Two Novel Concepts for Systematic Product Line Development

Alessandro Pasetti and Wolfgang Pree
Faculty of Computer Science, University of Constance, D-78457, Constance, Germany,
e-mail: pasetti@fmi.uni-konstanz.de, pree@acm.org

Key words: Product Line, Software Framework, Framelet, Design Pattern

Abstract: Framelets and implementation cases are new concepts to manage the complexity of product line development. Framelets are “small product lines” that address, as self-standing units, specific problems within the product line. They make no assumptions about application execution control and are designed to be composed with each other. A product line is obtained as a combination of framelets. Framelets simplify the development and extension of product lines, and make their integration with other product lines and with other software simpler. Implementation cases are introduced as ways to continuously monitor the adequacy of an evolving product line design. They describe an aspect of the product line instantiation process by specifying how an architectural feature for an application can be implemented using the constructs offered by the product line. Implementation cases narrow the abstraction gap between product line and application by forcing designers to think about the reification of the abstractions they are creating while at the same time giving them the opportunity to test the adequacy of these abstractions. Implementation cases can also be used to specify a product line or as cookbook-style recipes to document its usage. The discussion is made in the framework of a project with the European Space Agency to design a product line for satellite on-board software[1]. Heuristics for defining framelets and implementation cases are derived from the experience gained on this project and discussed in the paper.

[1] The views expressed in this paper are those of its authors only. They do not in any way commit the European Space Agency or reflect official European Space Agency thinking.

1. INTRODUCTION

Product lines are notoriously complex constructs. Their complexity has two aspects. *Quantitatively,* complexity stems from the sheer size of typical product lines that may encompass hundreds of classes embedded in an often tangled web of interconnections and semantic relationships. *Qualitatively,* it arises from their high level of abstraction, itself a consequence of their attempt to model whole application domains.

Complexity translates into long development times – it normally takes 3 to 4 times as long to build a product line as it does to build an individual application – and several design cycles before the product line reaches maturity [1,2,18,19,20]. Identifying ways of managing this complexity is one of the key objectives of product line research.

Recently, the suggestion was advanced [22] to tackle the quantitative aspect of product line complexity by developing product lines as combination of smaller – and hopefully more manageable – architectural units called *framelets*[2].

Earlier this year, the authors started a project for the European Space Agency to develop a product line for the Attitude and Orbit Control System (AOCS) of satellites. AOCS's are a complex, real-time, mission-critical systems and building a product line to model them promised to be a challenging assignment making this project a good candidate for testing the framelet approach. The AOCS product line was accordingly designed as a collection of framelets.

Implementation cases were introduced during the AOCS project to address the qualitative complexity of product lines. They are intended to narrow the abstraction gap between the product line and individual applications by modeling selected aspects of the product line instantiation process. They can be used to specify the product line, to check the adequacy of its design during the development process, and, finally, to provide cookbook-style recipes of how to use the product line.

The objective of this paper is to introduce the framelet and implementation case concepts, to describe how they were applied to the AOCS project, and how they can be generally applied to product line development.

Sections 2 to 4 present the AOCS and the AOCS project. Sections 5 and 6 introduce framelets and implementation cases and their application to the AOCS project. Section 7 explains how these concepts were integrated in the development process for the AOCS product line. Section 8 discusses related work.

[2] The name "framelet" derives from the term "framework" which is sometimes used as synonymous to "product line".

2. THE ATTITUDE AND ORBIT CONTROL SYSTEM (AOCS)

The structure of an AOCS is shown in figure 1. The AOCS is an embedded hard real-time control system. Its chief task is to periodically collect measurements from a set of sensors and convert them into commands for a set of actuators. The AOCS interacts with a ground control station from which it receives commands (*telecommands*) and to which it forwards housekeeping data (*telemetry*). Like all satellite systems, the AOCS must remain fully operational in the presence of any single failure and must survive prolonged periods of ground station outage. Robustness to faults and autonomy require the AOCS to perform failure detection and failure recovery actions.

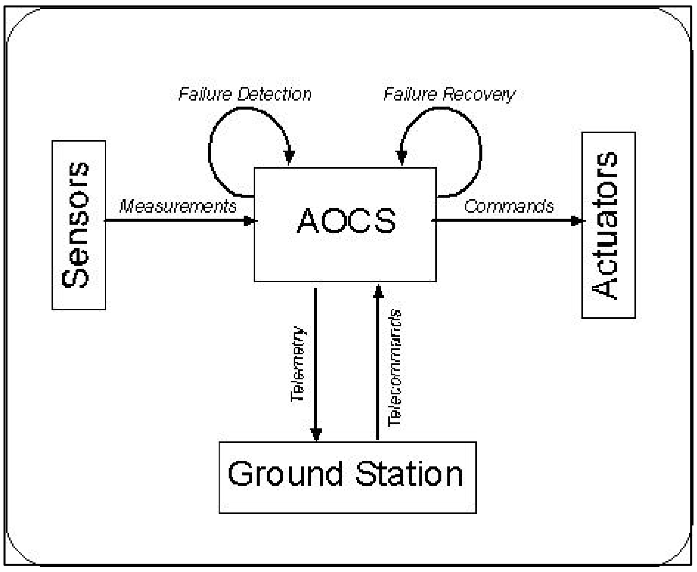

Figure 1. Structure of an AOCS System

The AOCS software is usually organized as a set of statically defined tasks running under the control of a cyclical scheduler.

The AOCS code size varies from mission to mission but normally lies between 10000 and 20000 lines of very compact code. This will certainly increase in the near future as more powerful processors are qualified for use in space. The software is normally written in Ada83 or C with sprinklings of assembler.

