Product-Line Requirements Specification (PRS): an Approach and Case Study

Stuart R. Faulk
University of Oregon
faulk@cs.uoregon.edu

Abstract

Software product-line engineering can provide significant gains in quality and productivity through systematic reuse of software's conceptual structures. For embedded safety- or mission-critical systems, much of the development effort goes into understanding, specifying, and validating the requirements. If developers can re-use rather than re-do requirements for families of similar systems, we can improve productivity while significantly reducing the opportunity for requirements errors.

This paper describes a systematic approach to developing a Product-line Requirements Specification (PRS) for such systems. The PRS explicitly represents the family's common requirements as well as the allowed variations that distinguish family members. When completed, the PRS definition also supports generation of well-formed Software Requirements Specifications (SRS) for members of the product line. We describe a process for developing a PRS starting from an analysis of a program family's commonalities and variabilities. The approach is illustrated with examples from a case study of a real family of systems, the Rockwell Collins Commercial Flight Control System product-line.

1. Introduction

Much of the effort of building complex software systems goes into understanding, specifying, and validating system requirements. For mission- and safety-critical systems, requirements errors represent a major source of development problems.

Prior work in product-line engineering has shown that we can substantially increase productivity while decreasing errors by systematically re-using (rather than re-creating) the work products for families of systems where system requirements are sufficiently similar (e.g., [1] and [3]). Embedded software for commercial product lines like printers, mobile phones, or flight-control systems are typically families in this sense.

To date, much of the product-line engineering research has focused on the reuse of work products relating to the software's architecture, detail design, and code. Our work focuses on the systematic reuse of work products relating to a software product-line's requirements.

Of particular interest is the systematic reuse of requirements for embedded, mission- and safety-critical systems. The high up-front costs incurred by their rigorous verification and validation provides an opportunity for substantial time and cost savings where requirements can be reused.

Our technical approach to requirements specification leverages prior work in "practical formal methods." Since the domain of interest includes industrial, safety-critical systems we seek to develop specifications for which properties like completeness, consistency and safety can be demonstrated, in the context of methods that are practical for real-world application [8]. To validate the approach as well as assess its practicality, we have applied it to portions of a real family of safety-critical systems, the Rockwell Collins Commercial Flight Control System (FCS) product-line [10].

Our long-term goal is to develop a systematic approach to specifying requirements for embedded-system product lines, then rapidly generating demonstrably correct requirements specifications for applications in the product line. This paper outlines one approach to this objective and illustrates it with examples from the FCS family.

To meet these long-term objectives, we have constrained the scope of our approach. By exploiting existing requirements methods and tools, ([7], [8]) we limit the scope to embedded applications and to the kinds of requirements directly addressed by the underlying models. Further, we have not attempted to provide general mechanisms for capturing variability; rather, we have addressed such mechanisms to the extent needed to capture the variability present in the real system.

2. Product-line Requirements Objectives

Our approach focuses on methods and techniques for creating a new kind of work product we call the *Product-line Requirements Specification* (PRS). Prior work suggests that two kinds of requirements documents are potentially useful in product-line engineering: a Product-Line Requirements Specification (PRS) and a Software Requirements Specification (SRS). Where an SRS describes the requirements for a single family member, the PRS specifies the requirements for a program family [12].

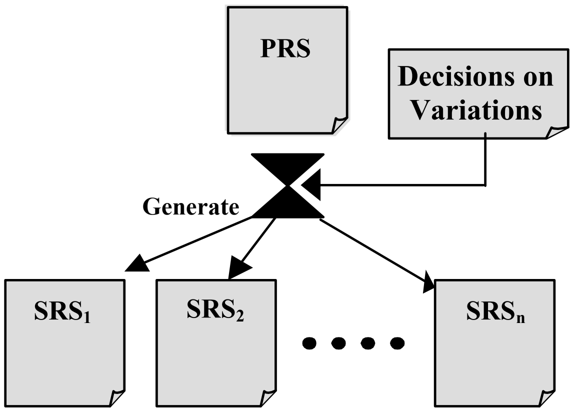


Figure 1: Member SRS derived from PRS

In a product-line development process, the PRS would be created in the *domain engineering* phase – i.e., the phase that scopes the product-line and develops the means to rapidly produce product-line members. As such, it serves two related but distinct purposes. First, it records decisions about the product line as a whole including which requirements are common to all members of the product line and which requirements may vary from one family member to the next. Second, it supports the *application engineering* phase (i.e., producing members of the product line) by providing the basis for rapidly creating the SRS for a given family member (Figure 1).

