Maßschneiderung von Infrastruktursoftwareproduktlinien durch statische Anwendungsanalyse

Horst Schirmeier und Olaf Spinczyk

Lehrstuhl für Informatik 4
Universität Erlangen-Nürnberg
{Horst.Schirmeier,Olaf.Spinczyk}@informatik.uni-erlangen.de

Abstract: Anwendungsspezifisch konfigurierbare Daten- bzw. Datenbank-Managementsysteme können allgemein als Infrastruktursoftwareproduktlinien betrachtet werden. In diesem Bereich gibt es bereits zahlreiche Forschungsergebnisse, wie beispielsweise die merkmalgetriebene Produktableitung. In diesem Artikel gehen wir davon aus, dass ein derartig strukturiertes und durch Merkmalselektion konfigurierbares System bereits vorliegt.

Unser Ansatz basiert auf der Beobachtung, dass viele der beim Instanziieren einer Infrastrukturproduktlinie manuell zu treffenden Konfigurierungsentscheidungen durch die Anwendungssoftware bestimmt werden, die diese Infrastruktur benutzt und für die sie maßgeschneidert wird. Wir beschreiben einen Ansatz, der die Komplexität der Konfigurierung in der Regel deutlich reduziert: Die Maßschneiderung wird teilautomatisiert, indem durch statische Analyse der Anwendung der Bedarf an bestimmten Merkmalen der Infrastruktur festgestellt wird.

Motiviert durch eine Reihe von Anwendungsfällen entstand ein Analyse-Werkzeug, das ein Modell der Anwendung erzeugt und damit in vielen Fällen gestattet, die Konfigurierung zu automatisieren.

1 Einleitung

Die Maßschneiderung von Daten- oder Datenbankmanagement für ein bestimmtes Anwendungsszenario, ein konkretes Anwendungsprogramm oder einen Kunden hat verschiedene Vorteile. So könnte beispielsweise ein Hersteller eines solchen Systems eine funktionale Individualisierung für spezielle Kunden oder Kundengruppen vornehmen. Gleichzeitig werden Ressourcen gespart, da (im konkreten Anwendungsfall) unbenötigter Programmcode nicht den Speicher im Zielsystem belastet oder Prozessortaktzyklen verbraucht. Dies ist gerade im Bereich kleiner ressourcenbeschränkter Systeme von erheblicher Bedeutung.

Unsere Arbeitsgruppe befasst sich schon seit vielen Jahren mit dieser Problematik – allerdings nicht in der Datenbankdomäne, sondern im Bereich von eingebetteten Betriebssystemen [LST^{+}06] und Applikationsproduktlinien [LSSP06]. Wir glauben, dass sich viele der Ansätze, die wir dort verfolgt haben, auch in die Datenbankdomäne übertragen lassen. Beide Arten von Software kann man allgemein als „Infrastruktursoftware“ betrachten.

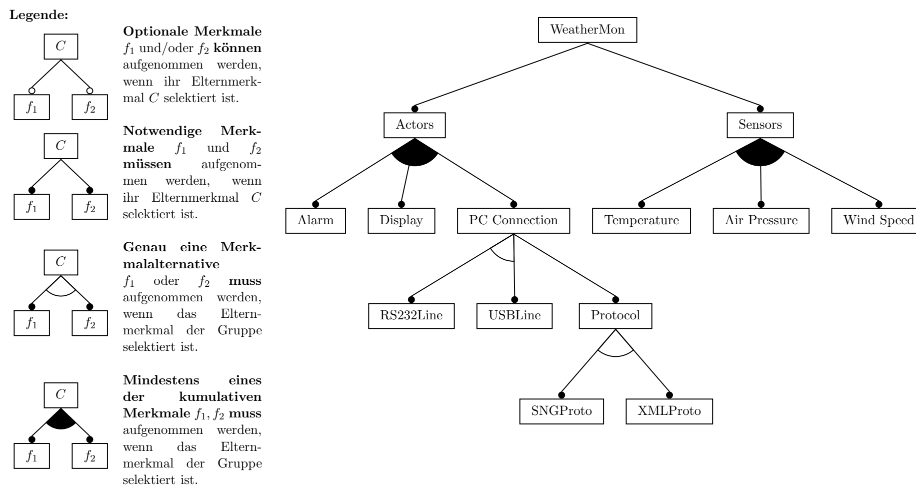

Abbildung 1: Vereinfachtes Merkmaldiagramm einer Wetterstations-Softwareproduktlinie [LSSP06]

