Variability Mechanisms and Lessons Learned in Practice

Authors:
Bo Zhang
Slawomir Duszynski
Martin Becker

IESE-Report No. 006.16/E
Version 1.0
May 2016

A publication by Fraunhofer IESE
Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft. The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competitive market position.

Fraunhofer IESE is directed by
Prof. Dr.-Ing. Peter Liggesmeyer
(Executive Director)
Prof. Dr. Dieter Rombach
(Director Business Development)
Fraunhofer-Platz 1
67663 Kaiserslautern
Germany
Variability Mechanisms and Lessons Learned in Practice

Bo Zhang*, Slawomir Duszynski**, and Martin Becker*

* Fraunhofer Institute Experimental Software Engineering (IESE); Kaiserslautern, Germany
** Robert Bosch GmbH, Corporate Research; Stuttgart, Germany

Bo.Zhang, Martin.Becker@iese.fraunhofer.de              Slawomir.Duszynski@de.bosch.com

ABSTRACT
In the design of complex and variable software systems, one of the key steps is to select the variability mechanism that defines how variable features are realized on the design and code level. Although different variability mechanisms were invented and applied in practice for decades, there are not many studies that compare these mechanisms based on practical experiences. This paper characterizes and compares seven variability mechanisms in terms of their techniques, binding time, granularity, and further aspects. It provides experiences of their usage, the practical benefits and challenges, as well as discusses existing solutions to the challenges based on related studies and our practice in industry.

CCS Concepts
• Software and its engineering→Software creation and management →Software development techniques→Reusability.

Keywords
Variability Mechanisms; Variability Design; Practical Experience.

1. INTRODUCTION
In software development and maintenance, engineers use variation management to accommodate various product requirements (either for a configurable system or for a product family) with high efficiency. However, as the complexity of such a variability-aware system grows over time, variability design and implementation often becomes a practical challenge. When dealing with variability design and system complexity, one of the key steps is to select the variability implementation mechanism (referred to as variability mechanism in this paper), or a combination of mechanisms, that is suitable for the given system.

The notion of variability mechanisms was addressed in [18], [31], [41] and [4] about 10 years ago by researchers in the software product line community. These mechanisms describe how variable features can be realized on the design and code level using different programming techniques. However, these mechanisms and corresponding techniques have been applied in practice even many years earlier [12][33]. In this paper, we considered the mechanism categorization in research papers and our practical experiences, and we abstracted from the actual technical realizations (such as e.g. build systems or configuration management systems) by distilling the underlying basic, technology agnostic idea of a mechanism. Consequently, we discuss seven variability mechanisms: Cloning, Conditional Compilation, Conditional Execution, Polymorphism, Module Replacement, Aspect Orientation, and Frame Technology.

The listed variability mechanisms differ in various aspects such as techniques, binding time, granularity, and further. For this reason, these mechanisms are applied in different development scenarios in practice. In the practical (especially industrial) case studies, each mechanism showed benefits but also brought challenges during system development. As each variability mechanism has its own applicability, these experience can motivate researchers and guide practitioners in developing new variability realizations as well as assessing and maintaining existing variability realizations.

Based on the studies of variability mechanisms and the practical experience in industry, in this paper we provide the following contributions:
1. Categorization and characterization of different variability implementation mechanisms.
2. Practical usage, benefits and challenges in adopting these variability mechanisms which are determined by their characteristics.
3. Successful solutions for solving these challenges using state-of-the-art techniques and tools in industry.

This paper is presented in the following structure. Section 2 shortly introduces the variability mechanisms and their characteristics. Section 3 discusses the practical benefits and drawbacks of the mechanisms and presents the cases of their successful use. Section 4 discusses the related work. Finally, Section 5 presents conclusions and future work.

2. CHARACTERIZATION OF THE VARIABILITY MECHANISMS
This section presents characterization of the seven variability mechanisms in various aspects such as techniques, binding time, granularity, etc., which is summarized in Table 1 and discussed as below. These objective characteristics indicate the applicability of each mechanism in practice.