While it is possible to create a software product line without developing a distinct specification of requirements (e.g., where the domain is well-understood by all the stakeholders), producing explicit specifications of both the family, and each family member, offers potential advantages:

- The PRS provides a single document characterizing the family as a whole for domain engineers and other stakeholders interested in the product line.
- The PRS provides a place to record decisions that pertain to the family but not to any particular member of the family - e.g., how the family is likely to evolve over time or the order in which the ability to generate different family members should be implemented.
- The SRS (derived from the PRS) for a family member provides a basis for requirements validation and verification.

To address the needs particular to a PRS as well as those for any derived SRS, we identified the following PRS objectives:

1. *Specify commonalities:* The PRS should specify all of the requirements common to members of the family.
2. *Specify variabilities:* The PRS should specify which requirements may vary from one family member to the next, the range of variation, and any constraints among variations or combinations of variations.
3. *Support requirements reuse:* To support systematic reuse and rapid product development, the PRS should contain the information necessary to generate the SRS for individual members of the family.
4. *Support analysis:* Requirements should be analyzable for key properties like consistency, completeness, or safety. It should be possible to systematically analyze the SRS, PRS or both for such properties.

Our work on product-line methods has shown that we can mitigate the risk of investing in product-lines by applying a systematic process (e.g., [1]). Thus, a goal of the work is to create and validate a PRS development process that can be applied in the context of an overall product-line development process (e.g., FAST [14]).

3. Related Work

This research integrates several previous lines of work: our work on software engineering processes for developing programs as families (Synthesis [4], [13]), our work on practical formal methods for embedded system requirements (SCR [8]), and our work on object-oriented requirements specification methods (CoRE [6],[7]).

Previous work on Synthesis and related technologies [14] created processes and methods supporting systematic reuse. We use the term *systematic reuse* to denote a software development process that plans for, and systematically develops, reusable work products including documentation, design and code. In particular, we exploit Commonality Analysis ([5], [14]) as a systematic process for characterizing a product line in terms of its commonalities and variabilities.

The SCR requirements research and related work has shown the effectiveness of "practical formal methods" for specifying, verifying, and validating embedded system requirements. On the practical side, the SCR method focuses on ease of use and producing specifications that are easy to understand, develop, and maintain. SCR's underlying formal model allows specifications to be (automatically) analyzed for properties like completeness and consistency as well a providing support for analyzing application dependent properties like safety.

The CoRE method [7] provides the capability to modularize SCR-style specifications – i.e., to divide the specification into relatively independent parts and control the relationships between the parts. CoRE's class structure provides mechanisms to limit dependencies and support properties like separation-of-concerns or ease-of-change in a requirements specification. In particular, CoRE provides facilities for modularization and encapsulation of requirements without unnecessarily constraining the design [6]. We exploit these capabilities to structure a PRS such that parts of the specification describing variable requirements can be changed without impacting other parts of the document.

Recent work has begun to address the development of product-line requirements including [9] and parts of [2]. However, these approaches rely on informal specification methods and do not address our concerns for formally analyzable specifications.

4. Approach

We assume that a PRS is produced as part of a product-line development process following commonality analysis. The commonality analysis process characterizes the product family by identifying the characteristics family members share (commonalities) and the ways in which family members may differ (variabilities). Output of the commonality analysis is a written specification of the product line's terminology, commonalities, variabilities, and dependencies we call the *Domain Definition*.

Our goal is to proceed, as systematically as possible, from commonality analysis to a well-defined PRS. The (idealized) process is as follows:

1. Organize the Domain Definition: organize the specification to group together requirements that vary together and separate those varying independently.
2. Create a Decision Model: create a model of the decisions that must be made to characterize a family member and any constraints on their relative order.
3. Encapsulate variations in CoRE: construct a CoRE-style requirements specification applying the information-hiding principle to localize and encapsulate related variations.
4. Define variations: specify how requirements vary as a function of the values of the variabilities.
5. Provide Traceability: trace the variabilities to conditional inclusion statements in the PRS.