Dieser Artikel beschreibt zunächst kurz den aus der Forschung an Software-Produktlinien [CE00] her bekannten Ansatz der merkmalgetriebenen Produktableitung. Dabei wird die Variabilität einer Software durch ein sogenanntes Merkmalmodell [KCH+90] (engl. *feature model*) beschrieben. Durch ein zweites Modell, das die Verbindung zwischen den abstrakten „Merkmalen" der Software und den Implementierungskomponenten herstellt, kann ein derartiges System durch die Selektion von Merkmalen konfiguriert werden. Trotz der Eleganz dieses Ansatzes gibt es doch noch eine Reihe von Problemen. So können die Merkmalmodelle in der Praxis sehr groß werden[1], was zu einem erheblichen manuellen Aufwand führen kann.

Der eigentliche Beitrag dieses Artikels besteht in der Beschreibung eines Ansatzes zur Konfigurierung von Infrastruktursoftwareproduktlinien. Dabei wird basierend auf statischer Anwendungsanalyse ein Modell erstellt, das es ermöglicht, die teilweise komplexen Konfigurierungsentscheidungen in sehr vielen Fällen automatisch zu treffen. Wir präsentieren dazu eine Reihe von Anwendungsfällen, Details zur Anwendungsanalyse und erste Erfahrungen mit unserem Werkzeugprototypen bei der Analyse von Anwendungen einer merkmalorientierten [ALRS05, Bat04] Version der Berkeley DB.

Der Rest dieses Artikels ist wie folgt strukturiert: Kapitel 2 beschreibt noch einmal mit weiteren Details und motivierenden Anwendungsfällen unseren Ansatz. In Kapitel 3 wird dargestellt, welche Eigenschaften der Anwendung für die Konfigurierung relevant sind und wie basierend darauf ein Anwendungsmodell aufgebaut wird. Die Anfragen an das Anwendungsmodell und die Verknüpfung mit dem Merkmalmodell der Produktlinie sind Thema von Kapitel 4. Darauf folgt eine kurze Beschreibung unserer prototypischen Werkzeugimplementierung und einer ersten Evaluation in Kapitel 5. Der Artikel endet mit einer Diskussion verwandter Arbeiten und einer Zusammenfassung in Kapitel 6 und 7.

[1] aus industriellen Anwendungen sind uns Produktlinien mit über 4000 konfigurierbaren Merkmalen bekannt.

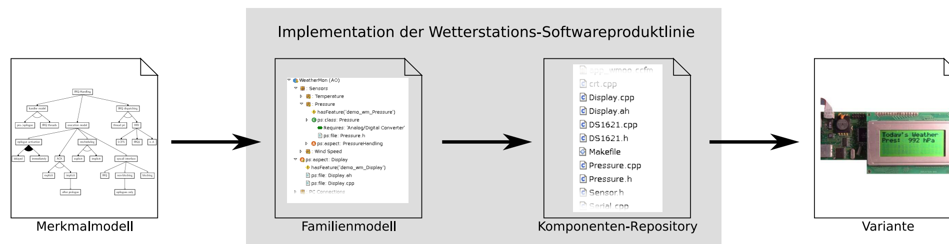

Abbildung 2: Konfigurierungsprozess der Wetterstations-Produktlinie: Zunächst selektiert man die gewünschten Merkmale im *Merkmalmodell* (1). Die Implementation beinhaltet ein *Familienmodell* und ein *Komponenten-Repository*; das Familienmodell bildet Merkmale auf *logische Implementationskomponenten* ab (2), die wiederum auf *physische Implementationsdateien* abgebildet werden (3). Die dadurch ausgewählten Dateien werden in ein Zielverzeichnis kopiert, compiliert, und in die *Wetterstationsvariante* gebunden (4).