Techniques. To link the abstracted variability mechanisms to their actual realizations, we list the respective development or programming techniques. While some mechanisms use general techniques which are supported by almost all programming languages (i.e., Cloning, Conditional Execution, Module Replacement), other mechanisms can only be applied in specific programming languages (Conditional Compilation, Polymorphism) or even require dedicated environment or tool support (Aspect Orientation, Frame Technology).
By using a variability mechanism, variable features are realized in the source code. In the software life-cycle, these variabilities are instantiated and bound to a variant at a certain point of time: at the construction time (preprocessing and compilation), or at the run-time. Mechanisms with early binding time resolve the variability configuration space early and potentially optimize running efficiency, while mechanisms with late binding time provide more flexibility and support dynamic system adaptations.

Granularity. Depending on the used variability mechanism, the granularity of variants differs. Some mechanisms (i.e., Cloning, Conditional Compilation, and Frame Technology) are essentially text-based and thus support any granularity of code variants in a source file. In contrast, the other mechanisms enforce a certain size or form of the variants within the code structure (e.g., a function, a class, or a file). While mechanisms supporting any granularity range are more flexible in variability realization, code variants using mechanisms with a limited granularity range are usually more disciplined, well-structured, and easier to maintain.

Explicit variation points. Depending on the used variability mechanism and its technique, the form of variation points differs. In the mechanisms of Conditional Compilation and Frame Technology, variation points are both realized as annotated code fragments (e.g., a #ifdef block), which are explicit and easy to identify. In contrast, the variation points of the other mechanisms are implicit. For instance, variation points of Cloning are arbitrary text, and variation points of Module Replacement are typically function calls or files which also can be non-variability code. Since variability is a cross-cutting concern with (often highly) scattered realization code, mechanisms with implicit variation points tend to cause challenges in development and maintenance.

Variant isolation. Depending on the used variability mechanism, code variants for each variation point are either written in one source file or written in isolated modules or files. Based on this difference, Kästner [21] grouped these mechanisms into two categories: annotative mechanisms (i.e., Condition Compilation, Conditional Execution) and compositional mechanisms (the remaining ones). The variability code using annotative mechanisms is more integrated but potentially more complex, while the variability code using compositional mechanisms is less complex but more fragmented.

Open variation. A related concern is whether a mechanism supports open, or only closed variation. Essentially, closed variation requires that the variable code is compiled together with the core code. Hence, the core is treated as a “white box”, and no new variants can be added independently (e.g., by external developers or the users). In contrast to that, the mechanisms which also allow open variability can enable the external developers to provide modules (plugins, etc.) extending the system with their own code after its compilation, while treating the core as a “black box” with defined variation points.

Non-code artifacts. Modern software systems include many artifact types besides code, e.g., models, data files, or text files. It might be required to make them variable as well – however, some of the presented mechanisms are applicable to code only.

Support of defaults. In variability realization, Patzke [33] argues that the default selection in a variation point can reduce the number of variants (by one) and simplify the variation logic. It is especially convenient when features of a variation point are optional (in this case the default is null and requires no code at all).

3. PRACTICAL EXPERIENCES WITH VARIABILITY MECHANISMS

This section introduces the usage of variability mechanisms in practice (especially in industry) as well as their practical benefits, challenges and corresponding solutions. Given their applicability in terms of granularity, binding time, etc., some mechanisms are more frequently used in practice than the others. Moreover, domain-specific usage differences exist: embedded software tends to use mechanisms with little performance footprint, such as Conditional Compilation and Module Replacement, while non-embedded domains, having less performance limitations, tend to more frequently use the run-time flexible mechanisms of Polymorphism and Conditional Execution. Finally, multiple variability mechanisms are used in combination in the development of large systems [25].

### 3.1 Cloning

#### Usage in practice

Cloning, i.e. copying a code or non-code artifact and evolving it further without maintaining a connection to the original, is practiced for single system development as well as in the variability context. The size of cloned artefacts can range from small code snippets, through meaningful functionality pieces such as classes and components, up to complete software systems including several asset types [15]. Technically, the clone is created by either physically copying the original artefacts to a new location, or by using configuration management branches. Cloning of artifacts and complete systems is also known as forking.

#### Practical benefits

i) Low effort. Cloning allows for a fast, low-effort creation of a new implementation variant, while not requiring any special tooling or knowledge. Because of that, Cloning can be selected as a variability mechanism when the quick delivery of a new variant functionality is more important than later code maintainability [15]. ii) Independence. Cloning the code induces no risk to the already existing variants - in contrast to that, with most other mechanisms bugs can be introduced to the old variants if e.g. the core code needs to be modified because of the new variant introduction. Furthermore, the changes introduced to the cloned code do not need to be coordinated with other variants, which again reduces the coding effort and allows for introducing changes which, without cloning, would be incompatible with the core code.