The challenges in building an AOCS product line lie in the hard real-time aspects of the AOCS, in its high degree of reliability, and in the transition from a conventional to an object-oriented design.

3. THE AOCS PROJECT

Work on the AOCS product line started in April of this year. At the time of writing (Nov. 99), the architectural design of the product line has been completed resulting in a product line comprising over 130 classes. A prototype product line will be implemented next year and tested on a breaboard satellite processor.

4. THE AOCS SOFTWARE ARCHITECTURE

Figure 2 shows the AOCS software architecture assumed by the product line.

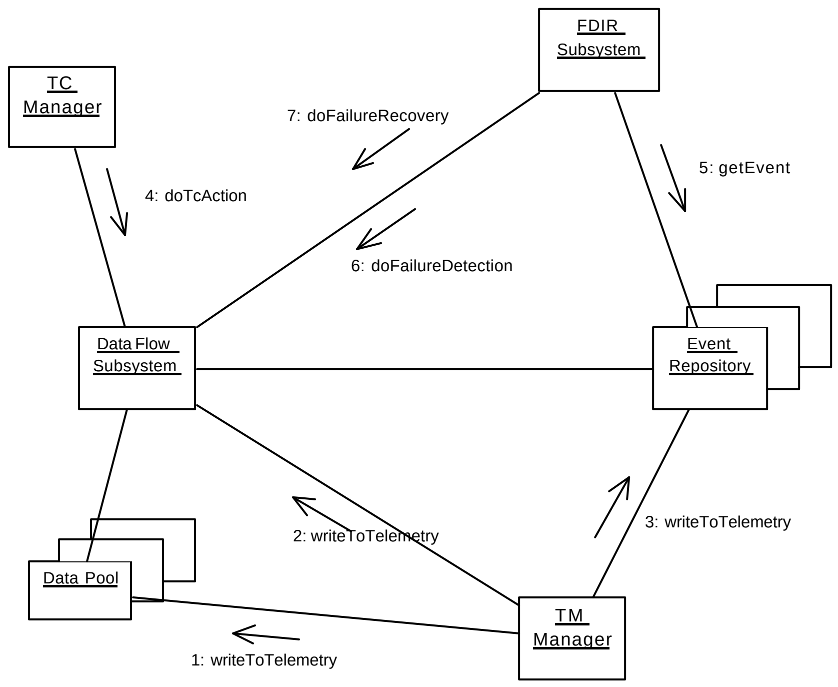

Figure 2. AOCS Software Architecture

The *telecommand manager* periodically checks whether any telecommands have been received and, if so, it asks them to execute themselves. The telemetry manager periodically issues messages to other objects asking them to write their internal state to the telemetry stream. The *data flow subsystem* handles the cyclical flow of data from sensors through attitude and orbit controllers to actuators. The *FDIR subsystem* is responsible for failure detection, isolation and recovery. Components communicate through shared memory areas. Cyclical data are stored in shared *data pools* and asynchronous events are stored in shared *event repositories*.

5. THE FRAMELET CONCEPT

Framelets are essentially small product lines that address, as self-standing units, specific problems within the product line. The product line itself is obtained by combining the framelets. The defining features of framelets are:

– *Small size:* product lines normally consist of a large number (sometimes running into several hundreds) of interrelated and cooperating classes. Framelets are smaller, typically including a dozen classes or so.
– *No execution control assumptions*: unlike product lines, that often assume that they have control of the application execution, framelets make no such assumption and are designed to be amenable to integration with each other and with other software.
– *Self-standing*: although framelets are intended to be integrated together to build a full product line, they should also be self-standing and in principle usable in isolation from the other framelets. This ensures that coupling between framelets is minimized and gives a rule for deciding the size of a framelet: a framelet should be as small as possible while retaining sufficient functionality to be independently usable.

Framelets address three important issues in product line development. Firstly, and as already mentioned, they provide a way of managing the *quantitative complexity* of product lines by breaking them down into smaller, loosely coupled units. This type of complexity management is especially valuable in mission-critical systems – like the AOCS – where framelets shift the focus from the product line as a whole to units at a lower level of complexity thus enhancing confidence in the results of reliability analyses.

Secondly, a framelet approach simplifies the *extension* of a product line since it makes it possible to add new functionalities to the product line by adding new framelets.

Finally, a framelet-based product line is easier to compose with other product lines since it does not assume that it has control of program execution and this assumption is a common obstacle to product line integration [4].

5.1 Framelets in the AOCS Product Line

A total of 12 framelets were defined for the AOCS product line consisting of an average of about 11 classes each. Their function is briefly summarized below:

– The *Monitoring Framelet* proposes an architectural solution to the problem of monitoring an object and its properties. It enhances re-usability by decoupling the management of the monitoring process from the execution of the monitoring checks.
– The *Communication Framelet* offers the infrastructure for managing data exchanges between components. It enhances reusability by decoupling the production of data from their consumption.
– The *Data Processing Framelet* provides facilities for the sequential processing of AOCS data. It enhances reusability by providing a standard interface for components that perform sequential processing on AOCS data and by allowing easy combination of data processing blocks.
– The *Aocs Unit Framelet* defines a generic interface to all external AOCS units (sensors and actuators). It enhances reusability by decoupling the users of unit data from the units themselves.
– The *Unit Reconfiguration Framelet* proposes an architecture to handle reconfigurations of AOCS sensors and actuators. It enhances reusability by decoupling the management of unit reconfigurations from the processing of unit data.
– The *Mode Management Framelet* proposes an architectural solution to the problem of endowing components with mode-dependent behaviour. It enhances reusability by separating the mode-specific algorithms from the mode switching logic.
– The *Manoeuvre Management Framelet* offers a harness for managing AOCS manoeuvres (orbit changes, attitude slews, etc). It enhances reusability by separating the management of manoeuvres from their execution.
– The *Failure Detection Framelet* defines an architecture to handle failure detection tasks. It enhances reusability by decoupling the management of such tasks from their implementation.
– The *Failure Isolation Framelet* defines an architecture to support failure tracing in the AOCS data flow.

