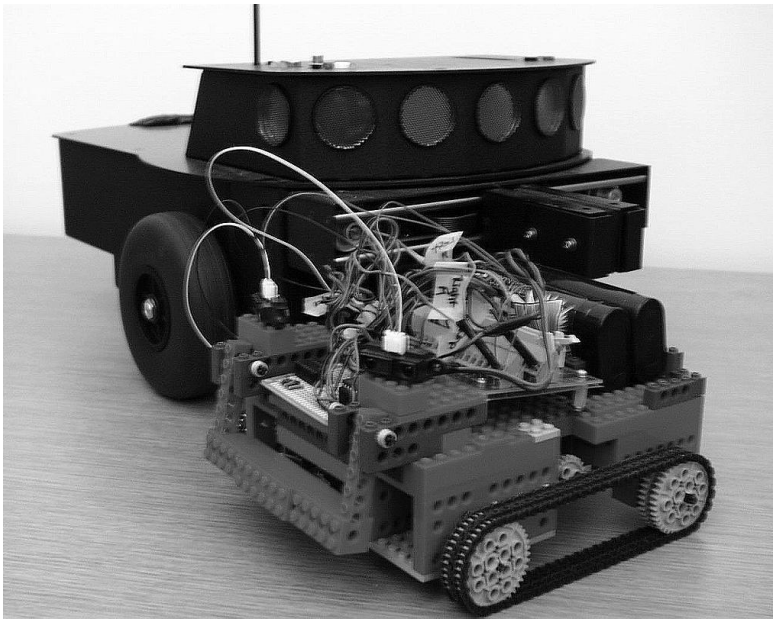

Figure 8: Pictures of the Mobile Robots

7.3 The whole family

The real mobile robotics domain is significantly more complex than space allows us to present in this paper. For example, we have not discussed whether or not the robot can move objects in its environment (with a gripper, for example). Nevertheless, we can illustrate some interesting properties of the domain even with this limited example. Suppose that we had four mobile robots at our disposal:

- A custom robot made out of Lego pieces with two infra-red sensors in the front for obstacle detection, a front bumper, and tank-tread locomotion. We will call this one LegoBot.
- A Pioneer robot made by ActivMedia [1] which has an array of sonar sensors, a gripper, collision detection via motor stalled, and wheels for locomotion. We will call this one Pioneer.
- A Pioneer (see 2) with a color vision system. We will call this one Pioneer w/ Vision.
- A small "pickle" robot that can roll around, and jump over small obstacles, and that is equipped with a camera. We will call this one Pickle.

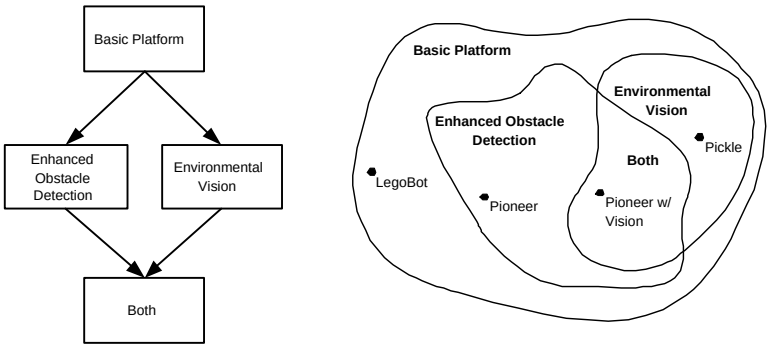

Figure 9: The mobile robot family along the hardware dimension

There are several ways of visualizing the mobile robot product family. First, we will examine the mobile robot family along the hardware dimension (Figure 9). Notice that family members can fall into one of four different categories. The robot may have only the basic capability, in which case it exists only for the family **Basic Platform**. This is the case for LegoBot. The robot may have either one or the other of the additional hardware capabilities specified by the **Enhanced Obstacle Detection** or **Environmental Vision**. Finally, the robot may posses both the additional capabilities of **Enhanced Obstacle Detection** and **Environmental Vision**; therefore, it lies in the intersection of those two subfamilies. This is only one slice of the system, however, and if we were to look at the mobile robot family along the behavioral dimension we would see a similar picture. A somewhat more effective means of viewing 2-dimensional product family is in a 2-dimensional grid as shown in Figure 10.

The representation is symmetrical in this case because of the one-to-one mapping between behavioral subfamilies and hardware subfamilies. The full mobile robotic domain, however, is not symmetrical. In the full domain, behaviors may be composed and combined to form a composite behavior. For example, we might envision a behavior which includes the door navigation, combined with a mapping function, a wall following behavior, and a high-level planner. The mapping and high-level planning behaviors will need to communicate with the lower level random exploration, door navigation, and wall following to direct the robot towards high-level goals. However, if the robot collides with an obstacle, then the lower level behavior will take over and recover from the collision. Thus the structuring of the behavioral dimension is much more complex and resembles Brooks' subsumptive architecture [7]. Furthermore, defining the behaviors independent of the hardware allows us to focus on only the behaviors and their interactions (a significant problem in an of itself).

