Tailoring Infrastructure Software Product Lines by Static Application Analysis

Horst Schirmeier and Olaf Spinczyk
Friedrich-Alexander University of Erlangen-Nuremberg
Department of Computer Science
Martensstr. 1, D-91058 Erlangen, Germany
{Horst.Schirmeier,Olaf.Spinczyk}@cs.fau.de

Abstract

Besides ordinary applications, also infrastructure software such as operating systems or database management systems is being developed as a software product line. With proper tool support these systems can be configured easily by selecting features in a feature model. However, in the future multi-level architectures of layered product lines will be common practice. For humans the feature-based configuration will become increasingly complex, as the number of configurable features will be tremendous. Our goal is to reduce this complexity. The approach is based on the observation that many configuration decisions could be automated by statically analyzing the code of layers on top of an infrastructure product line. Motivated by use cases the paper presents the concepts behind our analysis tool, which is able to automate the configuration in many cases. First results in the context of a feature-oriented version of the Berkeley DB illustrate the potential of this novel approach.

1. Introduction

Tailoring software systems for a specific application scenario or a certain client has many advantages. For example, it allows developers to customize the provided functionality for individual clients or client groups. At the same time resources are being saved, as (for the particular use case) unneeded program code does not waste memory or CPU cycles in the target system. Especially for resource-constrained systems this is of considerable relevance. Our research group has been working in this area of tailorable systems for several years, especially in the domain of embedded operating systems [9] and embedded application product lines [10].

This article first briefly describes "feature-driven product derivation". This is a known approach from the software product line research community where software variability is being described by a so-called *feature model* [7, 5]. A so-called *family model*, which connects abstract software *features* with actual implementation components, allows to configure such a system solely by feature selection. Despite the elegance of this approach there still remain several problems: Feature models can become very big in practice[1], which leads to a tremendous manual configuration effort.

The actual contribution of this article is an approach to partially automate the configuration of infrastructure product lines by applying static analysis of application code. We present several use cases, details about the application analysis, and first results, which we obtained by analyzing applications of a feature-oriented version [2, 3, 14] of the Berkeley DB with our tool prototype.

The remainder of this paper is structured as follows: Chapter 2 gives a more detailed overview of our approach and presents motivating use cases. Chapter 3 describes the application properties that are relevant for the configuration process and our application model. Queries on the application model and the connection to the product line's feature model are the topic of chapter 4. Chapter 5 contains a short description of our tool prototype and first evaluation results. The article concludes with a discussion of related work and a summary in chapters 6 and 7.

2. Motivation and Approach

2.1. Infrastructure Product Lines

Feature-driven product derivation begins with a *feature model*, which consists of features, sub-features, and optional additional constraints such as "conflicts" or "requires" relations. For example, fig. 1 shows the feature diagram of a refactored, feature-oriented version of the Berkeley DB written in the language FeatureC++. During the system configuration a variant is defined by selecting the needed features.

[1] From industrial applications we know software product lines with more than 4000 configurable features.

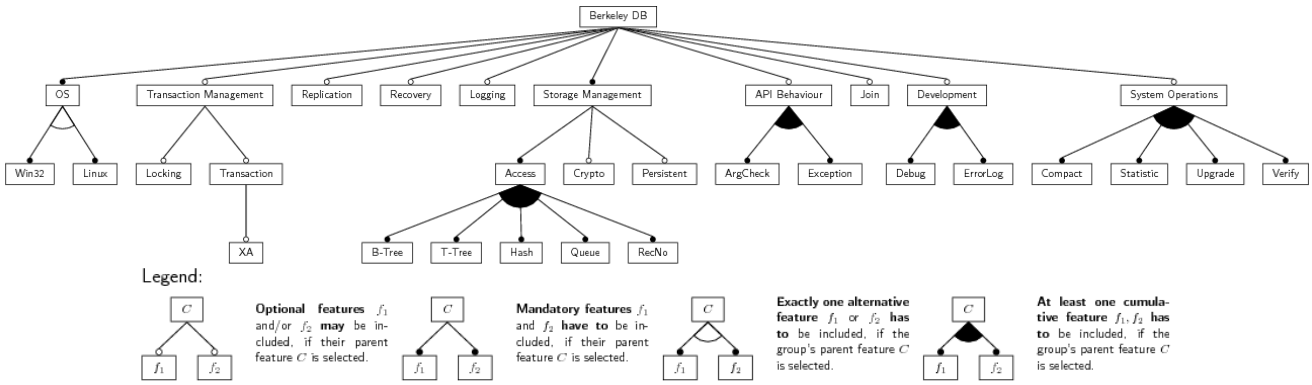

Figure 1. Feature diagram of a refactored, feature-oriented version of the Berkeley DB

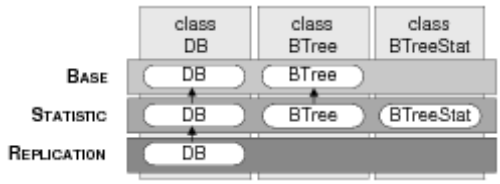

Figure 2. A feature-oriented design [14].