– The *Failure Recovery Framelet* defines an architecture to handle failure recovery tasks. It enhances reusability by decoupling the management of such tasks from their implementation.
– The *Telecommand Framelet* defines an interface for telecommands and an application-independent component to act as a telecommand manager. It enhances reusability by decoupling the management of telecommands from their execution.
– The *Telemetry Framelet* defines an interface for telemeterable objects and an application-independent component to act as telemetry manager. It enhances reusability by decoupling telemetry management from the collection of telemetry data.

5.2 Framelet Constructs

Framelets define architectural constructs that are then available for use in other framelets or directly in applications instantiated from the product line. Constructs that are defined in a framelet but used elsewhere are said to be *exported* by the framelet. Framelets export three types of constructs:

– *Components*[3]: pre-defined and configurable binary units that can be used "as is".
– *Interfaces*: sets of operations with their signatures.
– *Design Pattern*: architectural solutions for a design problem specific to the product line application domain.

Note that these constructs exist at different levels of abstraction. Components are concrete objects ready for use in an application. Interfaces are like abstract classes for which implementation in the form of an application-specific component must be provided. Design patterns are abstract architectural solutions to design problem that must be implemented by other components.

Components and interfaces are exported either "horizontally" to other framelets or "vertically" to applications that must be instantiated from the product line. Design patterns, instead, can only be exported horizontally since they do not exist at application level.

A product line consists of abstract classes together with components providing default (but overridable) implementations for (some of) the abstract classes. Thus, the abstract classes and components making up the

[3] In the present version of the AOCS product line, the "components" are essentially units of compilation exposing operations that allow them to be configured. In the future, the product line will be implemented upon a component infrastructure (probably CORBA in its TAO implementation).

product line are a subset of the interfaces and components exported by the framelets.

An example from the AOCS product line illustrates these concepts. Figure 3 shows three framelets with the constructs they export to each other (horizontal arrows) and to AOCS applications (vertical lines). The telemetry framelet, for instance, uses a design pattern defined in the mode management framelet and an interface defined in the unit framelet and it exports the `Telemeterable` interface both towards other framelets and towards applications.

Note that some constructs (eg. the event repository component) are exported both horizontally and vertically. Note also that not all framelets export all three types of constructs and not all framelets export both horizontally and vertically.

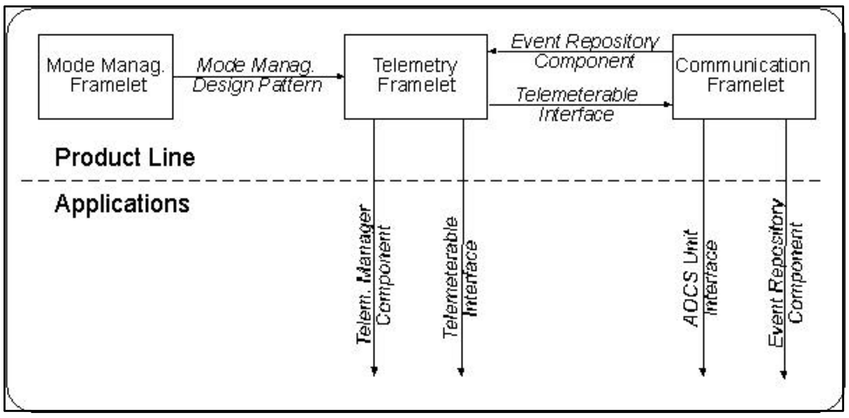

Figure 3. Example of Construct Export by Framelets

Table 1 shows the type of constructs exported by each framelet in the AOCS product line.

Table 1. Constructs Exported by AOCS Framelets

Framelet	**Exported Constructs**
Object Monitoring	Design Pattern + Interface
Component Communication	Components + Interfaces
Sequential Data Processing	Components + Interface
AOCS Unit Management	Interfaces
AOCS Unit Reconfiguration	Interface
Mode Management	Design Pattern

Framelet	Exported Constructs
Manoeuvre Management	Component + Interface
Failure Detection	Comp.+ Des.Patt.+ Interface
Failure Isolation	Components + Interface
Failure Recovery	Comp.+ Des.Patt.+ Interface
Telecommand Management	Component + Interfaces
Telemetry Management	Component + Interface

5.3 Framelets in the Design Process

Section 7 describes the development process followed in the AOCS project which could serve as a blueprint for a product line development methodology centered around framelets and implementation cases.

Framelets play a role in each stage of this development process. At the very early *system concept definition phase* they are identified (section 5.4 offers some heuristics on how this can be done) and their broad functionalities are defined.

In the following phase – the *framelet concept definition phase* – the external interfaces of the framelets are defined. This requires definition of the constructs exported by the framelets of the hot-spots exposed by them.

A product line is developed to improve reusability and each framelet should give a distinct contribution to enhancing the product line reusability. In this phase, the contribution to overall reusability of each framelet is analyzed. This helps identify redundant framelets.

In the case of the AOCS project, at the end of the system-level concept definition phase a total of 14 framelets had been identified. Going through the exercise of defining their individual contributions to reusability showed that two framelets were in fact not required thus leading to a reduction in the number of framelets from 14 to 12.