2 Motivation und Ansatz

Infrastrukturproduktlinien

Am Anfang des Prozesses der merkmalgetriebenen Produktableitung steht das *Merkmalmodell*, dessen Merkmale untereinander bestimmten *Constraints* unterliegen können. Abb. 1 zeigt das stark vereinfachte Merkmaldiagramm einer Wetterstations-Softwareproduktlinie, aus dem bei der Konfigurierung die benötigten Merkmale selektiert werden.

Die Implementation der Wetterstationssoftware beinhaltet ein *Familienmodell*, das die selektierten Merkmale auf logische Implementationskomponenten und letztendlich physische Dateien der Implementation abbildet (Abb. 2). Die dadurch ausgewählten Dateien bilden die Codebasis für die resultierende *Variante* der Wetterstations-Produktlinie.

Bei herkömmlicher merkmalgetriebener Produktlinienkonfigurierung kommt dem Entwickler der Anwendung, die auf der Infrastrukturproduktlinie aufsetzen soll, der gesamte Konfigurierungsaufwand zu. Bei feingranular konfigurierbaren Infrastrukturproduktlinien mit mehreren tausend selektierbaren Merkmalen wächst dieser Aufwand schnell in eine Größenordnung, bei der sich der Wunsch nach Werkzeugunterstützung aufdrängt.

Neben dem direkten Zeit-/Kostenfaktor für die Konfigurierung ist bei einer so großen Anzahl von zu treffenden Konfigurierungsentscheidungen auch zu erwarten, dass menschliche Fehlentscheidungen getroffen werden, die letztendlich zu einer nicht ressourcenoptimalen Variante führen.

Anwendungsanalyse zur (teil-)automatisierten Konfigurierung

Aufgrund der inhärenten „Benutzt“-Beziehung zwischen Anwendung und Infrastruktur bietet es sich zur Automatisierung der Konfigurierung an, den Bedarf an Merkmalen der Infrastruktur aus der Anwendung abzuleiten.

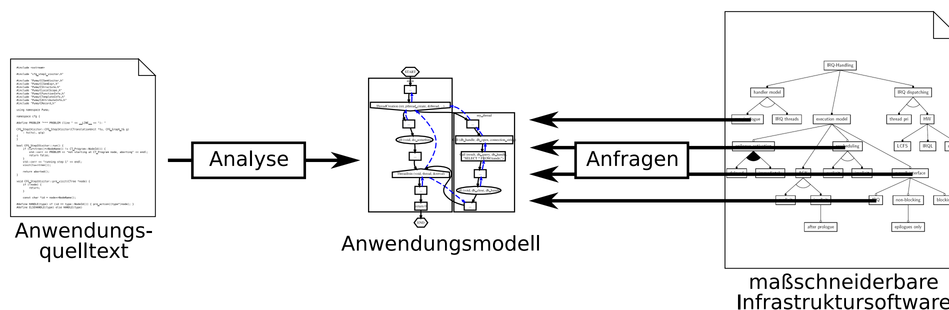

Abbildung 3: Das Anwendungsmodell, das im Analyseschritt aus dem Anwendungsquelltext entsteht, dient als Grundlage für Anfragen zur Maßschneiderung der Infrastruktursoftware.

Der hier verfolgte Ansatz sieht vor, zunächst mittels statischer Anwendungsanalyse ein Modell der Anwendung zu erzeugen, das von Details der benutzten Syntax im Anwendungsquelltext abstrahiert und für daran zu stellende Anfragen geeignet formuliert ist (Abb. 3).

Für die automatische Konfigurierung geeignete Merkmale der Infrastruktur können nun mit Anfragen an das Modell der Anwendung versehen werden, die die Frage nach dem Bedarf an diesem Merkmal beantworten (*ja/nein/evtl.*). Im Idealfall können große Teile des Merkmalbaums vorkonfiguriert werden, und dem Entwickler verbleiben nur noch diejenigen Merkmale zur Konfigurierung, die der Anwendung nicht automatisch anzusehen sind.