### Table 1. Characterization of Variability Mechanisms

<table>
<thead>
<tr>
<th>Variability Mechanism</th>
<th>Techniques</th>
<th>Binding Time</th>
<th>Granularity</th>
<th>Explicit Variation Points</th>
<th>Variant Isolation</th>
<th>Open Variation</th>
<th>Non-Code Artifacts</th>
<th>Defaults</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloning</td>
<td>Copy, branching</td>
<td>Construction time</td>
<td>Any</td>
<td>Implicit</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Conditional Compilation</td>
<td>Preprocessor</td>
<td>Construction time</td>
<td>Any</td>
<td>Explicit</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Conditional Execution</td>
<td>Cond. statements</td>
<td>Run-time</td>
<td>Limited</td>
<td>Implicit</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>Function pointers, overloading, etc.</td>
<td>Mostly run-time</td>
<td>Limited</td>
<td>Implicit</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Module Replacement</td>
<td>Build system</td>
<td>Mostly constr. time</td>
<td>Limited</td>
<td>Implicit</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Aspect Orientation</td>
<td>Code weaving</td>
<td>Mostly constr. time</td>
<td>Limited</td>
<td>Implicit</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame Technology</td>
<td>Frame adaptation</td>
<td>Construction time</td>
<td>Any</td>
<td>Explicit</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

#### Binding time.

In variability realization, Patzke [33] argues that the default selection in a variation point can reduce the number of variants (by one) and simplify the variation logic. It is especially convenient when features of a variation point are optional (in this case the default is null and requires no code at all).
Cloning of classes and components can be also deliberately used as a variability mechanism if their later similarity is expected to be low [42]. In this case, the rationale for using Cloning is the high effort and low benefit of using other variability mechanisms such as Conditional Compilation for the dissimilar code. Deliberate cloning is therefore performed if there is no prospect of a viable variant consolidation in the future.

Finally, Cloning can be used for prototyping and evaluating experimental functionality [24]. The evaluation can be either performed internally, or externally by deploying the new functionality for the customers. If the evaluation fails, the prototyped code can be easily discarded (due to its independence from other variants), and it only used up low development effort. In the success case, the cloned code can start experiencing Cloning challenges and should therefore be refactored to use other variability mechanism.

**Practical challenges.** Apart from the low predicted similarity case, the reasons for Cloning are mostly valid in the short-term perspective. Consequently, the Cloning disadvantages manifest themselves in the longer term, as:

i) Lower maintainability of the resulting code. Maintaining the duplicated code involves repetitive tasks, e.g. when a common change needs to be applied to every variant. That increased effort is further amplified by the need to separately analyze the change context in each clone, as every two clones might differ in subtle ways due to past customizations [37]. Similar problems also occur for new non-common functionality which should be carried over from one variant to another (so-called porting).

ii) Lower code quality. The knowledge of similarities between the clones tends to be quickly lost – in result, the developers might forget to port a bug fix or a functionality upgrade to some relevant clones [15]. In a longer time, such inconsistencies accumulate and lead to a lower quality of the variable code.

**Solutions to the challenges.** In specific cases, Cloning might be used as a well-justified variability mechanism. However, clones should not be left unmanaged. It is beneficial to track the cloned code, for example using a clone management tool [11], to keep track of all the locations relevant for a code change. Prototype branch functionality, as well as the clones introduced because of the time-to-market and short-term effort pressures, should be as soon as possible refactored to use other more maintainable variability mechanism such as Conditional Compilation.

Finally even the deliberate clones, introduced because of a low similarity prediction, should be periodically analyzed to verify whether cloning still remains the best variability mechanism. In the case of a high similarity, changing the mechanism selection by means of refactoring might pay off. This is exemplified by a case study of using the Variant Analysis tool on the cloned components of a large product line at Robert Bosch GmbH [14].