Finally, in the *framelet architectural definition phase*, the internal architecture of each framelet is worked out down to class level.

5.4 Heuristics for Framelet Identification

The breaking up of the product line into framelets is one of the crucial steps in the early part of product line design. In importance – and in difficulty – this task is comparable to that of identifying key application abstractions and mapping them to objects in conventional application design.

The experience gained in the AOCS project suggests five guidelines to facilitate this task presented in the following five subsections.

5.4.1 Mapping clusters of related requirements to framelets

In the AOCS project, the product line was derived from an analysis of a set of AOCS applications each described by a set of user requirements.

User requirements are often organized in groups of related requirements. One useful heuristic for identifying framelets is to find the requirement groupings that recur in many applications. In practice, this can often be done by inspecting the table of contents of several user requirement documents and identifying recurring sections.

Many methodologies for application development suggest that potential objects can be recognized by underlining often-recurring nouns in requirement specifications [5]. The heuristic proposed here to isolate potential framelets is similar but operates at a higher level of abstraction.

5.4.2 Building framelets around single or related hot-spots

A hot-spot (or hook) is a point where behaviour adaptation takes place [6]. Applications are instantiated from a product line by adapting the hot-spots.

The identification of hot spots is one of the early tasks in the product line design process. A major hot-spot can be the core around which a framelet is built.

5.4.3 Building framelets around design patterns

Design patterns are among the "building blocks" of product lines [16]. Applicable design patterns are identified at the beginning of the product line design process. Since design patterns typically consist of a handful of cooperating classes addressing a localized design problem without making any assumptions about execution control, they can serve as the basis for a framelet.

Note that this heuristic is related to the previous one because one important reason why design patterns are introduced in product lines is to model hot-spot variability [17].

5.4.4 Mapping tasks to framelets

Real-time applications are often organized around tasks representing separate threads of control. In most cases, tasks are created statically and therefore their number and function is defined at design time. Tasks typically encapsulate self-contained functionalities and have well-defined boundaries with each other. They are thus good candidate for framelets and one natural guideline in framelet identification is to look for typical application tasks and to map them to framelets.

Mapping framelets to tasks ensures that the framelets are functionally decoupled (thus facilitating their design) and that they do not make any assumptions about the control of their own execution which is explicitly delegated to a scheduler (thus facilitating their integration into a single product line).

Basing framelets on tasks, incidentally, solves one of the thorniest problems in the design of product lines for real-time applications. This type of product lines presents two intersecting architectural challenges: the design of the software architecture – ie. the definition of the classes and their relationships – and the design of the scheduling architecture – ie. the definition of the threads of control and of their relationships. Using a task-based approach and mapping tasks to framelets decouples these two architectural problems as it allows the system of classes to be designed independently of the scheduling policy. The latter affects the sequence and the manner in which the tasks are called but not their internal structure (with the possible exception of synchronization mechanisms for access control to shared resources).

5.4.5 Mapping abstract use cases to framelets

Abstract use cases are introduced in [7]. They are found by searching for patterns in a large number of use cases for applications in the product line domain. They embody abstract forms of behaviour that are common to many applications in the domain. Reference [7] proposes them as a way of identifying abstract classes in a product line but they could also be used to identify framelets since a framelet should ideally encapsulate one particular form of behaviour variability.

5.4.6 Framelet Heuristics in the AOCS Project

The above heuristics can be illustrated with examples from the AOCS project. Of the twelve framelets proposed for the AOCS product line, five – the manoeuvre management, telemetry, telecommand, failure detection and failure recovery framelets – correspond to clusters of requirements that will be found in virtually all user requirement documents for AOCS software.

The telemetry and telecommand framelets map directly to tasks in the AOCS software which normally devolves telemetry and telecommand management to dedicated tasks. The data processing framelet does the same as its function is to provide a component to implement attitude and orbit control algorithms and the implementation of these algorithms is usually allocated to a dedicated task.

The structure of AOCS units (sensors and actuators) is very application-specific and therefore unit management is a natural hot-spot in an AOCS product line. The AOCS unit framelet was built around this hot-spot. It consists of an abstract interface that encapsulates the generic operations that can be performed on any AOCS sensor or actuator but leaves the implementation open to individual applications.

Failure detection, failure recovery and manoeuvre management are also highly application-specific and gave rise to hot-spots in the AOCS product line which then became the basis for three framelets. In their case, however, it was possible to identify some application-independent behaviour that was packaged as components exported by the framelets to end-applications.

The algorithms used to process the AOCS data are another obvious hot-spot which was encapsulated in the data processing framelet.

Most components in an AOCS software are required to adapt their behaviour in response to changes in their environment. This effect is usually achieved by endowing them with mode-dependent behaviour: components are given control over several algorithms – one for each mode – and employ application-specific rules to select the one to be executed at any given time. The selection is a function of the state of other components. A design pattern (derived from the strategy pattern of [9]) was devised to provide components with mode-dependent behaviour and became the basis of the mode framelet.

Monitoring of component properties is another common task in AOCS systems and for it, too, a generic design pattern was devised that became the basis of the monitoring framelet.