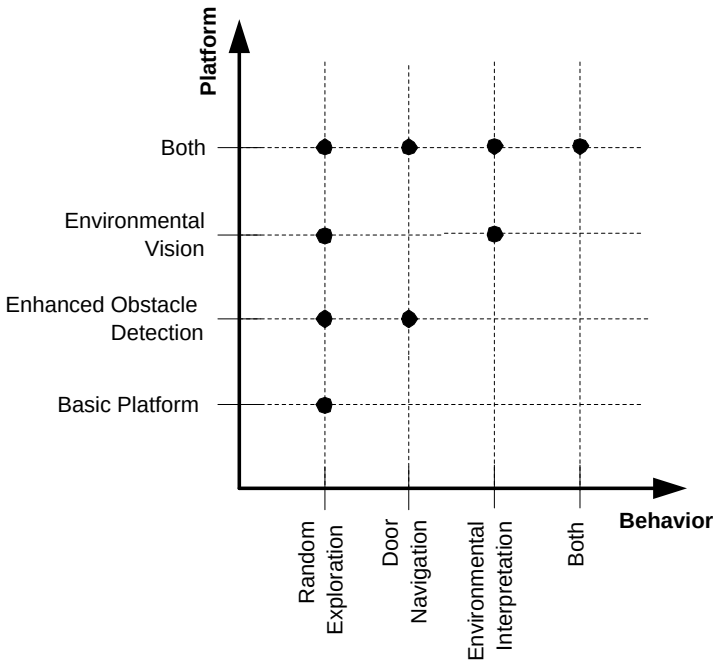

Figure 10: A possible 2-dimensional view of the robot product-line

These combinations of behaviors might require several different sets in the hardware domain, which will have sub-families that define, for example, robots with grippers, robots with bumpers, robots that have radio communications devices, and so forth. Thus, it is generally not the case in the full domain that a behavior will require exactly one subset in the hardware dimension or that the behavior and hardware dimensions have the same structure. By defining the intersection of the hardware dimension with the behavioral dimension, we define which family members are viable and which are not.

The division of the system into behavioral and hardware dimensions is a classical one which; however, these are not the only two dimensions possible. For instance, performance, for example, battery life, might be modeled as a separate dimension of the system.

8 Evaluation

The structuring technique presented results in the creation of more families within the domain than with a traditional approach. However, these sub-families are more cohesive and simpler than would be the case if we created just one top level-family. We believe that this provides several benefits. First, the top-level family can now be much broader than was previously possible. Second, the overall family can be expanded and contracted by adding and subtracting sub-families. Finally, these techniques will allow a family to be more easily refactored as the definition of the family evolves over time.

The ability to draw a larger product family was an essential requirement for the structuring technique. This grows out of our own experiences with mobile robotics [11, 23], where we had difficulty in applying the product family approach. This difficulty stems from the fact that the mobile robotics domain is both *n-dimensional* and *hierarchical*.

The mobile robotics domain breaks down along two clear dimensions: the hardware platform and the desired behavior. Each hardware platform conforms to a basic specification: it can move forward and backward, turn left and right, sense whether or not an object is in front of it. The hardware platform may also be equipped with a variety of sensors and actuators that give it additional capabilities; and, the various sensors differ greatly in the speed and accuracy with which they provide information. Thus, on the hardware side, there are many different configurations that must be modeled.

On the behavior side, we can imagine that a basic behavior might be a random exploration where the primary goal of the robot is collision avoidance and recovery. More complex behaviors can be added, for example, wall following, going through doors, and finding particular objects. Furthermore, those behaviors may be composed and combined to form a composite behavior. We might envision a behavior which includes the door

navigation, a wall following behavior, and a high-level planner. The high-level planning behavior needs to communicate with the random exploration, door navigation, and wall following to direct the robot towards high-level goals. However, if the robot collides with an obstacle, then the lower level behavior will take over and recover from the collision. Thus structure of the behavioral dimension is much different from the hardware dimension and resembles Brooks' subsumptive architecture [7].

Certainly, a domain such as mobile robotics which absolutely requires n-dimensional and hierarchical product families will necessarily be more complex than a domain that does not require these techniques. Nevertheless, any domain can benefit from reuse of the artifacts at the top of the family hierarchy and a more traditional cost-benefit will exist towards the leaves of the family (along each particular dimension). Even in a domain such as the FGS, the requirements benefit from the ability to clearly separate the concerns of the various modes and denote constraints specifically to when two modes occur together.