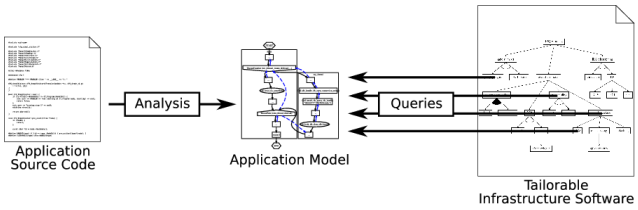

Figure 3. The analysis step produces an application model, which is the basis for queries that help tailoring the infrastructure software.

The implementation consists of so-called *mixin layers*, each implementing a certain feature (Fig. 2). Building upon a base layer, each subsequent layer *extends* the layers below by additional functionality, or it *refines* their functionality. In the configuration process, selecting features from the feature model determines the set of *mixin layers* to be included (and *woven* by the FeatureC++ weaver to traditional C++ source files) in the resulting Berkeley DB *variant*.

With traditional feature-driven product line derivation, the application developer has to make all configuration decisions manually in order to tailor the underlying infrastructure product line for his application's needs. With highly configurable product lines and several thousands of selectable features the configuration time and effort quickly grows into unacceptable dimensions. Besides the direct time/cost factors for the configuration process, a large number of configuration decisions also increases the possibility for human mistakes, leading to a resource suboptimal variant. The development of appropriate tool support is the logical consequence.

2.2. Application Analysis for (partially) Automated Configuration

The inherent *uses* relationship between application and infrastructure suggests to derive the need for infrastructure features from the application itself. Therefore, we create an application model by analyzing the application statically. This model abstracts from syntactic details of the application source code (Fig. 3)[2]. Infrastructure features that are suitable for automatic detection can be associated with queries on the application model. These queries answer the question whether the application needs the feature. Possible answers are ...

"Yes" The application definitely needs this infrastructure feature, it can be preselected in the feature model.

"No" The application definitely has no need for this feature, it can be deselected in the feature model.

"Maybe" The need for this feature cannot be determined with this query, the selection has to be done manually.

Ideally, large parts of the feature tree can be pre-configured. The developer has to manually select only features that cannot be derived from the application sources ("Maybe").

2.3. Use Cases

Prior to the design of the application meta model, a series of *use cases* was defined. These use cases describe certain

[2]It is not possible to abstract from the underlying programming paradigm. The presented model elements can describe C++ programs and programs written in similar languages.

structures in the source code of the application to be examined. A design goal for the application meta model was to support the recognition of these structures. The defined use cases can be classified as follows:

Invocation patterns: Sequences of infrastructure API invocations, like the bare existence of a certain call, or temporal interrelation between calls. For example, a nested invocation pattern as in the following listing can have feature relevance: Stack-like allocation and deallocation of memory can allow a hypothetical *libc* product line to use a much simpler heap management.

```
1  void f() {
2     char *buf2 = malloc(23);
3     ...
4     free(buf2); }
5  int main() {
6     char *buf1 = malloc(42);
7     ...
8     f();
9     f();
10    free(buf1); }
```

Predicates on parameters: Special patterns in parameters of infrastructure API calls. For example, certain flags in bit masks as shown in this example:

```
1  dbenv->open(home, DB_CREATE | DB_INIT_LOCK |
2    DB_INIT_LOG | DB_INIT_MPOOL | DB_INIT_TXN, 0);
```

This application uses the Berkeley DB as a database infrastructure[3]. From the flag combination used to open a database environment object the need for the feature *Transactions* can directly be deduced.

Object or class properties: The actual type of an object passed to an infrastructure API function or the number of concurrently existing instances of a certain type.