Table 2. Heuristics for AOCS Framelets

Framelet	**Heuristics**
Object Monitoring	Design Pattern
Component Communication	Hot-Spot
Sequential Data Processing	Hot-Spot + Task
AOCS Unit Management	Hot-Spot
AOCS Unit Reconfiguration	Hot-Spot
Mode Management	Design Pattern
Manoeuvre Management	Hot-Spot + Requirement Cluster
Failure Detection	Hot-Spot + Requirement Cluster

Framelet	**Heuristics**
Failure Isolation	Hot-Spot
Failure Recovery	Hot-Spot + Requirement Cluster
Telecommand Management	Task + Requirement Cluster
Telemetry Management	Task + Requirement Cluster

Table 2 shows the heuristics used for each framelet in the AOCS product line. Note that the last heuristic – based on abstract use cases – was not used in the AOCS project because AOCS systems are not normally described by use cases.

5.5 Expressing the Framelet Design

The problem of expressing the design of a framelet is akin to the more general problem of expressing the design of a product line which remains an area of on-going research [8]. In the AOCS project, the framelets were described in informal language supported by UML diagrams and pseudo-code. This descriptive style, however, is unsuitable for purposes of design review and documentation. For the second part of the project, more formal modeling guidelines have been developed.

The following aspects of a framelet need to be modelled:

– The constructs exported by the framelet
– The framelet hot-spots
– The framelet internal architecture

As discussed in section 5.2, framelets export three types of architectural constructs: abstract classes, components and design patterns.

Abstract classes and components can be described with any of several available formalisms for object-oriented design modeling. In the AOCS project, UML is used but other choices are possible.

Description of the framelet design patterns is problematic since no accepted formalism exists for design pattern modeling. In some cases, the design patterns come from a pattern catalogue in which case reference can be made to the description in the catalogue. When new patterns are instead used, they can be described in informal language following the model of [9] that has becomes a *de facto* standard for design pattern description.

Description of framelet hot-spots is also problematic and for the same reason: a lack of accepted modeling formalisms. Here, two levels of description are proposed corresponding to two different phases in the framelet design process (see section 7). At the early concept-definition phase, hot-spots are not yet mapped to concrete syntactic constructs and cannot be easily modeled through class diagrams. A systematic classification (partially derived from [13] and [14]) is therefore proposed that provides the following information for each hot-spot:

– *Visibility level:* two values are possible: *framelet-level* or *product line level.* Some hot-spots exist only at the framelet level as they are intended to provide hooks for other framelets during the product line assembly process. Such hot-spots are said to have a framelet-level visibility. Other hot-spots instead carry over to the product line as they are intended as hooks where application developers can insert application-specific items during the application instantiation process. Such hot-spots are said to have a product line-level visibility.
– *Adaptation time*: hot-spots provide a means of adapting framelet behaviour. Two adaptation times are possible *compile-time* and *run-time*, depending on whether behaviour adaptation is done statically (eg. using inheritance or template instantiation) or dynamically (eg. using object composition).
– *Adaptation method:* a hot-spot is a point where framelet behaviour can be adapted. The following adaptation mechanisms are possible: enabling\disabling a feature; tuning an existing feature; replacing a feature; augmenting a feature; adding a new feature
– *Pre-defined options*: in some cases, the framelet itself offers pre-defined options for a hot-spot. For instance, the control algorithm in an satellite attitude controller component is clearly a hot-spot because different satellites have different types of control algorithms. However, the product line may offer some plug-in components implementing common types of control algorithms.

At the later architectural design phase – when the internal framelet architecture takes shape – hot-spots acquire a more specific form as they become overridable methods or plug-in points for components. Hence, at this stage, they are amenable to formal description using a formalism such as UML suitably extended for this purpose. In the AOCS project, the approach of reference [10] is adopted where UML stereotypes are introduced to model the following adaptation mechanism:

– *variation methods* (methods that can be overridden during product line instantiation)
– *extension classes* (classes whose interface is extended during product line instantiation)

– *extension interfaces* (classes or interfaces from which concrete classes are derived during product line instantiation)

These adaptation mechanisms cover all forms of framelet hot-spots.

Figure 4 shows the framelet design description techniques as a function of the design phase. The design phases are as described in section 7.

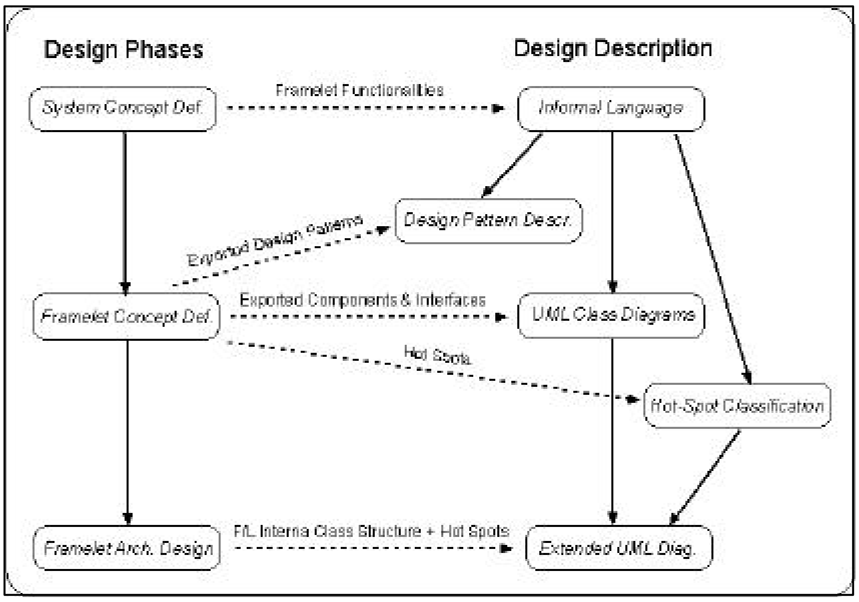

Figure 4. Framelet Design Description

5.6 Product Line Design Modelling

The modelling guidelines laid down in the previous section, address the problem of describing individual framelets. They need to be complemented by rules to model the product line as a whole.