### 3.2 Conditional Compilation

**Usage in practice.** Being used for decades, Conditional Compilation has proven to be one of the most frequently used variability mechanisms. For instance, many industrial product lines (especially in the embedded systems domain) have used Conditional Compilation [33][34], and many large open-source systems (including the Linux kernel) have also used Conditional Compilation for variability realization [26][17]. Although the preprocessor technique theoretically can be used in many languages (including Java, C#, Fortran, Visual Basic, D, etc.) [21], in practice it is mainly used in C/C++ languages, where code variants are conditionally included or excluded by `#ifdef` statements. (For brevity, in this paper we use `#ifdef` to also subsume the related directives of `#if`, `#elif`, `#ifdef`, `#ifndef`, etc.). However, preprocessors for non-code artifacts also exist [35].

**Practical benefits.** Conditional Compilation has been frequently used for variability realization due to its practical benefits: i) Easy to introduce. In particular, the C-Preprocessor technique is well-known and has been used for decades in practice. ii) Not limited to variant granularity. The enclosed code fragments in `#ifdef` blocks can be either classes, functions, type declarations, or variables because the code preprocessing is independent of the normal code syntax (this also brings disadvantages as discussed below). iii) Explicit variation points identified by `#ifdef` blocks, especially when the variability names are defined following a certain naming convention as in [45]. iv) No efficiency penalties because the variations are instantiated during code preprocessing and do not exist at run-time anymore.

**Practical challenges.** While adopting Conditional Compilation in variability realization and management, practitioners often encountered challenges due to code complexity.

i) In real large systems, preprocessor-based variability code is usually complex and difficult to understand. On the one hand, `#ifdef` blocks are sometimes nested with a complex structure. This is because a single `#ifdef` block (considered as a variation point) may contain two conditional branches and therefore supports at most two code variants. If a variation point needs to contain more than two code variants, then multiple `#ifdef` blocks need to be written in a nested structure. On the other hand, the logical expression in `#ifdef` statements can be complex and involves multiple macro constants (i.e., variability names). Liebig et al. [26] analyzed `#ifdef` code in 40 open-source product lines, which measured and identified code complexity in terms of `#ifdef` nesting, tangling, scattering, etc. According to the empirical study of Hunsen et al. [20], these issues exist in not only open-source systems but also many industrial systems.

ii) The realizations of commonality and variability (i.e., `#ifdef` code) are integrated and interleaved in the same code file, which makes it often extremely large and complex [34]. Kästner [21] claimed that `#ifdef` annotations obfuscate the normal source code layout, which makes the combined core code difficult to understand. Moreover, Liebig et al. [27] investigated the specific issue of undisciplined `#ifdef` code, where part of a code structure (e.g., a function definition or a statement) is enclosed in an `#ifdef` block as variability code while the other part is outside the `#ifdef` block as common code. This also makes the core code difficult to read and understand, which may lead to syntactic, type, and semantic errors [21].

iii) Besides the development challenges, the variability code complexity also causes challenges in maintenance. In an industrial case study with the Danfoss company [45], we have analyzed 31 releases of C-Preprocessor-based variability code evolved over four years. This study shows that the number of variabilities and variation points have continued to increase over time, and their implementation using Conditional Compilation became overly complex in terms of `#ifdef` nesting, tangling, scattering, etc., which is called variability code erosion. In particular, based on the code analysis we have identified code hotspots which indicate high maintenance effort regarding change propagation and quality assurance. In fact, the maintenance activities could be even more expensive in practice because the variability code and the commonality code are usually interleaved (as mentioned above). Whenever the variability code is changed and tested, the commonality code in the same file (that is usually more stable) needs to be read, understood, and tested anyway.
Solutions to the challenges. Given the challenges of variability code using Conditional Compilation, some solution approaches with tool support have been applied in practice. Inspired by the early study of Liebig [26], Zhang [46] developed a variability reengineering approach called VITAL (Variability Improvement Analysis). The VITAL approach and tool support automatic analysis of C-Preprocessor-based variability code. VITAL has been applied in several industrial projects (such as [45]) for extracting variability code elements, measuring variability-specific code complexity, and identifying code erosion during the evolution history. Based on the VITAL results, developers can control variability code complexity and either fix existing variability code erosion or prevent erosion in the future code. Besides, the TypeChef tool developed by Kästner [21][23] has been used for identifying type errors C-Preprocessor-based code in different systems.