Domain specific properties: Use cases specific to certain infrastructure and application domains. For example, the following (extremely simplified) code excerpt shows an application that spawns a second thread and only therein interacts with the database library.

```
1  void *mythread(void *param) {
2     db_result_t result;
3     db_handle_t db_handle = db_open(
          connection_string);
4     result = db_query(db_handle, "SELECT * FROM
          kunde;");
5     ...
6     db_close(db_handle);
7     return NULL; }
8  int main() {
9     pthread_t thread;
10    int ret = pthread_create(&thread,NULL,mythread,
          NULL);
11    if (ret) { /* error handling */ return 1; }
12    for (int i = 0; i < 10; ++i) do_something();
13    return pthread_join(thread, NULL); }
```

In this case it would make sense to instantiate the database product line without support for concurrency.

[3] examples_cxx/EnvExample.cpp from Berkeley DB, version 4.5.20

node type	semantics
Call	function calls, also to built-in operators; complex expressions get decomposed
If	any kind of branch; loops or `switch` statements are also mapped to this
Entry/Return	function entry/return points
Start/End	begin and possible ends of the overall control flow
Creation/ Destruction	points in the control flow where objects get created/destroyed

Table 1. Meta model elements

The concept *thread* and its representation in API calls (`pthread_create`) are domain specific, as for the analysis of the code knowledge beyond the C++ syntax and semantics is necessary.

3. Code Analysis

In order to recognize the patterns described in the last section, we implemented an analysis tool, which creates the application model in several steps. This section will explain the underlying meta model and the steps of the analysis process.

3.1. Application Meta Model

In order to recognize complex invocation patterns and to conduct data flow analysis, a control flow graph with additional data flow information was chosen to abstract from the application sources. The control flow graph consists of a few simple node types (similar to UML activity diagrams) that are shown in Table 1.

The nodes are connected by unconditional or conditional (*true/false*) edges, or by function calls/returns (*call/return*); the data flow is represented by edges pointing backwards to the previous uses of a variable. Beyond this base model further node and edge types can be defined by domain-specific semantic libraries.

Detailed information about particular constructs, e.g. types and values of expressions, in the application is not directly available in the control flow graph. This information can be queried by the underlying C++ code analysis library. Our application model only holds references into these data structures. Thereby, the application model can remain simple and focus on control flow information, while it is still possible to access this data for later transformations or queries.

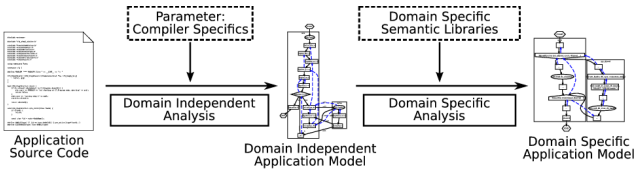

Figure 4. Application analysis steps: Domain-independent analysis is limited to the C++ language syntax and semantics. This basic application model can be extended by domain-specific semantics for domain related model queries.

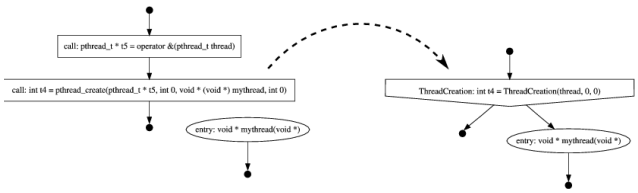

Figure 5. Graph transformation: Based on knowledge about the *thread* concept *Calls* to the library function `pthread_create` are replaced by a new node of type *ThreadCreation*.

3.2. Model Creation Steps

The initial analysis step for model creation is domain-independent and exclusively based on the syntax and semantics of C++ ("fundamental static analysis"). A domain-specific analysis extends this application model by new node and edge types, representing an adequate abstraction of the application for specific queries. Domain-specific semantic libraries control the necessary graph transformations (Fig. 4).

3.3. Domain-Independent Analysis

The process of domain-independent analysis begins with traversing the syntax tree of every translation unit of the application. Thereby nodes and edges of the basic application model are issued. The syntax trees are being generated by an underlying C++ analysis library.

In a second step the resulting control flow graphs for each translation unit are joined: *Calls* to functions implemented within the application get resolved by adding *call/return* edges from *caller* to *callee*; method calls on polymorphic objects generate edges for all possible targets.

3.4. Domain-Specific Analysis

In the case of the application that accesses the database only in one thread of control (last *use case* in subsection 2.3) the domain-independent analysis would yield an incomplete application model. The function `mythread` would seem to be dead code, because there is no control flow edge ending at this function. The model has to be enriched by additional knowledge about the concept *thread of control* and its semantic. During this domain-specific transformation step, the *Call* to the library function `pthread_create`, which creates the new thread, will be replaced by a new *ThreadCreation* node that has outgoing edges to spawned threads (Fig. 5). Besides the syntax and semantics of the programming language this *knowledge-based analysis* exploits knowledge about the *thread* concept and the meaning of library functions such as `pthread_create` or `pthread_join`.

