A Method for Reducing Arbitrary Complexity in Reusable Embedded Systems Code - The Frame Technology Idiom

Thomas Patzke
Fraunhofer Institute Experimental Software Engineering
Product Line Architectures
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany

Abstract

Many software development efforts aim at building more reusable software, for example through product line engineering approaches such as PuLSE[1] [4]. In the embedded systems application domain, the dominant programming languages C and C++ offer a rich spectrum of mechanisms for handling variability. However, most organizations have adopted ad-hoc mechanisms [11], resulting in overly complex code that cannot be reused in the long term. For example, a monoculture of conditional compilation forces all modules to consist of parts with different change rates, while mechanisms that depend on programming language semantics disallow extension at arbitrary variation points.

Thus, ad-hoc implementations lead to unnecessary variability management complexity, which increases the effort for understanding, using and evolving the code as a whole. Our method aims at reducing variability management complexity by guiding the developer through the variability implementation process, so that the code contains just enough complexity as required in the given context.

This paper discusses one sub-element of the method's support components: the frame technology mechanism, which generalizes the reuse concepts of most other mechanisms, such as cloning, polymorphism, conditional compilation, or aspect-orientation.

1. Overview

A problem in many software development efforts is to develop reusable artifacts, and to keep them reusable over long periods of time. Common code alone is not sufficient to make software implementations usable in similar contexts, and hence truly reusable implementations contain common code with some intrinsic variation.

[1] PuLSE is a registered trademark of Fraunhofer IESE

As the code evolves in practice, the same set of mechanisms is often applied again and again. A monoculture of mechanisms emerges, which makes it harder and harder to manage new or to reconfigure existing variabilities. For example, conditional compilation leads to modules of common and nested interdependent variable parts, whose dependencies are hard to control in the long run [10].

In order to avoid these difficulties, we developed an incremental product line implementation method which aims at code complexity reduction by balanced variability mechanism selection. Initially, an implementation of a single system or a set of systems exists, plus new unrealized reuse requirements. These two elements serve as inputs for our method, whose output is a new product line implementation that is just as complex in terms of variability management as required. The method uses three support components to assist the software developer in building the new product line code: A pattern language of variability mechanisms, a collection of product line evolution scenarios, and a complexity metrics suite.

The remainder of the paper is concerned with the pattern language, and presents frame technology as one particular idiom. For reasons of space, the details of the other idioms, the evolution scenarios and the metrics are not discussed. The evolution scenarios represent typical issues a software developer faces when he evolves reusable code. They serve to identify a particular evolution task in a given situation, and they suggest minimal sequences for realizing the variabilities. The metrics suite is used to assess whether the software's variability management quality becomes critical.

2. Related Work

Our approach combines and extends ideas from the following areas: variability mechanisms, practical reuse methods, continuous evolution, and simplicity/complexity.

A number of variability mechanism collections exist

[2, 17, 7, 9]. In contrast to these, we do not just catalog what works theoretically and in unspecified application areas, but focus on practical mechanisms in embedded systems development.

Practical reuse methods [11, 6, 3] often suggest only transformations from design to implementation. We consider a wider context, for example, existing code, mechanisms and tools, or developer experience. We also apply reuse optimization concepts, as offered by frame technology [3], to existing variability mechanisms.

Approaches for managing the long-term evolution of software and non-software artifacts have been suggested in [12, 1, 8]. [12] suggests software evolution laws. As in [1], our method is iterative, based on sequences, and never regards the artifacts as finished. In contrast to refactorings [8], we address a larger group of transformations that are not just concerned with preserving the code's execution logic at runtime. We focus on varying any kind of software element that is fixed at execution time, such as data structures or execution logic.

The importance of complexity-awareness has also been discussed elsewhere [1, 5, 18, 16]. Our approach explicitly focuses on maintaining simplicity by 'just enough' variability management.

3. Variability Mechanism Pattern Language

As a major building block for guiding the developer in applying appropriate variability mechanisms, we developed a pattern language of C/C++ variability mechanisms. The variability mechanism idioms do not stand in isolation, they are meant to be used together, as an interconnected whole, a pattern language. As part of each idiom description, the forces it solves is discussed, which characterizes its reusability profile and (unnecessary) complexities.

Within the larger method, the variability mechanism idioms are meant to be used by the software developer to evolve an existing software implementation into a new product line implementation. In order to help the developer to rapidly access relevant subtopics, each idiom is organized akin to the GoF format. For reasons of space, we only present excerpts from [14].

The following list gives an overwiew of the idioms and their intents. Each idiom was chosen for inclusion in the pattern language because it is either in active use for managing variability in embedded systems development, as demonstrated in its 'Known Uses' subsection, or because there are indicators that the mechanism is becoming important for variability management in C/C++. The set is not meant to be fixed, but open for extension and contraction.