Output of the process is a PRS in the form of an annotated CoRE model. Briefly, CoRE provides a set of mechanisms for structuring the mathematical relations of Parnas' Four-Variable model [11] as a hierarchy of object classes. Applying the CoRE structuring mechanisms to the SCR version of the Four-Variable model helps address our long-term goal of applying the SCR analysis and property proving tools [8].

Briefly, SCR specifies the behavioral requirements of a system as a set of relations between the quantities monitored by the system and the quantities controlled by the system. Each of the monitored and controlled quantities (e.g. pressure) is represented by a variable. The required value of each controlled quantity, over all possible system states, is then given in terms of the possible values of the monitored quantities. Intermediate variables are introduced to simplify writing the relations. These include *conditions* (predicates over system state), *events* (predicates over successive states), *terms* (expressions over one or more variables), and *modes* (state machines capturing history).

Expressed in terms of SCR's formal model, a CoRE class structure induces a partition (or set of partitions) over the *directly-depends-on* relation. In its simplest form (the one relevant here), the SCR variable A (term, mode, etc.) directly depends on SCR variable B if B appears in the expression used to define the required value of A.

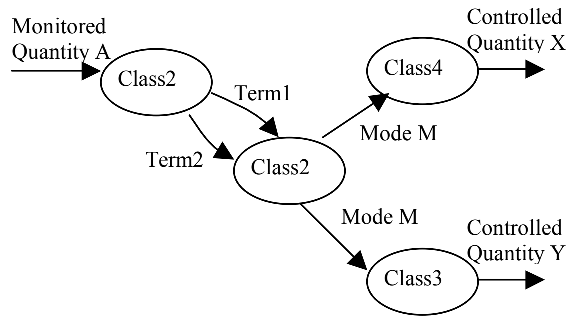


Figure 2: a simple CoRE Class Structure

A simple CoRE class structure is illustrated in Figure 2. Here the SCR model is partitioned into four classes. Class2 defines Term1 and Term2 using the Monitored Quantity A. Mode machine M is defined in Class2; its definition depends on Term1 and Term2. Controlled Quantity X depends on mode machine M as does Y. In a more complex CoRE specification, each class may be further partitioned into child classes.

CoRE's class structuring mechanism allows one to apply information hiding and abstraction to an SCR-style specification. Originally, these capabilities were introduced to control the effects of requirements changes (i.e., by encapsulating likely changes). In the current work, we use the class structure to localize and encapsulate the effects of requirements variations. The general strategy is to group together parts of the requirements that vary together and encapsulate them in a class. Conversely, the class interface is used to abstract from local variability.

This encapsulation and abstraction serves two purposes. First, it helps put the requirements that vary together in one place so they can be specified, understood or changed relatively independently. Second, it supports developing an overall class structure that is common (does not change) to all the members of the product line. While it is not, in general possible to constrain the effects of every variation to one place in the specification, these organizing principles help limit the effects of variations on the structure of the specification making it easier to read, analyze, and derive instances from.

Steps of our process preceding development of the class structure help organize the product-line's requirements so it is clear which requirements vary together and how choices among the variabilities depend on one another. We organize the Domain Definition to bring together related definitions of commonalities and variabilities. In particular, requirements that change depending on the same choice of variability are brought together in one part of the Domain Definition and organized according to their inter-dependencies. This helps impose structure on the output of the commonality analysis process and helps clarify any dependencies among the variabilities.

We then create a Decision Model for the product-line. The decision model captures the decisions an application engineer must make to define a member of the family, and the order in which he or she must make them. As such, it captures the dependencies among variations. For example, where one must make a choice about the value of variability B only if a particular value of variability A is chosen. The decision model provides much of the information needed to describe how to derive the SRS for a family member from the PRS for the family.

We annotate the PRS with embedded meta-text to specify how the requirements structure varies depending on the assignment of specific values to the variabilities. Wherever the choice of requirements depends on the value of a variability, meta-text is added to specify exactly how the requirements change depending on the value of the variability – for example, whether a particular table is included or a particular class. In developing the meta-text constructs, we have not attempted to solve the general problem of variations and dependencies. Rather, we have added constructs only as needed to address variability in our case study.