Use Cases

Vor dem Entwurf des Anwendungsmodells wurde eine Reihe *Use Cases* definiert, die bestimmte Strukturen in den Quellen der zu untersuchenden Anwendungen beschreiben. Entwurfsziel für das Anwendungsmodell war, die Erkennung dieser Strukturen zu ermöglichen.

Die definierten *Use Cases* lassen sich wie folgt klassifizieren:

Prädikate auf Parametern Genauere Untersuchung der Parameter, die die Anwendung der Infrastruktur-API übergibt. Hier ist beispielsweise die Prüfung auf bestimmte Flags in Bitmasken gemeint, wie in der Beispielanwendung aus Abb. 4, die die Berkeley DB als Datenbankinfrastruktur benutzt[2]. Aus der Flagkombination, mit der hier ein Datenbank-Environment-Objekt [Ora06] geöffnet wird, kann direkt der Bedarf am Merkmal *Transaktionen* geschlossen werden.[3]

Aufrufmuster Die Erkennung bestimmter Muster in Aufruffolgen der Infrastruktur-API,

[2]examples_cxx/EnvExample.cpp aus der Version 4.5.20 der Berkeley-DB

[3]Das offizielle Release der Berkeley-DB von Oracle bietet nicht die Möglichkeit, zur Übersetzungszeit ohne Unterstützung für Transaktionen konfiguriert zu werden; hier kam eine refaktorierte, merkmalorientierte [ALRS05, Bat04] Version der Berkeley DB zum Einsatz.

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

Abbildung 4: Ausschnitt einer Beispielanwendung, die die Berkeley-DB benutzt: Aus der Flagkombination lassen sich Rückschlüsse auf den Bedarf an bestimmten Merkmalen ziehen.

```
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  }
9  int main() {
10    pthread_t thread;
11    int ret = pthread_create(&thread, NULL, mythread, NULL);
12    if (ret) { /* error handling */ return 1; }
13    for (int i = 0; i < 10; ++i) do_something();
14    return pthread_join(thread, NULL);
15 }
```

Abbildung 5: Ein Merkmal-relevantes Aufrufmuster: Wenn die Benutzung dieser Datenbankbibliothek auf einen einzigen Faden beschränkt ist, kann diese ohne jegliche Unterstützung von Nebenläufigkeit konfiguriert werden.

beispielsweise Stack-artige Belegung und Freigabe von Ressourcen, oder Beschränkung der API-Benutzung auf einen einzigen Faden in einer mehrfädigen Anwendung. Der in Abb. 5 gezeigte (extrem vereinfachte) Code-Abschnitt zeigt z.B. eine Anwendung, die einen zweiten Faden erzeugt und nur in diesem mit einer Datenbankbibliothek interagiert: In diesem Fall wäre es denkbar, die benutzte Datenbankproduktlinie ohne Unterstützung für Nebenläufigkeit zu instantiieren.

Datenflussanalyse Die Verfolgung von Variableninhalten und -typen in der Anwendung, beispielsweise zur Feststellung der maximalen Anzahl gleichzeitig geöffneter Datenbanken und Transaktionen, sofern diese durch Objekte repräsentiert werden.

3 Analyse-Werkzeug

Zur Erzeugung des Anwendungsmodells wurde der Prototyp eines Analyse-Werkzeugs geschaffen, das in mehreren Schritten aus den C++-Quellen der zu untersuchenden Anwendung eine Graphdatenstruktur erzeugt.