In a framelet-based approach, the product line is obtained as a combination of framelets. To describe the product line architecture therefore means to describe the interactions of the framelets. Framelets interact through the architectural constructs they import from each other (see section 5.2) and through the framelet-level hot-spots. There are at present no established formalisms to describe these kinds of framelet-level interactions.

A suitable formalism should convey, at the very least, the information shown in figure 3 for a subset of the AOCS framelets: it should show the framelets with the constructs they export either to other framelets or to end-applications. A supporting tool should allow users to select framelet subsets or to zoom in on the internal architecture of individual framelets (which, as discussed in section 5.5, can be represented using UML).

Figure 3 does not explicitly portray framelet hot-spots although some hot-spots are shown implicitly (eg. exported interface are obviously hot-spots). Its formalism should therefore be augmented with a descriptions of other types of hot-spots such as plug-in points in exported components.

In summary then, available modelling techniques – complemented by formal classification schemes – are suitable to describe individual framelets but there are no good ways to represent the interactions of framelets and hence the product line architecture as a whole.

6. IMPLEMENTATION CASES

As the design of the framelets for the AOCS project was proceeding, the need was felt to check its adequacy without having to wait for the prototyping phase. The concept of *implementation case* was introduced to cover this need.

A product line is a tool to help developers rapidly build an application within the product line domain. An implementation case describes an aspect of this application instantiation process by specifying how a component, an architectural feature, or a functionality for an application in the product line domain can be implemented using the constructs offered by the product line.

An example from the AOCS project will clarify this definition. The AOCS product line is a tool to assist the development of AOCS applications. An important functionality of any AOCS application is the ability to perform satellite attitude and orbit control. Accordingly, the following implementation case was formulated: “build a component implementing the attitude and orbit control algorithms”. This implementation case was then worked out by showing how the constructs offered by the product line can be combined to build the required component.

Thus, implementation cases define an objective for a localized instantiation action. They are said to be *worked out* when they are accompanied by a description of how their instantiation objective can be achieved using the product line.

When product line design is completed, implementation cases can be worked out in detail, essentially resulting in cookbook-style recipes for using the product line. When the design is still underway, only partial working out of the implementation case is possible since not all the product line constructs are yet available or finalized. However, even at early design stages, going through the implementation cases remains very useful because the exercise can reveal shortcomings in the already defined constructs and can point towards constructs that are still needed.

In the AOCS project, implementation cases were defined early in the design and were then gradually worked out as the design progressed. Typically, whenever a new construct was introduced, its effectiveness was tested by working out an implementation case that used it. Where necessary, new implementation cases were introduced to cover the functionalities introduced by newly defined constructs. This process of refinement of implementation cases was the single most important source of changes in the framelet design and it is believed that it replaced at least one iteration cycle in the product line design.

Thus used, implementation cases address the qualitative aspect of product line complexity which stem from their high level of abstraction. They force designers to think about the reification of the abstractions they are creating while at the same time giving them the opportunity to test their adequacy in concrete application development settings.

The term “implementation case” was coined by analogy to the term “use case” as employed by some methodologies for application development. A use case describes the way an application is intended to be used [12]. Use cases cannot be defined for a product line because a product line is *not* a working application and it is not *used* in the same sense in which an application is used. An *implementation case* is its equivalent in the sense that a product line is a tool to help implement applications and implementation cases describe how a feature of an application can be imple mented.

6.1 Implementation Cases in the Design Process

In the AOCS project, implementation cases were used primarily to continuously check the adequacy of the product line design. During the system-level concept definition phase, implementation cases were defined at a very high level by simply describing the objective of the instantiation action they represented. They were then periodically revisited during the design process and progressively refined to reflect the advancing state of the product line definition. At the end of the architectural design phase, they were described at the pseudo-code level. Typically, the pseudo-code was intended to demonstrate how pre-defined product line constructs – components and abstract classes – could be used to achieve the objective prescribed by the implementation case.

Implementation cases could play at least two additional roles in the product line development process. Firstly, like use cases, they could help describe the product line since a sufficient number of them could cover all the functionalities of a product line and could thus be used as a way of specifying it. The acceptance test for the product line then becomes its ability to achieve the instantiation objectives specified by the implementation cases. The ease with which this can be done is a measure of the quality of the product line: a well-designed product line should offer abstractions and components that let users quickly and naturally work out the implementation cases.

Secondly, implementation cases can become part of the product line user manual. At the end of the product line development process, they are available as commented pseudo-code. They are therefore ready for inclusion in the product line user manual where they can serve as cookbook-type recipes showing how small applications or fragments of applications can be constructed.

6.2 Identification of Implementation Cases

Implementation cases should cover all the functionalities offered by the product line. In the early concept definition phase, an implementation case should be defined for each framelet. Subsequently, as the framelet design matures, implementation cases should be defined to cover all the constructs exported by the framelets (see section 5.2) and all the hot-spots exposed by them. Full coverage is verified by generating a traceability matrix mapping the framelet constructs and hot-spots to the implementation cases.

6.3 Implementation Case Description

There is no formalism for describing implementation cases. In the AOCS project, implementation cases are described in an informal but systematic manner. UML cooperation diagrams support the description. For each implementation case, the following information is provided:

- Implementation case objective
- Implementation case description
- Framelets involved in implementation case
- Framelet constructs involved in implementation case
- Framelet hot-spots involved in implementation case
- Related implementation cases
- Pseudo-code showing how implementation case is worked out

Note that in the approach proposed here implementation cases are defined incrementally during the design process. Hence, the information items listed above are not all supplied at the same time. They are instead provided gradually as the product line design matures and the constructs for working out the implementation cases become available. The degree of maturity of implementation case description can be used as a measure of the maturity of the product line design.