5. The Collins FCS Example

We have experimentally validated the effectiveness of our approach by applying it to a portion of a commercial avionics product line: the Collins Flight Control System (FCS) family produced by Rockwell Collins Avionics. This work has concentrated on a part of the flight guidance system, the mode control logic. An aircraft's flight guidance system compares measured aircraft state (position, speed, and attitude) with the desired state. The system generates commands to alter the aircraft's roll and pitch to minimize the difference between the measured and desire state. When engaged, the autopilot translates these commands to movements in the aircraft's control surfaces. Different flight-control laws may be used to determine how commands are generated depending on pilot choices and aircraft conditions. These relationships are implemented in the system's mode logic. The logic for determining the current mode is a significant part of the system's real-time control logic.

In related efforts, Collins Commercial Avionics developed a CoRE/SCR style requirements specification of the mode control logic for a General Aviation class aircraft [10] and a Domain Definition for the FCS family. The Collins' specifications provided input for defining the FCS product-line requirements.

The FCS product line includes the flight control software for a number of civilian light aircraft using Collins avionics. Our case study has created a PRS for a subset of the FCS-family mode control logic. While it represents only a portion of the complete FCS system, the requirements are some of the most complex in these systems. FCS family members also share the characteristics of being embedded, real-time, and safety-critical. In the following sections, we illustrate our PRS definition process using examples from the Collins FCS.

6. Organizing the Domain Definition

A well-organized Domain Definition provides the basis for creating the decision model and class structure of a PRS. Since our goal for the class structure is to localize and encapsulate requirements that vary together, we first structure the information in the Domain Definition so such requirements can be found in one part of the document. We then order the variabilities so that (where possible), requirements that depend on the value selected for some variability V are subordinated in the text. We apply the following organizational heuristics.

1. *Most common first*: requirements that hold for all members of the family or vary for all members are placed first and grouped according to subject (e.g. requirements for the *roll mode* are grouped together).
2. *Subordinate dependencies*: where inclusion of one requirement depends on another, that requirement is subordinated in the Domain Definition. Organizing the model in this way supports readability, and subsequent construction of both the decision tree and the class structure.
3. *Preserve traceability*: we assign unique identifiers to the commonalities and variabilities reflecting their type and place in the Domain Definition. This supports traceability in subsequent development steps.

Example: This organization of the domain definition is illustrated in Figure 3. Each numbered line represents a commonality (denoted by a "C") or variability (denoted by a "V"). The numbers provide traceability. Applying "most common first," commonalities of a given requirements type that hold for all members of the family are listed first followed by the related variabilities. Where commonalities or variabilities depend on which value of a variability is chosen, these are subordinated and indented. For example, the variability:

4.7.1.2 V 5.1.3 There may or may not be a roll knob to adjust the roll reference

4.7.2 Lateral Modes

4.7.2 C 1 Every *FCS* has one or more *lateral modes* exactly one of which must be active at any time.
4.7.2.C 2 Every *FCS* has a *default (basic) lateral mode*
4.7.2 V 3 The *basic lateral mode* varies from aircraft to aircraft.
4.7.2 V 4 The FCS may or may not select the basic lateral mode upon transfer of flight guidance computations.
4.7.2 V 5 The set of lateral modes varies from aircraft to aircraft.
    4.7.2.V 5.1 An FCS may or may not have a Roll Mode.
        4.7.2 C 5.1.1 Every FCS that has a roll mode uses a roll reference.
        4.7.2 C 5.1.2 The roll reference is synchronized when the SYNC switch is enabled and pressed with the flight director on.
        4.7.2 V 5.1.3 There may or may not be a roll knob to adjust the roll reference.
            4.7.2 V 5.1.3.1 The roll knob may have a detent angle of 0, 5, or 6 degrees.
        4.7.2 V 5.1.4 Roll/Heading transition angle can assume values of 5 or 6 deg.

Figure 3: Example of organized domain definition from Collins Flight Control System mode requirements

becomes a choice only if the FCS family member has a Roll Mode, hence it is subordinate to the variability:

4.7.2.V 5.1 An FCS may or may not have a Roll Mode.

This organization assumes rather simple, hierarchical dependencies between family requirements (or simple conjunctions, disjunctions or negations over such requirements). Clearly, other dependency relationships are possible such as multiple dependencies or those that cannot be partially ordered. These would require a more robust representation of the dependency relationships (e.g., sets of tables). However, in the FCS case study we found that where more complex dependencies appeared to occur, the definition could usually be decomposed and rewritten in hierarchical form. Thus, for the FCS case study this simple approach proved sufficient.