Anwendungsmodell

Aus der Anforderung, die in den *Use Cases* definierten Strukturen in einer untersuchten Anwendung erkennen zu können, entstand das Anwendungsmodell. Zur Erkennung komplexerer Aufrufmuster und zur Analyse des Datenflusses bot sich hier ein Kontrollflussgraph mit zusätzlichen Datenflussinformationen an. Der Kontrollflussgraph besteht

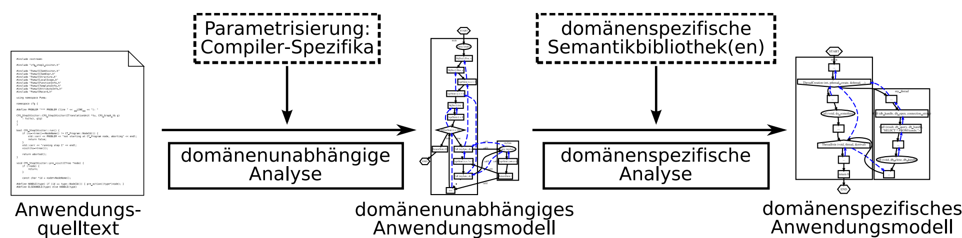

Abbildung 6: Die Analyse der Anwendung erfolgt schrittweise: Die domänenunabhängige Analyse beschränkt sich auf den Sprachumfang von C/C++; die darauf aufsetzenden Domänenspezifika ermöglichen die Erweiterung des Anwendungsmodells für domänenbezogene Modellanfragen.

(ähnlich UML-Aktivitätsdiagrammen) aus wenigen einfachen Knotentypen:

Knotentyp	Semantik
Call	Funktionsaufrufe, auch von eingebauten Operatoren; komplexe Ausdrücke werden zerlegt
If	jegl. Verzweigungen; auch Schleifen oder `switch`-Statements werden darauf abgebildet
Entry/Return	Ein-/Austrittspunkte von Funktionen
Start/End	Beginn und die möglichen Enden des Gesamtkontrollflusses
Creation/ Destruction	Punkte im Kontrollfluss, an denen Objekte erzeugt oder zerstört werden

Die Knoten sind über unbedingte oder bedingte (*true*/*false*-)Kanten, oder über Funktionsaufrufe bzw. -rückkehr (*call*/*return*) verbunden; der Datenfluss wird über rückwärts gerichtete Kanten auf die jeweils vorhergehende Benutzung einer Variable dargestellt. Über dieses Grundmodell hinaus können domänenspezifische Semantikbibliotheken weitere Knoten- und Kantentypen definieren, die für spezielle Anfragetypen benötigt werden.

Detailinformationen über die jeweiligen Konstrukte in der Anwendung sind nicht direkt im Kontrollflussgraphen verfügbar, sondern sind von dort über Referenzen in die Datenbank der unterliegenden C++-Analysebibliothek erreichbar. Das ermöglicht es, das Anwendungsmodell einfach zu halten und auf den Kontrollfluss zu konzentrieren, diese Informationen aber dennoch für spätere Transformationen oder Anfragen zur Verfügung zu haben.

Schritte zur Modellerzeugung

Die Analyse der Anwendung zur Erzeugung des Modells (Abb. 6) geschieht zunächst domänenunabhängig, d.h. rein auf Syntax und Semantik von C++ basierend („grundlegende statische Analyse“ [KP96]). Das daraus entstehenden Anwendungsmodell wird in der domänenspezifischen Analyse um neue Knoten- oder Kantentypen erweitert, die eine geeignete Abstraktion der Anwendung für spezielle Anfragen bereitstellen. Domänenspezifische Semantikbibliotheken steuern die dafür notwendigen Transformationen des Graphen.

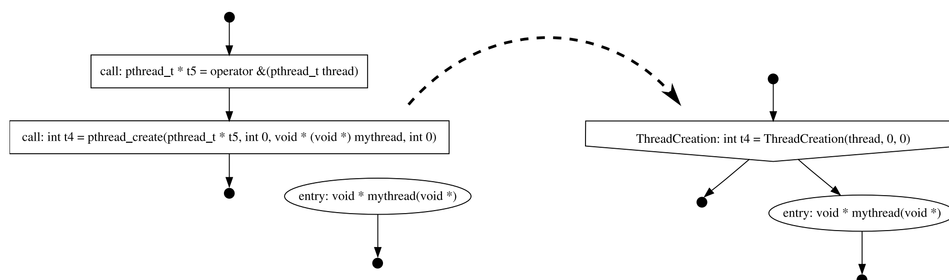

Abbildung 7: Graphtransformation: Mit dem Wissen um das Konzept „Faden" werden *Calls* der Bibliotheksfunktion `pthread_create` durch einen neuen Knotentyp (*ThreadCreation*) ersetzt.