3.3 Conditional Execution

Usage in practice. While variation points using Conditional Compilation are implemented as #ifdef blocks, in Conditional Execution they are usually implemented as if-else blocks. If there are more than two alternatives, then the variation point can be refactored as a switch block. Since Conditional Execution supports run-time binding, it is often used for realizing features that are variable at run-time (e.g., dynamic system adaptation) or are configured after compilation (e.g., using a configuration file). Similar to Conditional Compilation, in practice the variability names are normally defined following a certain naming convention in order to distinguish them from normal variables used in conditional statements.

Practical benefits. An obvious benefit of Conditional Execution is that it is easy to use with almost no learning effort. Moreover, variabilities using this mechanism are instantiated in a very late phase, which provides high flexibility and ease management of unforeseen requirements. For this reason, it is considered by Bosch et al. [8] as a promising trend in software variability, although this introduces complexity and other side effects.

Practical challenges. Despite the benefits above, developers have encountered several challenges: i) Code variants must be behavioral and fine-grained because they are implemented as conditional code blocks. ii) It is hard to distinguish between the variation logic and code functionality because code variants are written together with commonality code in one file. iii) The mechanism enforces inclusion of all variant elements from code compilation till running, which decreases compilation speed and efficiency at run-time.

Solutions to the challenges. From our practical experience, developers tend to avoid Conditional Execution as long as the run-time variability is not required. Especially for embedded or resource-constrained systems, a good alternative mechanism to use is Conditional Compilation.

3.4 Polymorphism

Usage in practice. In practice three types of Polymorphism are in wide use: Subtype Polymorphism (e.g. function pointers, inheritance), Parametric Polymorphism (e.g. C++ templates) and Overloading. While Overloading occurs at construction time, Subtype Polymorphism and Parametric Polymorphism may occur at either construction time or run-time. In particular, Subtype Polymorphism is often used for decoupling different concerns (e.g., interfaces and the underlying application logic) that may have different evolution rates.

Practical benefits. As a compositional mechanism, common and variant elements are written in separated files, modules, or classes, so that they can be maintained and evolved in parallel. Furthermore, Polymorphism provides high flexibility for run-time variability instantiation. Finally, the open variation of Polymorphism is frequently used in software frameworks, as it enables users to provide own code extending framework capabilities without modifying the (already compiled) framework code.

Practical challenges. While adopting Polymorphism in practice, several challenges have been experienced: i) The running efficiency is relatively low when variable features are bound with separated or even fragmented variant elements at run-time. In the case of Subtype Polymorphism, this issue is even more severe than Conditional Execution due to complex code structure. ii) As Polymorphism may use more advanced techniques (e.g., function pointers, overloading) comparing to Module Replacement or Conditional Execution, it increases the risk of software defects, especially run-time errors such as illegal pointers (This is why some industrial embedded systems standards such as MISRA [30] disallow the usage of function pointers.).

Solutions to the challenges. Similar to Conditional Execution, developers tend to avoid Polymorphism as long as the runtime variability is not required (especially in embedded systems development). Considering the separated variant elements, Module Replacement would be a good alternative and can be easily adapted. If runtime variability or post-build extensibility is needed, the use of design patterns can help achieve a well-maintainable and understandable variability implementation [13].

3.5 Module Replacement

Usage in practice. Module Replacement is mainly realized at compile time, but can be also realized at preprocessing time (by specifying an included header file) or link time (by specifying a path to linked libraries). In practice it is often used for selecting among files or larger subsystems. For instance in C, these subsystems are often realized as identically named .c and .h files which are stored in sibling directories. Then by binding a variability with a respective directory, an entire subsystem is selected for compilation.

The technical mechanisms of plugins and services constitute a border case between module replacement and polymorphism and can be assigned to each of them depending on the realization details.

Practical benefits. Module Replacement is a well-known and easy to use mechanism, as long as the variant elements can be modularized (e.g., as a function). Moreover, like in Polymorphism, common and variant elements are written in separated files, which allows them to evolve in isolation. Also, there is no efficiency penalty because variabilities are resolved before the run-time.

Practical challenges. Despite the ease of use and other benefits, a major challenge of Module Replacement is that variation points are not entirely visible in the code because they are represented by normal function calls. This often makes maintenance of variation points and respective variant modules difficult. Besides, this mechanism enforces the selection of one variant and does not support the default realization. In the case of optional variability, even an empty module or function must be provided in isolation.