7. Creating the Decision Model

We represent the set of choices that distinguish the members of a family using a decision model. In general, a decision model specifies the choices among possible variations must be made to distinguish a family member and any constraints on the ordering of those choices.

For the FCS case study, a simple tabular representation of the underlying decision tree captures the dependencies between requirements by its form while remaining easy to read or write. The decision table is derived from our structured domain definition as follows:

- For each variability, we create a variable ranging over the possible choices.
- Where the inclusion or exclusion of requirements depends on a variable's value, these requirements are subordinated in the table.
- For each decision, the table gives a name, a brief description of the decision, the range of possible values for the decision, and the traceability number.

Figure 4 shows a piece of the FCS mode logic decision table corresponding to the subsection of the Domain Definition given in Figure 3. For example, the variability:

4.7.2.V 5.1 An FCS may or may not have a Roll Mode.

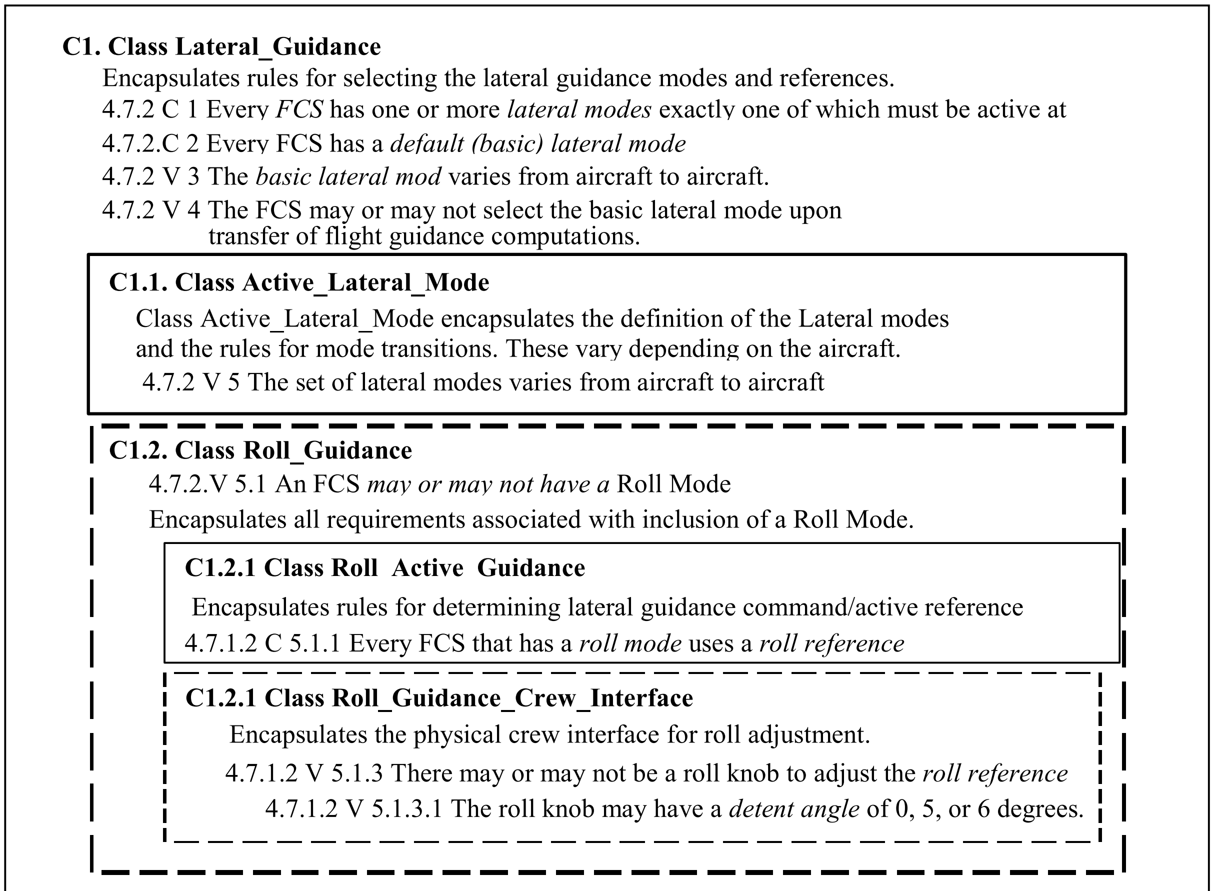
is represented in the decision table as follows:

- The variable "RollMode" is assigned to represent the variability (decision)
- A description of the necessary decision is provided ("Does the FCS have a Roll Mode?")
- Possible values of the decision are specified ("yes" and "no" for RollMode)
- The corresponding traceability number is given

In constructing the table, we carried over the subordination scheme of the Domain Definition. This helps one to determine, from the table's form, which decisions depend on others in the table. While this simple translation will not serve for all possible relations between variations, it proved adequate to the needs of the FCS family and illustrates one approach to preserving relationships and traceability between the work products.

Mnemonic	Decision			Values	Traceability
LatBasic	What is the basic lateral mode?			Roll, Heading	4.7.2 V 3
RollMode	Does the FCS have a Roll Mode?			Yes, No	4.7.2.V 5.1
RollKnob	Yes	Does the FCS have a Roll Knob?		Yes, No	4.7.2 V 5.1.3
DetentAngle	“	Yes	What is the detent angle of the Roll Knob?	0, 5, 6	4.7.2 V 5.1.3.1
RollHdgAngle	“	What is the Roll Heading Transition Angle?		5, 6	4.7.2 V 5.1.4
RollLimit	“	What is the Bank Limit in Roll Mode?		31.5, 32	4.7.2 V 5.1.5

Figure 4: Decision table for FCS Roll Mode



C1. Class Lateral_Guidance

Encapsulates rules for selecting the lateral guidance modes and references.

4.7.2 C 1 Every *FCS* has one or more *lateral modes* exactly one of which must be active at

4.7.2.C 2 Every FCS has a *default (basic) lateral mode*

4.7.2 V 3 The *basic lateral mod* varies from aircraft to aircraft.

4.7.2 V 4 The FCS may or may not select the basic lateral mode upon transfer of flight guidance computations.

C1.1. Class Active_Lateral_Mode

Class Active_Lateral_Mode encapsulates the definition of the Lateral modes and the rules for mode transitions. These vary depending on the aircraft.

4.7.2 V 5 The set of lateral modes varies from aircraft to aircraft

C1.2. Class Roll_Guidance

4.7.2.V 5.1 An FCS *may or may not have a* Roll Mode

Encapsulates all requirements associated with inclusion of a Roll Mode.

C1.2.1 Class Roll Active Guidance

Encapsulates rules for determining lateral guidance command/active reference

4.7.1.2 C 5.1.1 Every FCS that has a *roll mode* uses a *roll reference*

C1.2.1 Class Roll_Guidance_Crew_Interface

Encapsulates the physical crew interface for roll adjustment.

4.7.1.2 V 5.1.3 There may or may not be a roll knob to adjust the *roll reference*

4.7.1.2 V 5.1.3.1 The roll knob may have a *detent angle* of 0, 5, or 6 degrees.

Figure 5: Illustration of a PRS class structure

8. Encapsulating Variations in CoRE

The PRS is written in terms of a CoRE model. Input to this stage includes the Domain Definition and the decision model. The output is a meta-specification that can be used to generate a CoRE-style SRS for any family member.

Figure 5 gives a notional view of part of the FCS class hierarchy represented as a set of embedded rectangles. An embedded rectangle denotes an embedded (child) class. If the embedded class is common to all instances of its parent class, the rectangle is solid. If its inclusion depends on the value of a variability in the parent class, the rectangle is dashed. For example, the class Roll_Guidance is included in the SRS for a family member only if the Lateral_Guidance class has a roll mode. The nested class Roll_Active_Guidance will be included only if its parent is included. Each class rectangle contains a brief description of the requirements encapsulated by the class and the commonalities and variabilities encapsulated by the class definition. [1]

[1] CoRE uses a different notation but this one presents the relationships more clearly in a small example.