Schrittweise Transformation des Syntaxbaums

In der domänenunabhängigen Analyse wird zunächst für jede Übersetzungseinheit der Anwendung der Syntaxbaum, der von der unterliegenden C++-Analysebibliothek geliefert wird, durchlaufen. Dabei werden die Knoten und Kanten des Grundmodells erzeugt.

In einem zweiten Schritt werden die entstandenen Graphen der einzelnen Übersetzungseinheiten zusammengefügt: *Calls* zu Funktionen/Methoden, deren Implementation innerhalb der Anwendung liegt, werden aufgelöst und *call-*/*return*-Kanten zwischen *Caller* und *Callee* gelegt; Methodenaufrufe an polymorphen Objekten erzeugen in dem Fall für alle möglichen Ziele Kanten.

Domänenspezifische Analysen

Im eingangs motivierenden *Use Case* der Anwendung, die in einem zweiten Faden Zugriffe auf die Datenbank durchführt (Abb. 5), muss nach der domänenunabhängigen Anwendungsanalyse weitere Semantik in das Modell gesteckt werden, um Anfragen bezüglich des Konzepts „Faden" formulieren zu können. Der *Call*-Knoten, der die Bibliotheksfunktion `pthread_create` zum Erzeugen eines neuen Fadens ruft, wird in diesem Beispiel mit einem neuen Knotentyp (*ThreadCreation*) ersetzt, der für jeden entstehenden Faden eine ausgehende Kante besitzt (Abb. 7).

Hier handelt es sich um wissensbasierte Analyse [KP96]: Das Wissen um das Konzept „Faden" und seine Repräsentation in Bibliotheksaufrufen wie `pthread_create` oder `pthread_join` geht über Syntax und Semantik der Programmiersprache C++ hinaus. Ähnliche Probleme existieren beispielsweise bei Callback-Funktionen, die von GUI-Bibliotheken oder dem Betriebssystem asynchron aufgerufen werden können. Auch hier wird eine spezielle Semantikbibliothek benötigt.

Um wissensbasierte Analysen später in externe Semantikbibliotheken auslagern zu können, die bei Bedarf mit einer Infrastrukturproduktlinie ausgeliefert werden können, wird eine Formalisierung der Domänenspezifika samt ihrer Transformationen notwendig; Ansätze in dieser Richtung finden sich in [ÅLS+03, KP96].

4 Modellanfragen

Um letztendlich teilautomatisierte Konfigurierung der unterliegenden Infrastrukturproduktlinie zu erreichen, muss deren Merkmalbaum um Anfragen angereichert werden. Diese stellen den Bedarf der Anwendung an einzelnen Merkmalen fest, und vermindern damit den Konfigurierungsaufwand für den Entwickler.

Im Beispiel der Anwendung, die ein CDbEnv-Objekt der Berkeley-DB mit bestimmten Flags öffnet (Abb. 4) könnte eine solche Anfrage zunächst informell so beschrieben werden:

Genau dann, wenn alle folgenden Bedingungen zutreffen, wird das Merkmal *Transaktionen* benötigt.

- In der Anwendung existiert ein *Call* zur Methode `CDbEnv::open()`.
- Für den zweiten Parameter gilt: `param & 0xB8000 = 0xB8000`[4]

Der Ad-Hoc-Ansatz, diese Anfrage an das Anwendungsmodell auszuwerten, besteht in der manuellen Suche im Kontrollflussgraphen. Der Graph wird, ausgehend vom *Start*-Knoten, nach erreichbaren *Call*-Knoten abgesucht, die die Methode `open` eines Objekts vom Typ CDbEnv rufen. Die Suche terminiert, sobald der Graph komplett abgesucht oder ein Knoten gefunden wurde, auf den die zweite Bedingung zutrifft (s.o.).