Solutions to the challenges. Considering the implicit variation points and (typically lots of) separated variant modules, a practical solution is to manage them with tool support. For instance, by using the variation management tool pure::variants [35], variation points and variant modules can be referenced in the so-called family model for code construction and maintenance.
3.6 Aspect Orientation

Usage in practice. Aspect Orientation relies on the code weaving technique, which is normally not a built-in feature of common programming languages but requires external tool support such as AspectJ. In general, Aspect Orientation has been applied in software system development for some years. However, according to the recent review by Rashid et al. [36] its usage is mainly in code tracing, logging, and exception handling. There is still not enough evidence of its successful practical usage in variability realization. Similar technology-centric approaches have repeatedly been proposed as Subject-Oriented [19], Feature-Orientation [7][2], Change-Oriented [16], and Delta-Oriented [39].

Practical benefits. As a compositional mechanism like Polymorphism and Module Replacement, Aspect Orientation has similar benefits in theory such as separation of common and variant elements. Functions of the common elements are potential variation points, at which the system behavior might be extended or replaced. Furthermore, Aspects provide means of consolidating variant behavior which would otherwise be scattered over many code modules and tangled with common code as well as with other variations. This might lead to better code maintainability. With Aspect Orientation, new variants can be added without changing the common elements. Depending on the aspect weaver, switching between compile time and runtime binding is possible. Besides, the mechanism also supports default implementation.

Practical challenges. As Aspect Orientation is normally not supported by a programming language itself, it is difficult to be applied rapidly in development and probably needs more learning effort. It is probably the reason why it has not been widely used in industry. Moreover, Kästner et al. [22] have conducted a case study to refactor variability realizations in the Berkeley database system using AspectJ, which turned out to be unsuitable for implementing variable features. Similarly, a literature review by Amin et al. [1] shows that the major challenges of Aspect Orientation are the increase in code size and low code comprehensibility.

Solutions to the challenges. Considering the reported challenges and no sufficient evidence of successful practical usage, it is advised to carefully consider the benefits and challenges before using this mechanism. If Aspect Orientation is already used in existing variability realizations, and the challenges prevail, variability code refactoring can be considered using alternative mechanisms such as Conditional Compilation or Module Replacement.

3.7 Frame Technology

Usage in practice. Basset [6] introduced the Frame Technology that uses adaptable code frames and assemble them by a frame processor. Several commercial or academic frame processing tools exist (but with different syntax) and have been used in practice. XVCL [44] has been used to eliminate redundancies in parts of the Standard Template Library of the C++ programming language [5]. FP [32] has been used for framing product line assets (code, requirements, architecture) of resource-constrained embedded systems in the automotive and consumer electronics application domain. Moreover, in another case study [33] it also has been used to simultaneously manage variability in C and Java languages. In the same case study, Frame Technology has been shown to provide better product line evolution potential than other mechanisms.

Practical benefits. Frame Technology shows several benefits in practice: i) Like other compositional mechanisms such as Polymorphism, it decouples common and variant elements; ii) Like Conditional Compilation, variation points using Frame Technology are explicitly identified, and variant elements are only handled as textual elements but do not have to form any cohesive procedural element; iii) Variabilities are resolved at construction time which avoids efficiency penalty at run-time.

Practical challenges. Although there are some studies and experiences as mentioned above, this mechanism has not been widely used because it requires special tool support, and the respective technique is relatively unknown to developers, especially in industry. Moreover, a standard syntax for Frame Technology is still missing, and the respective tool support is limited.

Solutions to the challenges. Considering the benefits of Frame Technology, we recommend further research in this direction to provide strong tool support and adopt this mechanism in a long-term perspective. For industrial use, however, a careful consideration of benefits and challenges is recommended.