6.4 Implementation Cases in the AOCS Project

At the time of writing, 14 implementation cases are defined for the AOCS product line. An example definition that follows the description guidelines of section 6.3 is shown in table 3.

Table 3. First Implementation Case Example

Objective
Implement an attitude slew manoeuvre

Description
Attitude slews are common types of manoeuvres performed by satellites. The AOCS product line encapsulates manoeuvres in components. This implementation case shows how to build a component encapsulating an attitude slew manoeuvre.

Framelets
Manoeuvre Management Framelet

Framelet Constructs
`AocsManoeuvre` Interface
(exported from Manoeuvre Management Framelet)

Framelet Hot Spots
AOCS Manoeuvre Definition
(exposed by Manoeuvre Management Framelet)

The level of description is that adequate to the early phase of the framelet design process. As the framelet design proceeded, pseudo-code was added to concretely show how the attitude slew manoeuvre component is built. The pseudo-code is not shown as understanding it would require more background on the AOCS product line than can be provided here.

Table 4 shows a second example of implementation case from the AOCS project. The level of description is the same as in the previous example. Note how this implementation case *extends* the previous one in the sense that it uses its output and logically follows it up. The relationship of extension among implementation cases is conceptually similar to the relationship of extension for use cases.

Table 4. Second Implementation Case Example

Objective	Build a telecommand to perform an attitude slew manoeuvre
Description	Attitude slews are normally started by a ground command (telecommand). This implementation case shows how to build a telecommand to perform an attitude slew manoeuvre. It is assumed that the attitude slew manoeuvre is encapsulated in the component built in the implementation case of table 3.
Framelets	Telecommand Framelet Manoeuvre Management Framelet
Framelet Constructs	`Telecommand` Interface (exported from Telecommand Framelet) `AocsManoeuvre` Interface `ManoeuvreManager` Component (exported from Manoeuvre Management Framelet)
Framelet Hot Spots	Telecommand Definition (exposed by Telecommand Framelet)
Related Implementation Cases	Attitude Slew Manoeuvre Implementation Case (this implementation case uses the component built in the attitude slew manoeuvre implementation case)

7. DESIGN PROCESS

Figure 5 shows the design process followed in the AOCS product line project. Design began with a system-level concept definition phase whose main tasks were:

– definition of product line functionalities
– definition of overall design constraints
– identification of product line hot-spots
– identification of applicable design patterns
– identification of framelets

The design then split into two parallel branches with the left branch being in turn subdivided into sub-branches each corresponding to a framelet.

Framelet development proceeded in two phases. In the *framelet-level concept definition phase* the external interfaces of the framelet were defined in terms of the architectural constructs they exported (section 5.2) and of the hot-spots they offered. In the *framelet architectural design phase,* the internal class architecture of the framelets was defined.

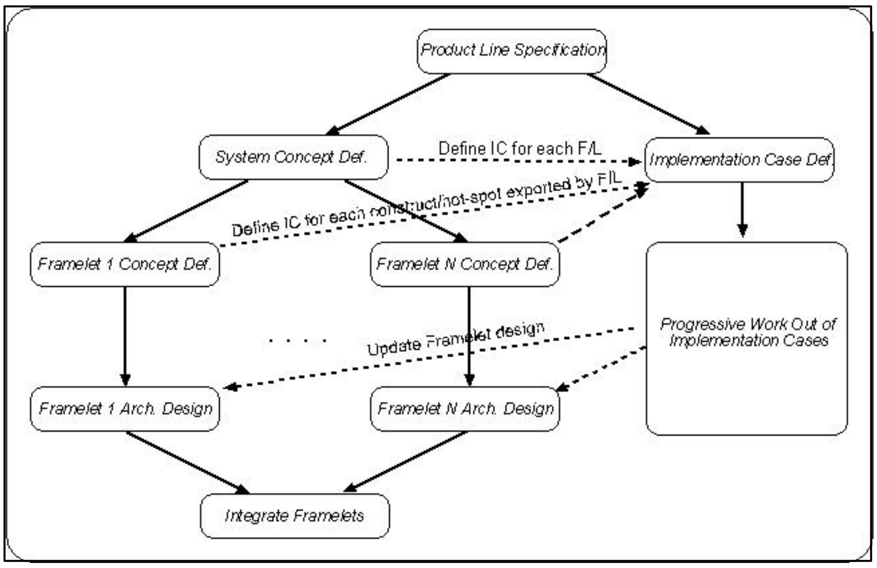

Figure 5. AOCS Product Line Design Process

In the right branch of the design process tree, implementation cases were defined and then gradually worked out in parallel to the definition of the framelet architecture. There was a constant interaction between the two branches. On the one hand, the insights gained while working out the implementation cases prompted changes in the framelet design while on the other hand the introduction of new architectural constructs in the framelets resulted in the definition of new implementation cases to cover them (see section 6.2).

More work is required to ascertain whether this design process can constitute a general product line methodology applicable in other contexts. However, since the primary purpose of a design methodology for product lines must be to manage their intrinsic development complexity, it seems clear that the framelet and implementation case concepts, which address both dimensions of product line complexity, should play a prominent role in it.

8. RELATED WORK

Framelets were introduced in [3,22]. The work reported here goes beyond these references in that it refines the framelet concept in the light of experience from a real product line.