Another benefit of the technique is the ability to expand and contract the family as necessary. This ability is essential because it allows a more incremental development of product-lines than is facilitated by current approaches. Furthermore, it facilitates *family refactoring*; that is, the family can be redefined more easily as the product line evolves over time. Thus, this structuring technique has much potential to increase the usefulness of the product family approach.

One of the barriers to traditional product family approaches is that the whole organization must change to accommodate product-line oriented development. Many resources are required to develop the domain engineering support for the entire product line while at the same time continuing to produce products for existing customers. Our approach allows an organization to start out with a high-level product family and reuse just a few key pieces between the major product areas. As the payoff from this reuse makes more organizations resources available, the organization can then afford to make the family more rich (by refactoring and/or adding sub-families) and thus achieving more payoff from the effort.

Of course, these benefits do not come for free. The broader and more flexible view of product families allowed by our techniques will result in families which are more complex than traditional families. In addition, because of this broader view, it may be more difficult to determine what constitutes a viable family under our approach. Almost anything is related in some fashion or other and it may be difficult for organizations to decide when to define an encompassing family for a particular group of subfamilies. Nevertheless, we feel that these techniques hold promise and may serve to advance the frontiers of product-line engineering.

The cost-benefit analysis of our product-line engineering approach is more difficult because one must not only consider the cost of developing domain engineering support

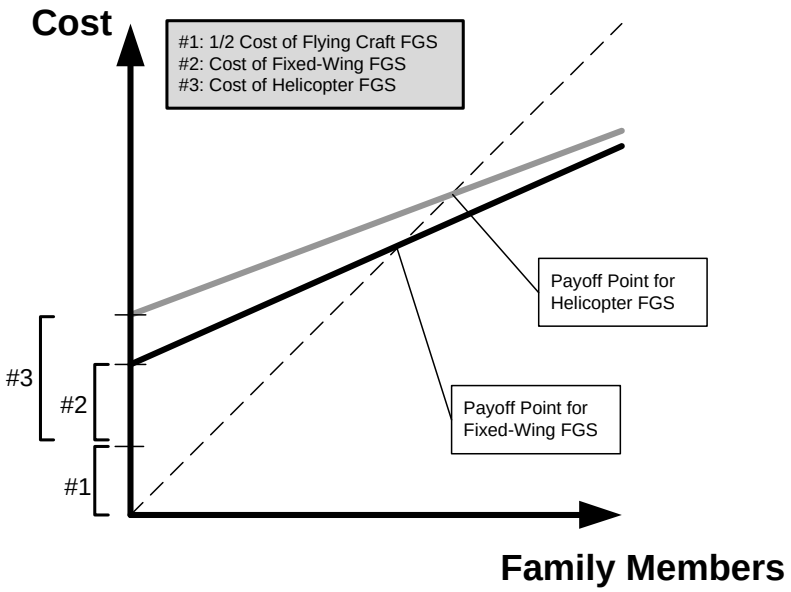

Figure 11: Cost-benefit of the FGS Family

of the particular sub-family in which the member resides, but also all sub-families above that one in the product family hierarchy. Suppose that we wanted to build a family of FGS systems for both fixed-wing aircraft and helicopters. The cost-benefit analysis for this family is shown in Figure 11. To build either a fixed-wing aircraft or a helicopter, we must have built the assets in the Flying Craft FGS family; therefore, we can amortize the cost of the Flying Craft FGS over the fixed-wing and helicopter families. This is the cost #1 in the figure. Next, if we want to build the assets for the fixed wing family, we must spend some additional amount over an above the shared cost for the Flying Craft FGS. This is noted by the cost #2 in the figure. Once we know what both of these costs are, we can determine how many fixed-wing FGS systems we must build in order for the family development effort to be justified. However, the cost of building the helicopter assets may well be *different* from the cost of building the fixed-wind assets (this is cost #3 on the figure). Therefore, if the helicopter assets are more expensive to construct we will have to build more of the helicopter FGS members to justify the costs. As the structure of the family becomes more complex, for example, through the creation of a deeper hierarchies and/or the use of multiple dimensions with constraints between them, this relationship will become more complex.

9 Conclusions and future work

In this paper, we have reflected on some current issues with product-line engineering and presented our attempts towards extending the product family approach to better address these issues as well as support what we call *n-dimensional* and *hierarchical* product families.

We examined problems that we and others have had in developing product families. We concluded that the difficulties in modeling near-commonalities and variability dependencies stemmed more from the structure of the domain itself and the lack of an ability to adequately capture that structure.

Further, we presented a simple structuring technique based on set theory. This allows the analyst to capture the structure of the domain and not be biased by implementation or design concerns. This approach allows thinking about *n-dimensional* and *hierarchical* product families and encourages many different views of the system.