Cloning: Given a source code part which has proven its usability in existing software systems, adapt it to suit the changing needs of a new system. Cloning allows you to rapidly evolve common code without risking to harm its previous users.

Conditional Execution: Separate common from variable algorithms by extracting the variable algorithms into functions, which are conditionally invoked by the common code, depending on runtime parameter states. Conditional Execution allows you to manage predicted optional or alternative variable code, without introducing separate modules.

Polymorphism: Avoid coupling modules with common functionality to modules with variable functionality by letting the common modules define and invoke callbacks which the variable modules implement. Subtype Polymorphism allows parts of an algorithm to change at runtime.

Late Module Binding: Decouple common from variable source code modules by only partially implementing the common parts, deferring the completion of the missing realizations to a later stage in the compilation process. Late Module Binding allows common and variable modules to evolve independently, without runtime efficiency penalties in the resulting executable.

Conditional Compilation: Decouple common from variable source code, so that the variable code is highlighted, and can be automatically included or excluded from compilation. Conditional Compilation allows you to manage optional or alternative variable code next to common code, without introducing new modules.

Aspect-Orientation: Facilitate decoupling common from variable modules by allowing the common source code in the common module to be extended or contracted by variable code at its function boundaries. Aspect-Orientation allows arbitrary common functions to be extended or overridden in predicted or unpredicted ways.

Frame Technology: Decompose textual information according to its stability over time, so that modules which need to change less frequently become nearly independent of modules that evolve more often. Frame Technology facilitates to keep source code localized which shares the same change rate, especially in cases where otherwise the programming language syntax would enforce this code to crosscut several modules.

4. Frame Technology

This section presents excerpts from the frame technology idiom, a variability mechanism developed by Bassett [3]. In recent years, the author of this paper has been developing various kinds of frame technology tool support [15] and applied its concepts in numerous embedded systems projects, for example in the automotive and consumer electronics application domain.

4.1 Motivation

A transceiver module of a wireless sensor node requires an optional acknowledgement feature in more reliable prod-

ucts. This is its pseudocode, realized with Conditional Compilation:

```
/* transceiver.h */
#ifdef HAS_ACKNOWLEDGE
extern bool acknowledge;
#endif
void send(char*);
```

```
/* transceiver.c */
#ifdef HAS_ACKNOWLEDGE
bool acknowledge=true;
#endif

void send(char* msg)
  initialize_transceiver();
#ifdef HAS_ACKNOWLEDGE
  acknowledged=false;
#endif
  for n iterations
    send(msg)
#ifdef HAS_ACKNOWLEDGE
    if acknowledge_received()
      acknowledged=true;
      return;
#endif
```

Consider that new products are developed which require different acknowledgement realizations, for example to improve reliability when successive messages are acknowledged. Using Conditional Compilation, each of these would require the transceiver source code modules to be modified, possibly corrupting common code which is used by existing products. For that reason, it is desirable to extract the common, unchanged code and the variable code which evolves more often into separate modules. The given implementation makes it hard to efficiently realize such a separation, because the variable code is distributed across two modules, the header file and the implementation file, and in the latter, it appears as part of an iteration.

Using Frame Technology, the common and variable code parts are decomposed into separate modules, as shown in the following pseudocode:

```
/* transceiver.frame */
OUTFILE transceiver.h
void send(char*);
OUTFILE transceiver.c
void send(char* msg)
  initialize_transceiver()
  for n iterations
    send(msg)
```

```
/* acknowledgement.frame */
ADAPT transceiver.frame
INSERT_BEFORE 'void send(char*)'
extern bool acknowledge=true;
INSERT_BEFORE 'void send(char* msg)'
bool acknowledged=false;
INSERT_AFTER 'initialize_transceiver()'
acknowledged=false;
INSERT_BEFORE 'send(msg)'
    if acknowledge_received()
      acknowledged=true;
      return;
```

An advanced preprocessor, uses either the transceiver.frame or the acknowledgement.frame file as input and produces the corresponding .c and .h files, which can then be used as C compiler input.

4.2. Applicability

Frame Technology should be used

- if certain source code parts within or across modules always change together, but cannot be easily extracted into a single module using C/C++ programming language mechanisms
- to manage alternative variabilities independently
- when source code and other textual product line artifacts should evolve together
- when there should be a global point of modification and configuration of variabilities
- to both highlight variabilities and hide commonalities
- when it makes sense to distinguish between different levels of context-sensitivity in variable source code

4.3 Consequences

Frame Technology is an advanced variability mechanism for source code in arbitrary programming languages, and other textual artifacts. It has the following strengths and weaknesses:

+ Arbitrary text parts can be managed as variabilities, even syntactically incomplete code such as partial loops or isolated return statements. The code parts can have arbitrary granularity, from single variables or functions to entire subsystems. Code parts from several programming languages can be managed together, as well as non-code artifacts.
+ Variability management at explicit, different levels of context-sensitivity is facilitated, so that the modules become nearly decomposable [16]; they are organized in reuse hierarchies according to their stability over time.
+ There are no effects on resource efficiency.
+ The variable parts are highlighted, and at the same time the common parts are hidden. The common parts can be modularized completely independent from the variable ones.
+ Commonalities and *defaults* are used as first-class elements, instead of variabilities. This makes negative variabilities [6] (same as-except [3], contraction [13]) as easy to express and use as positive variabilities (extension). Each default also reduces the number of separate variable elements by one.