The class structure illustrated in Figure 5 was constructed following the heuristics discussed in Section 4. We use the Domain Definition and Decision Models to help identify requirements that vary together and dependencies among the variabilities. We then try to organize the class structure so the requirements that vary together are in the same class definition. We apply this approach iteratively where some subset set of the requirements depends on a variability. This has the effect of creating a child class that will be included or excluded in the SRS depending on the value of the variability. For example, to manage the commonalities and variabilities given in Figures 3 and 4:

1. Since all members of the family have lateral modes, a class is defined that encapsulates the rules for determining the current lateral mode and references (most common first).
2. Since every aircraft has lateral modes but the exact set of modes and the rules for transitioning between modes differs from one aircraft to the next, the class Active_Lateral_Mode is defined to encapsulate the mode class definition (encapsulating dependencies).
3. Since aircraft differ in the set of lateral modes, and the rules for selecting guidance commands for

Table 1 - mode_Active_Lateral Transition Table

From	Events	To
Any	@T(Flight_Director_On) OR @Basic_Selected	mode_**<<LatBasic>>**
Any	@T(Nav_Guidance.term_Active)	mode_NAV

`<<if RollMode=Yes then "`

Any	@T(Roll_Guidance.term_Active)	mode_ROLL

`">>`

Instantiated with the values: **`LatBasic = ROLL`** `and` **`RollMode = Yes`** results in the following table:

From	Events	To
Any	@T(Flight_Director_On) OR @Basic_Selected	mode_ROLL
Any	@T(Nav_Guidance.term_Active)	mode_NAV
Any	@T(Roll_Guidance.term_Active)	mode_ROLL

Figure 6: Generating a table for an FCS family member

different modes, we create a separate class for each possible mode that encapsulates all the requirements (common and variable) associated with that mode. This puts all of the rules associated with a mode (which are likely to change together) in one place. It also allows all requirements associated with a particular mode to be included or excluded from an SRS as a group when the mode itself is a variation. For example, since some aircraft have a Roll Mode and some do not, the class Roll_Guidance is defined to encapsulate all the requirements that should be included when there is a Roll Mode and omitted when there is not (Encapsulating dependencies).

4. In those aircraft that do have a Roll Mode, there may or may not be a roll knob that allows the crew to adjust the roll. Since the requirements associated with the roll knob are included or excluded as a group, we add a sub-class of Roll_Guidance, the class Roll_Guidance_Crew_Interface to encapsulate these requirements. It encapsulates detailed variations associated with the roll knob such as the possible detent angle.

While it is an oversimplification to say that the class structure will always follow the dependency relation, the two clearly go hand-in-hand since encapsulation in a common class is the mechanism for localizing interdependent requirements. Thus, the structure of the decision table helps guide construction of the class structure.

9. Defining Variation

In developing the FCS case study we have deliberately kept the set of extensions to CoRE/SCR syntax as small as possible. This is consistent with our long-standing principle of adding constructs to CoRE/SCR only when a clear need for them has first been demonstrated. Representing all of the variation in the FRS has required only three basic meta-text constructs as follows:

- *Text Replacement Variables:* a distinguished symbol that is used in the family specification to denote a particular variation. On instantiation of a family member, it is replaced by one of its possible values.
  Example: For the FCS example the variable LatBasic to represent the choice of the Lateral Basic mode. It has the possible replacement values "Roll" and "Heading." The variable LatBasic is used in the PRS wherever the basic mode should appear and is replaced with the name of the basic mode when the family is instantiated.
- *Decision Variables:* variables representing variabilities in the decision model. Decision variables can be used in expressions (i.e., as the test value of an if statement) to select among possible variations.
  Example: In the FCS specification, the decision about whether a variation does or does not have a roll mode is represented by the variable `RollMode` with possible values Yes and No.
- *Nested if-then-else:* A meta-text construct that is used to select portions of the specification for inclusion or exclusion based on the value of a decision variable. For the following examples we use the form: <<**`if`** *`condition`* **`then`** *`text`* *`[`***`else`** *`text]`***`>>`** where "<<" and ">>" denote the start and end of the construct, *`condition`* is a Boolean expression over one or more of the variables, *`text`* is any legal CoRE/SCR syntactic construct, and the optional **`else`** clause has the usual semantics. An **if** construct may be nested by the inclusion of dependent **if** clauses in place of the text.