Similar problems exist, for example, in the context of callback functions that are called asynchronously by GUI libraries or the operating system. Our approach is to handle all these cases by special domain-specific semantic libraries.

4. Model Queries

In order to (partially) automate the configuration process of an infrastructure product line, its feature tree will be augmented with queries. These model queries determine the application's need for specific features and reduce the configuration complexity for the application developer.

For example, the need for the feature *transactions* in the last example application from subsection 2.3 (an application which opens a CDbEnv object of the Berkeley DB with certain flags) could be queried informally as follows:

If and only if all following conditions apply, the feature *transactions* is needed.

- The application contains a *Call* to `CDbEnv::open()`.
- For the second parameter this condition is true: `param & 0x5C001 = 0x5C001`[4]

The ad hoc approach to evaluate this query on the application model is to manually search the control flow graph. The graph is searched for reachable *Call* nodes invoking the `open` method of an CDbEnv object, beginning at the *Start* node. The search terminates once the graph has been completely searched or a node is found which satisfies the second condition (see above).

[4] `0x5C001` corresponds to the flag combination necessary for transaction usage (`DB_INIT_LOCK | DB_INIT_LOG | DB_INIT_MPOOL | DB_INIT_TXN`) [11].

A better solution would be to find an appropriate formalism for model queries like this. Thereby it would be possible to describe and recognize complex application model structures while hiding query implementation details. For predicates on parameters, predicate logic with predicates for basic queries like "a call to function X exists" (with conditions on actual function call parameters) suffices. Recognition of complex invocation patterns would require a more powerful formalism that allows us to formulate the temporal relationship between sub-patterns. The concept of *temporal logic* [4] fulfills this requirement [8, 1, 12, 13] and, thus, has been chosen.

For the programmer we provide a description language that can be translated into expressions of temporal logic. The following listing shows how the query would look like:

```
1 feature TRANSACTIONS:
2    necessary,sufficient:
3       CDbEnv::open(?, bits_set?(0x5C001), ?)
```

The listed conditions are categorized as *necessary* and/or *sufficient* (in the mathematical sense). This allows us to reason about the configuration decision (*yes/no/maybe*[5]) from the results of evaluating each condition.

5. Implementation and Evaluation

The prototype implementation of the analysis tool is based on the *Puma* C++ analysis library originating from the AspectC++ project[6]. It can parse complex C++ applications, conduct the fundamental static analysis steps mentioned before, and creates the basic application model. Based on this library we have implemented the generation of data flow edges and support for model queries. Knowledge-based analyses and the connection from queries to the feature model of an infrastructure software product line are yet to be implemented.

A refactored, feature-oriented [2, 3] version of the Berkeley DB was chosen as a basis for evaluation of our tool prototype. It can easily be (statically) tailored by feature selection.

For a first evaluation of the approach we had to conduct a detailed analysis of the (at the time of writing) configurable features of this Berkeley DB edition: It was necessary to find out if the need for a certain feature can be deduced from an application's source code and how precisely this need manifests. As a summary of the results Table 2 shows the configurable features and the conditions to be checked within the application. The need for the features *Debug*, *ArgCheck* and *ErrorLog* is generally not deducible from an application. For the remaining 15 of 18 features it was possible to formulate suitable conditions.

The analysis of a benchmarking example application (which is part of this FC++ Berkeley DB variant) led to an application model with about 600 control flow graph nodes. The graph was searched in order to evaluate the aforementioned conditions. The "test case" column in the table shows the thereby made configuration decisions.

Note that a "no" does not necessarily mean that the product line can be configured without the respective feature. After determining the application's needs the internal feature dependencies of the infrastructure (described in its feature model) must be taken into account, too.

6. Related Work

We believe that in the product line domain the difference between application and infrastructure product lines with its implications for our problem definition is not being paid enough attention up to now. Merely the concept of *Application-Oriented System Design* [6] takes this distinction into account. Here the set of infrastructure symbols referenced by the application determines which product line variant is needed. Apart from the comparably simple static analysis the main difference to our approach is the lack of logical isolation between analysis and configuration, established by a feature model.