The relationship of implementation cases to use cases has already been mentioned. Additionally, implementation cases have an antecedent in the cookbook recipes of [11,21]. The latter, however, were proposed as ways of documenting product lines providing "how to" examples of their use for application developers. Implementation cases are more ambitious. They can certainly serve as cookbook recipes but their primary value is as tools for the continuous verification of product line design. Furthermore, like the use cases after which they were named, they could be used to specify a product line.

Implementation cases are also related to the Software Architecture Analysis Method (SAAM) scenarios [15]. SAAM scenarios can act as tools to measure the adaptability of an application to future changes. A SAAM scenario describes a hypothetical change in the application specification and considers the ease with which the application design and implementation can be modified to meet the new specifications. An implementation case is similar in that it describes a scenario for adapting an architecture to a particular set of requirements. The difference is that SAAM scenarios are targeted at individual applications which are not specifically designed to be adapted whereas implementation cases are targeted at product lines that exist precisely to be adapted. Both SAAM scenarios and implementation cases resemble use cases in modeling an interaction with a software system but the latter focus on runtime interactions (the use of an application) whereas the former focus on static interaction between the software designer and the software architecture.

9. CONCLUSIONS AND FUTURE WORK

This paper presented the framelet and implementation case concepts and discussed their application to the AOCS project. The architectural design of the AOCS product line was completed in about nine months. Basing the design on framelets and monitoring it with implementation cases were the keys to achieving such a rapid turnaround time probably avoiding at least one design iteration cycle.

Framelets and implementation cases are regarded as mature concepts. Further work will probably have to concentrate on developing formalisms to express them and tools to support product line design based on their use.

Methodological issues are a second area requiring attention. Section 7 outlined the development process adopted in the AOCS project but this falls far short of a comprehensive methodology. Defining such methodology is an urgent task if framelets and implementation cases are to yield the same benefits to other product line projects as they did to the AOCS project.

REFERENCES

[1] M. Fayad, D. Schmidt, R. Johnson, *Application Frameworks,* p. 3-28, M. Fayad, D. Schmidt, R. Johnson, *Building Application Frameworks*, Wiley Computing Publishing, 1999

[2] G. Pomberger, W. Pree, *Quantitative and Qualitative Aspects of Object-Oriented Software Development,* International Symposium on Object-Oriented Methodologies and Systems (ISOOMS '94, Springer-Verlag), Palermo, 21-23 September 1994

[3] W. Pree, K. Koskimies, *Framelets – Small is Beautiful,* p. 411-414, M. Fayad, D. Schmidt, R. Johnson, *Building Application Frameworks,* Wiley Computing Publishing, 1999

[4] M. Mattsson, J. Bosch, *Composition Problems, Causes, and Solutions,* p.467-487, M. Fayad, D. Schmidt, R. Johnson, *Building Application Frameworks*, Wiley Computing Publishing, 1999

[5] M. Awad, J. Kuusela, J. Ziegler, *Object Oriented Technology for Real Time Systems,* Prentice Hall, 1996

[6] W. Pree, *Meta Patterns – A Means of Capturing the Essential of Reusable Object Oriented Design,* Proceedings of the 8th European Conference on Object-Oriented Programming, Bologna, Italy, July 1994

[7] G. Miller *et al., Capturing Framework Requirements,* p. 309-324, M. Fayad, D. Schmidt, R. Johnson, *Building Application Frameworks*, Wiley Computing Publishing, 1999

[8] J. Bosch *et al., Framework Problems and Experiences*, p. 55-82, M. Fayad, D. Schmidt, R. Johnson, *Building Application Frameworks*, Wiley Computing Publishing, 1999

[9] E. Gamma *et al., Design Patterns – Elements of Reusable Object Oriented Software,* Reading, Massachusetts: Addison-Wesley, 1995

[10] M. Fontoura, *A Systematic Approach to Framework Development,* PhD Thesis, Computer Science Department, Pontificial Catholic University of Rio de Janeiro, 5 July 1999

[11] G. Krasner, S. Pope, *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80,* Journal of Object-Oriented Programming, 1(3), 1988

[12] J. Jacobson *et al, Object-Oriented Software Engineering – A Use Case Driven Approach,* Reading MA, Addison-Wesley, 1992

[13] W. Pree, *Hot-Spot Driven Development*, p. 379-394, M. Fayad, D. Schmidt, R. Johnson, *Building Application Frameworks*, Wiley Computing Publishing, 1999

[14] G. Froelich *et al., Reusing Hooks,* p.219-236, M. Fayad, D. Schmidt, R. Johnson, *Building Application Frameworks*, Wiley Computing Publishing, 1999

[15] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice,* Addison Wesley Longman, 1998

[16] R. Johnson, *Frameworks=(Components+Patterns),* Communications of the ACM, Vol. 40, N. 10, p. 39-42, Oct. 1997

[17] W Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995

[18] A. Weinand, E. Gamma, R. Marty, *ET++ - An Object-Oriented Application Framework in C++*, OOPSLA'88, Special Issue of SIGPLAN Notices, Vol. 23, No. 11, 1988
[19] Taligent, Inc., *The Power of Frameworks*, Reading, Massachusetts: Addison-Wesley, 1995
[20] K. Pirklbauer, R. Plösch, R. Weinreich, *Object-Oriented Process Control Software*, Journal of Object-Oriented Programming, April 1994
[21] A. Goldberg, *Smalltalk-80 / The Interactive Programming Environment*, Addison-Wesley, 1984
[22] W. Pree, K. Koskimies, *Rearchitecturing Legacy Systems—Concepts & Case Study,* WICSA '99: First Working IFIP Conference on Software Architecture, San Antonio, Texas, 22-24 Feb. 1999