We presented a limited examples of the approach, based in the flight guidance system and mobile robotics domains.

In the future, we intend to continue developing this approach to product-line engineering. We have not addressed in this work how such a product family structure would be illicited nor have we addressed how the structures identified at this early phase of product family development might support the work on product families being done in the architecture community on product family design and implementation. Furthermore, in the future, we intend to provide a more detailed description of the formal foundations of this approach and how it could be leveraged if the family members were described in a formal specification language.

Acknowledgements

The authors wish to thank Steven P. Miller at the Rockwell-Collins Advanced Technology Center for his thoughtful comments and encouragement with regards to this work as well as his insights on how the structuring techniques presented here might be applied to the systems manufactured by Rockwell-Collins, Inc.

References

[1] Activmedia robotics website. Makers of the Pioneer robot. http://www.activrobots.com/.

[2] Mark A. Ardis and David M. Weiss. Defining families: The commonality analysis. In *Nineteenth International Conference on Software Engineering (ICSE'97)*, pages 649–650, 1997.

[3] D. Batory and S. O'Mally. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[4] L. Baum, M. Becker, L. Geyer, and G. Molter. Mapping requirements to reusable components using design spaces. In *The Fourth International Conference on Requirements Engineering (ICRE'00)*, June 2000.

[5] Lothar Baum, Lars Geyer, Georg Molter, Steffen Rothkugel, and Peter Sturm. Architecture-centric software development based on extended design spaces. In *Development and Evolution of Software Architectures for Product Families: The Second International Workshop on Development and Evolution of Software Architectures for Product Familes (ARES)*, number 1429 in Lecture Notes in Computer Science, pages 197–204. Springer, February 1998.

[6] J.L. Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(8):711–721, August 1986.

[7] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[8] Lisa Brownsword and Paul Clements. A case study in successful product line development. Technical Report CMU/SEI-96-TR-016, Software Engineering Institute, Carnegie-Mellon University, October 1996.

[9] G. Campbell, J O'Connor, C. Mansour, and J. Turner-Harris. Reuse in command and control systems. *IEEE Software*, 11(5):70–79, September 1994.

[10] Grady H. Jr. Campbell, Stuart R. Faulk, and David M. Weiss. Introduction to synthesis. Technical Report INTRO-SYNTHESIS-PROCESS-90019-N, Software Productivity Consortium, Herdon, VA, 1990.

[11] Debra M. Erickson. Structuring formal requirements specifications for reuse: A mobile robotics case study. Masters Project, University of Minnesota, April 2000.

[12] Stuart R. Faulk. Product-line requirements specification (PRS): An approach and case study. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering (RE'01)*, pages 48–55, August 2001.

[13] H. Gomaa. Reusable software requirements and architectures for families of systems. *Journal of Systems and Software*, 25(3):189–202, August 1995.

[14] Juha Kuusela and Juha Savolainen. Requirements engineering for product families. In *Proceedings of the Twenty-Second International Conference on Software Engineering (ICSE'00)*, pages 60–68, June 2000.

[15] W. Lam. Creating reusable architectures: Initial experience report. *ACM SIGSOFT Software Engineering Notes*, 22(4):39–43, 1997.

[16] W. Lam, J.A. McDermid, and A.J. Vickers. Ten steps towards systematics requiremens reuse. *Requirements Engineering*, 2(2):120–113, 1997.

[17] Thomas G. Lane. Studying software architecture through design spaces and rules. Technical Report CMU/SEI-90-TR-18, Software Engineering Institute, Carnegie Mellon University, November 1990.

[18] Robyn R. Lutz. Extenting the product family approach to support safe reuse. *Journal of Systems and Software*, 53:207–217, 2000.

[19] Steven P. Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 44–53, 1998.

[20] J. Neighbors. The draco approach to constructing sofware from reusable components. *IEEE Transactions on Software Engineering*, 10(5):564–574, 1984.

[21] D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, March 1976.

[22] R. Prieto-Diaz. Domain analysis: An introduction. *ACM SIGSOFT Software Engineering Notes*, 15(2):47–54, 1990.

[23] Jeffrey M. Thompson and Mats P.E. Heimdahl. Extending the product family approach to support n-dimensional and hierarchical product lines. In *The Fifth IEEE International Symposium on Requirements Engineering*, August 2001.

[24] David M. Weiss. Defining families: The commonality analysis. Technical report, Lucent Technologies Bell Laboratories, 1000 E. Warrenville Rd, Naperville, IL 60566, 1997.

[25] David M. Weiss and Chi Tau Robert Lai. *Software Product Line Engineering: A Family-Based Software Development Process.* Addison-Wesley, 1999.