+ Unpredicted changes are supported, because variation points are open parametersx of variation which can be overridden in arbitrary ways. This is in contrast to closed parameters, such as traditional preprocessor macros, which only allow selecting from a fixed number of predefined options.

+ An exponential growth in the number of modules in an evolving system is avoided, because adding new alternatives in alternative variabilities or multiple coexisting possibilities only leads to a linear growth in modules.

- Additional tool support and training is required.

- The entire source code of the reused components must be available; black-box reuse is not supported.

4.4 Implementation

These issues should be considered when using Frame Technology as a product line implementation technology [14, 15, 3]:

- **Evolution.** Frame hierarchies explicitly separate modules with different evolution rates. If a frame processor also provides closed parameters, they can be used to mark several default versions or deprecated code parts. Existing modules can easily be made adaptable without modifying their source code by adding frame annotations (the source code must not contain annotation text). Adapting these modules facilitates parallel evolution without inconsistent co-evolution of commonalities. If it turns out during evolution that certain frame parts have higher or lower change frequencies than initially conceived, this can be easily refactored by moving the part up or down the frame hierarchy.

- **Organization of Parts.** The following rules help to decide where to place a code part within a frame hierarchy: When two code parts have the same evolution frequency, they should be placed in the same level of the frame hierarchy. In addition, when these code parts are always reused together, they should be placed in the same frame. When the parts can be independently reused, e.g. as alternative variabilities, they belong into sibling frames. And when they do not have the same change rate, i.e., one part implies a reuse of the other, but the other can be reused alone, then they belong into different levels.

- **Selection of Defaults.** Frame Technology makes extensive use of default text. When a common part is framed, and variation points are defined, a meaningful default code should also be provided in many cases, rather than leaving the default code empty. This has the advantage that the template code provided by the frame becomes more comprehensible when it is adapted, because the default serves as a best example. It can also make the ancestor frames smaller, which is always a desirable goal. Another observation concerning defaults is that the set union of all defaults in a frame does not always need to result in a meaningful product instance; the frame may be reused more efficiently if each default alone does not require extensive overriding.

5. Conclusion

This report gave an overview of a method for reducing complexity in reusable code, presenting frame technology as a mechanism which generalizes reuse concepts of most other mechanisms. Follow-up papers will present method, scenario and metrics details.

References

[1] C. Alexander. *The Nature of Order, Book 2: The Process of Creating Life*. Center for Environmental Structure, 2002.

[2] V. Alves. *Implementing Software Product Line Adoption Strategies*. PhD Thesis, Federal Univ. of Pernambuco, 2007.

[3] P. G. Basset. *Framing Software Reuse: Lessons From The Real World*. Yourdon Press, 1997.

[4] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. Pulse: A methodology to develop software product lines. *Proc. SSR 5: 122-131*, 1999.

[5] F. P. Brooks. *The Mythical Man-Month (20th Anniversary Edition)*. Addison-Wesley, 1995.

[6] J. O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1998.

[7] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[9] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse. Architecture, Process and Organization for Business Success*. ACM Press, 1997.

[10] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. Refactoring a legacy component for reuse in a software product line: a case study. *Journal of Software Maintenance and Evolution 18(2): 109-132*, 2006.

[11] C. Krueger. The 3-tiered methodology: Pragmatic insights from new generation software product lines. *Proc. SPLC 11: 97-106*, 2007.

[12] N. Madhavji, J. Fernandez-Ramil, and D. Perry. *Software Evolution and Feedback. Theory and Practice*. Wiley & Sons, 2006.

[13] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering, 5(2): 128-138*, 1979.

[14] T. Patzke. Mechanisms, processes and metrics for implementing and evolving reusable embedded systems. *IESE Report 021.08/E*, 2008.

[15] T. Patzke and D. Muthig. Product line implementation with frame technology: A case study. *IESE Report 018.03/E*, 2003.

[16] H. A. Simon. The architecture of complexity. *In Simon, H. A., The Sciences of the Artificial (2nd ed.), MIT Press*, 1994.

[17] M. Svahnberg and J. Bosch. A taxonomy of variability realization techniques. *Softw., Pract. Exper. 35(8): 705-754*, 2005.

[18] N. Wirth. A plea for lean software. *IEEE Computer 28(2): 64-68*, 1995.