These three constructs proved sufficient to both represent variation in our example and provide a basis for generating requirements for family members. A large-scale example makes this clearer but is beyond the scope of this paper. Intuitively, nested, conditional inclusion allows one to include or exclude arbitrary blocks of text, tables, etc. based on the values assigned to the variabilities. With proper encapsulation in the class structure, generation proved straightforward for all the situations we encountered.

10. Results and Conclusions

Applying the techniques described herein, we developed a PRS for a portion of the lateral modes of a Collins FCS with the property that the top-level class structure was common to all family members. Variabilities were constrained to single classes or encapsulated within a class. From the PRS, we were able to generate the corresponding SRS each FCS family member.

We were able to structure the PRS to address our goals, including systematic reuse, by applying software engineering principles like information-hiding. We do so understanding it is appropriate in developing product lines to treat common design attributes as family requirements.

While the approach worked for the FCS family it is clear that additional techniques must be added if we are to address more complex kinds of variation. What kinds of variation we will encounter in practice remains an open question. The FCS family is only one data point but, if representative, suggests that simple relationships are the more common case. We also found that apparent complexity could sometimes be resolved by careful attention to how commonalities and variabilities were expressed.

Much work remains to be done. The approach is designed to generate output that can be read and analyzed by the SCR tool set. Actually re-casting the specification in a format the tool can read remains to be done. This, however, does not address the issue of analyzing the family requirements specification itself (as opposed to a generated SRS) for properties like consistency and completeness. Obviously, these properties have a different meaning when applied to the requirements for a set of systems. Both the appropriate kinds of analysis and the means remain open research issues.

11. Acknowledgements

This work has benefited throughout from the collaboration with Rockwell Software Process Technology Group at Collins Advanced Technology Center, particularly Dr. Steve Miller. This paper has benefited greatly from thoughtful reviews from the program committee. The work would not have been possible without the generous financial and academic support from the Rockwell Science Center and Dr. Joseph Lee.

References

[1] Ardis, M. and J. Green, "Successful Introduction of Domain Engineering into Software Development," *Bell Labs Technical Journal*, July 1998, pp. 10.

[2] Bosch, J., *Design and Use of Software Architectures Adopting and Evolving a Product-Line Approach*, Addison-Wesley, 2000.

[3] Brownsword, L. and P. Clements, *A Case Study in Successful Product Line Development* (CMU/SEI-96-TR-016). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1996.

[4] Campbell, G., S. Faulk, and D. Weiss, "Introduction to Synthesis," INTRO_Synthesis_PROCESS-90019-N, Software Productivity Consortium, Herndon, VA, 1990.

[5] Coplien, J., D Hoffman, and D Weiss, "Commonality and Variability in Software Engineering." *IEEE Software* 15, 6 November/December 1998, pp 37-45

[6] Faulk, S., J. Brackett, J. Kirby, and P. Ward, "The Core Method for Real-Time Requirements," *IEEE Software*, Vol. 9, No. 5, Sept. 1992.

[7] Faulk, S., et al., *Consortium Requirements Engineering Guidebook*, Version 1, SPC-92060-CMC, Software Productivity Consortium, Herndon, VA, 1993.

[8] Heitmeyer, C., R. Jeffords, and B. Labaw, Tools for formal Specification, verification and validation of requirements, in *Proceedings of the 12th Annual Conference on Computer Assurance* (COMPASS '97), Gaithersburg, MD, June 1997.

[9] Kuusela, J. and J. Savolainen, "Requirements Engineering for Product Lines," *Proceedings, 22nd International Conference on Software Engineering*, Limerick, Ireland, 4-11 June, 2000, pp. 60.

[10] Miller, S.P., "*Specifying the mode logic of a flight guidance system in CoRE and SCR*." Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP'98), St. Petersburg, FL, March 1998.

[11] Parnas, D.L., and J. Madey, "Functional Documentation for Computer systems," *Science of Computer Programming,* v. 25(1), 41-61, Oct. 1995.

[12] Parnas, D.L, "On the Design and Development of Program Families", *IEEE Transactions on Software Engineering*, SE-2: 1-9, 1976

[13] *Reuse-Driven Software Processes Guidebook* (SPC-92019-CMC, Version 02.00.03). Herndon, Va.: Software Productivity Consortium, 1993.

[14] Weiss, David M. & Lai, Chi Tau Robert. Software Product-Line Engineering: A Family-Based Software Development Process. Reading, MA: Addison-Wesley, 1999