4. RELATED WORK

In the development of configurable systems or software product lines, Gacek and Anastasopoulos [18] initially presented 11 variability implementation mechanisms. Moreover, they characterized these mechanisms in terms of usage scenarios (i.e., interface, implementation, and initialization), binding time, scalability, and so on. In fact, many of these mechanisms are programming techniques such as dynamic class loading and static libraries. They also mentioned their practical experiences regarding benefits and challenges of these mechanisms. To our best knowledge, this is the first comparative study of variability mechanisms. Later, Muthig and Patzke [31] discussed 6 variability mechanisms suitable in component-based development. However, they did not treat some popular mechanisms such as Conditional Execution or Module Replacement. Bachmann and Elements [4] provided a list of 9 adapted variability mechanisms and described the related stakeholders, skills, tools, and cost. Svahnberg et al. [41] and Apel et al. [3] describe several variability realization techniques. Their categorizations are, however, oriented towards the technical realizations of the mechanisms (e.g., architecture reorganization, build systems), while in this paper we categorize the abstracted mechanisms regardless of their technical implementation. Essentially, these two categorizations are orthogonal and complementary to our paper.

In the recent years, there were studies focusing on variability realizations using a single variability mechanism such as Cloning [37][15] and Conditional Compilation [27][45], but the transversal comparison of these mechanisms, especially focusing on their practical usage, benefits, and challenges, seems to be lacking. Besides, there are other studies of product line realization assessment such as [10] and [28]. However, they are usually generalized and concern the assessment of product line architecture or the variability specification. In contrast to our paper, they are not specific to a certain variability realization mechanism.

Based on the described characteristics of variability mechanisms, developers can assess existing variability realization and take further decisions. In this paper, we have introduced our experiences in analyzing and assessing systems implemented by the mechanisms of Cloning and Conditional Compilation, and our corresponding tools are Variant Analysis [14] and VITAL [46] respectively. In both these areas, there exist several other approaches and analysis tools, such as those developed for Cloning by Yoshimura et al. [43], Mende et al. [29] or Rubin et al. [38], and those developed for Conditional Compilation by Kästner et al. [23] or by Sincero et al. [40]. However, the comparison of these approaches is beyond the scope of this paper.
5. CONCLUSION
As a key topic in variability design and implementation, this paper evaluates seven different variability mechanisms based on their objective characteristics and experiences in practice. On the one hand, we characterized each of these variability mechanisms in terms of granularity, binding time, and other aspects. On the other hand, based on recent studies and our industrial experiences in these mechanisms we discussed their usage, practical benefits and challenges, and existing solutions to corresponding challenges as summarized in Table 2. Considering all these factors, we provide recommendations to determine the most appropriate mechanism(s) in variability design, and also discuss other issues such as maintenance and evolution in later phases.

As for future work, it will be interesting to extend this research to a comprehensive empirical study discussing more aspects and involving more participants from industry. This will help to collect more practical feedback about how variability mechanisms have been applied in practice, which will motivate researchers and guide practitioners in variability design and implementation.

6. ACKNOWLEDGMENTS
We thank Thomas Patzke and our industrial collaborators for their work in research and industrial projects which contributed to this paper.

7. REFERENCES

Table 2. Practical Experiences with Variability Mechanisms

<table>
<thead>
<tr>
<th>Variability Mechanism</th>
<th>Practical Benefits</th>
<th>Practical Challenges</th>
<th>Recommendations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional Compilation</td>
<td>Easy to introduce. No limited granularity. Explicit variation points. No efficiency penalties.</td>
<td>#ifdef code is hard to understand due to #ifdef nesting, tangling, scattering, etc. #ifdef code tends to erode and is hard to maintain during evolution.</td>
<td>Analyze and manage the complexity of #ifdef code against variability erosion. Refactoring in the case of maintenance problems.</td>
</tr>
<tr>
<td>Conditional Execution</td>
<td>Easy to introduce and understand. Supports run-time binding. High flexibility.</td>
<td>Limited to small granularity. Hard to distinguish between the variation logic and code functionality. Efficiency penalties.</td>
<td>Avoid it if runtime binding is not necessary. Consider Conditional Compilation instead.</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>Separation of common and variant elements. High flexibility. Each variant element can evolve in isolation.</td>
<td>Lower efficiency due to fragmentation of variant elements. Increased risk of software defects.</td>
<td>Avoid in embedded context if runtime binding is not necessary. Consider Module Replacement.</td>
</tr>
<tr>
<td>Module Replacement</td>
<td>Benefits of Polymorphism. No efficiency penalty.</td>
<td>Hard to identify variation points and variant elements.</td>
<td>Managing variation points and variant modules with tool support.</td>
</tr>
</tbody>
</table>