7. Conclusion and Future Work

The assumption that the need for features of an infrastructure software product line can be derived from an application by static analysis was confirmed by our experiments with the analysis tool prototype. As we only evaluated the tool with very simple examples, we cannot draw final conclusions regarding the scalability of our approach yet. Nevertheless, in the examined benchmarking tool all features we identified as "derivable from the application" could be recognized, and we expect that this will even succeed with far more complex applications due to the interface characteristics. In order to gain more evidence we plan to analyze more and larger applications. In the act of doing this we expect to identify more relevant types of model queries (*use cases*).

We believe that future infrastructure software product lines would benefit even more from our approach if they were already *designed* with the knowledge that feature needs can be derived from application code or already configured higher level infrastructure software layers. A reasonable compromise has to be found between infrastructure API ease of use for the application developer on the one hand and enabling a high feature detection rate on the other hand.

[5] Necessary conditions evaluated as *false* together with sufficient conditions evaluated as *true* indicate an error within the application or in the specified conditions.

[6] http://www.aspectc.org/

Feature	Condition for feature need within the application (*S*=sufficient, *N*=necessary)	Test case
Debug	-	maybe
BTree	*S*: parameter *type* of DB::Open(), flag DB_BTREE set *N*: flag DB_BTREE or DB_UNKNOWN set	yes
RecNo	analogue to BTree, with flag DB_RECNO	no
TTree	analogue to BTree, with flag DB_TTREE	no
Qam	analogue to BTree, with flag DB_QUEUE	no
Hash	analogue to BTree, with flag DB_HASH	no
Logging	*S&N*: parameter *flags* of CDbEnv::open(), flag DB_INIT_LOG set	yes
Statistic	*S&N*: usage of DB::stat*, CDbEnv::*stat*, DB_SEQUENCE::stat	no
Transaction	*S&N*: parameter *flags* of CDbEnv::open(), flags DB_INIT_TXN | DB_INIT_MPOOL | DB_INIT_LOCK | DB_INIT_LOG set	yes
XA	*S&N*: parameter *flags* of DB::Create(), flag DB_XA_CREATE set	no
ArgCheck	-	maybe
Replication	*S&N*: usage of CDbEnv::rep*	no
Crypto	*S&N*: usage of DB::*encrypt*, CDbEnv::*encrypt*	no
Verify	*S&N*: usage of DB::verify	no
Recovery	*S&N*: parameter *flags* of CDbEnv::open(), flag DB_RECOVER or DB_RECOVER_FATAL set	no
Compact	*S&N*: usage of DB::compact	no
Upgrade	*S&N*: usage of DB::upgrade	no
ErrorLog	-	maybe

Table 2. Examined Berkeley DB features, conditions for their need within the application, and the actual need in our benchmarking tool test case.

References

[1] R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. L. Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *18th IEEE Int. Conf. on Automated Software Engineering (ASE '03)*, pages 196–204, Montreal, Canada, Mar. 2003. IEEE.

[2] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *4th Int. Conf. on Generative Programming and Component Engineering (GPCE '05)*, Tallinn, Estonia, Sept. 2005.

[3] D. Batory. Feature-oriented programming and the AHEAD tool suite. In *26th Int. Conf. on Software Engineering (ICSE '04)*, pages 702–703. IEEE, 2004.

[4] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 2(8):244–263, 1986.

[5] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* AW, May 2000.

[6] A. Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, Aug. 2001.

[7] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, Nov. 1990.

[8] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. AW, 2002.

[9] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *EuroSys 2006 Conference (EuroSys '06)*, pages 191–204, New York, NY, USA, Apr. 2006. ACM.

[10] D. Lohmann, O. Spinczyk, and W. Schröder-Preikschat. Lean and efficient system software product lines: Where aspects beat objects. In A. Rashid and M. Aksit, editors, *Transactions on AOSD II*, number 4242 in LNCS, pages 227–255. Springer, 2006.

[11] Oracle Corporation. Oracle berkeley db documentation. http://www.oracle.com/technology/documentation/berkeley-db/db/index.html, 2006.

[12] Y. Padioleau, J. L. Lawall, and G. Muller. SmPL: A domain-specific language for specifying collateral evolutions in linux device drivers. In *International ERCIM Workshop on Software Evolution*, 2006.

[13] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in linux device drivers. In *EuroSys 2006 Conference (EuroSys '06)*, pages 59–71. ACM, Apr. 2006.

[14] M. Rosenmüller, T. Leich, and S. Apel. Konfigurierbarkeit für ressourceneffiziente Datenhaltung in eingebetteten Systemen am Beispiel von Berkeley DB. In *BTW Workshop on Tailor-made Data Management*, Aachen, Germany, Mar. 2007.