Besser wäre es, einen geeigneten Formalismus für die Formulierung der Anfragen zu finden, der auch die Beschreibung und Erkennung sehr komplexer Strukturen im Anwendungsmodell zulässt, aber gleichzeitig die Komplexität des Kontroll- und Datenflussgraphen verbirgt.

Für das Formulieren von Prädikaten auf Parametern genügt Prädikatenlogik, die Prädikate für grundlegende Anfragen wie „es existiert ein Call zur Methode X“ vorsieht und mit den aktuellen Parametern dieses Calls parametrisierbar ist. Das Auffinden komplexer Aufrufmuster in der Benutzung der Infrastruktur-API erfordert mächtigere Formalismen, die es erlauben, den temporalen Zusammenhang zwischen Teilmustern zu formulieren; das Konzept der temporalen Logik [CES86] scheint hier vielversprechend [Lam02, ÅLS+03, PLM06a, PLM06b].

Eine programmiererfreundlichere Beschreibungssprache, die automatisch in Ausdrücke der obengenannten Formalismen übersetzt werden kann, könnte etwa die einleitende Anfrage folgendermaßen formulieren:

```
1 feature TRANSACTIONS:
2    necessary,sufficient: CDbEnv::open(?, bits_set?(0xB8000), ?)
```

Die aufgeführten Bedingungen werden im mathematischen Sinne als *notwendig* und/oder *hinreichend* kategorisiert, um aus dem (Nicht-)Zutreffen der Bedingung(en) auf die passende Konfigurierungsentscheidung (*ja/nein/evtl.*[5]) schließen zu können.

[4] `0xB8000` entspricht der Flagkombination, die notwendig ist, um Transaktionen einsetzen zu können (`DB_INIT_LOCK | DB_INIT_LOG | DB_INIT_MPOOL | DB_INIT_TXN`) [Ora06].

[5] Im Fall von nicht erfüllten notwendigen und gleichzeitig erfüllten hinreichenden Bedingungen liegt ein Fehler im Anwendungsprogramm vor.

5 Implementierung und Evaluation

Die auf der Puma-Bibliothek des AspectC++-Projekts[6] basierende prototypische Implementierung des Analyse-Werkzeugs ist in der Lage, komplexe C++-Anwendungen zu parsen, grundlegende und wissensbasierte Analysen durchzuführen und ein Anwendungsmodell zu erstellen. Die Erzeugung von Datenflusskanten, die Verarbeitung formalisierter Wissensbibliotheken und Anfragen, und die letztendliche Verknüpfung von Anfragen mit dem Merkmalmodell einer Infrastruktursoftwareproduktlinie stehen noch aus.

Als Ausgangsbasis für die Evaluation unseres Werkzeuges bot sich eine refaktorierte, merkmalorientierte [ALRS05, Bat04] Version der Berkeley DB an, die leicht statisch durch Merkmalselektion konfigurierbar ist. Zur ersten Evaluation unseres Ansatzes untersuchten wir die konfigurierbaren Merkmale dieser Fassung der Berkeley-DB genauer, um festzustellen, ob und auf welche Art und Weise einer Anwendung der Bedarf an diesen Merkmalen anzusehen ist. Die folgende Tabelle zeigt die konfigurierbaren Merkmale und die zu überprüfende Bedingung in der Anwendung:

Merkmal	**Erkennung des Bedarfs in der Anwendung** (*H*=hinreichend, *N*=notwendig)	**Testfall**
Debug	-	evtl.
BTree	*H*: Parameter type von DB::Open(), Flag DB_BTREE gesetzt *N*: Flag DB_BTREE oder DB_UNKNOWN gesetzt	ja
RecNo	analog BTree, mit Flag DB_RECNO	nein
TTree	analog BTree, mit Flag DB_TTREE	nein
Qam	analog BTree, mit Flag DB_QUEUE	nein
Hash	analog BTree, mit Flag DB_HASH	nein
Logging	*H&N*: Parameter flags von CDbEnv::open(), Flag DB_INIT_LOG gesetzt	ja
Statistic	*H&N*: Benutzung von DB::stat*, CDbEnv::*stat*, DB_SEQUENCE::stat	nein
Transaction	*H&N*: Parameter flags von CDbEnv::open(), Flags DB_INIT_TXN | DB_INIT_MPOOL | DB_INIT_LOCK | DB_INIT_LOG gesetzt	ja
XA	*H&N*: parameter flags von DB::Create(), Flag DB_XA_CREATE gesetzt	nein
ArgCheck	-	evtl.
Replication	*H&N*: Benutzung von CDbEnv::rep*	nein
Crypto	*H&N*: Benutzung von DB::*encrypt*, CDbEnv::*encrypt*	nein
Verify	*H&N*: Benutzung von DB::verify	nein
Recovery	*H&N*: Parameter flags von CDbEnv::open(), Flag DB_RECOVER oder DB_RECOVER_FATAL gesetzt	nein
Compact	*H&N*: Benutzung von DB::compact	nein
Upgrade	*H&N*: Benutzung von DB::upgrade	nein
ErrorLog	-	evtl.

Der Bedarf an den Merkmalen *Debug*, *ArgCheck* und *ErrorLog* ist einer Anwendung allgemein nicht anzusehen; für die restlichen 15 der 18 Merkmale ließen sich geeignete Bedingungen aufstellen.

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

Die Analyse einer Benchmark-Beispielanwendung, die Teil der FC++-Variante der Berkeley DB ist, führte zu einem Anwendungsmodell mit etwa 600 Kontrollflussgraph-Knoten. Der Graph wurde durchsucht, um die oben genannten Bedingungen auszuwerten; die Spalte „Testfall" der Tabelle zeigt die so getroffenen Konfigurierungsentscheidungen.

Zu beachten ist allerdings, dass ein „nein" hier noch nicht zwingend bedeutet, dass die Produktlinie ohne dieses Merkmal konfiguriert werden kann: Nach Feststellen des Bedarfs der Anwendung müssen die internen Merkmalabhängigkeiten der Infrastruktur berücksichtigt werden, die durch das Merkmalmodell beschrieben werden.

6 Verwandte Arbeiten

Wir meinen, dass im Bereich der Produktlinien bisher der Unterschied zwischen Anwendungs- und Infrastrukturproduktlinie mit seinen Implikationen für unsere Problemstellung zu wenig beachtet wird. Lediglich das Konzept des *Application-Oriented System Design* [Frö01] berücksichtigt diesen Unterschied. Dabei wird aus der Menge der aus der Anwendung referenzierten Symbole der Infrastruktur geschlossen, welche Variante der Produktlinie benötigt wird. Neben der vergleichsweise einfachen statischen Analyse besteht der Hauptunterschied zu unserem Ansatz darin, dass die durch ein Merkmalmodell geschaffene logische Entkopplung zwischen der Analyse und der Konfigurierung fehlt.

7 Zusammenfassung und Ausblick

Die Vermutung, dass man durch Anwendungsanalyse den Bedarf an Merkmalen einer Infrastruktursoftwareproduktlinie feststellen kann, hat sich durch erste Versuche mit dem Prototypen unseres Analysewerkzeuges bestätigt. Da wir das Werkzeug jedoch nur mit sehr einfachen Beispielen evaluiert haben, können wir noch keine endgültigen Schlüsse über die Skalierbarkeit des Ansatzes treffen; in der untersuchten Benchmark-Anwendung konnten jedoch alle Merkmale, die wir als aus der Anwendung ableitbar identifiziert haben, erkannt werden, und wir vermuten, dass dies aufgrund der Schnittstellencharakteristik auch bei sehr viel komplexeren Anwendungen oft funktionieren wird. Zur Untermauerung dieser Vermutung werden wir weitere und vor allem größere Anwendungen analysieren müssen. Dabei sollen auch weitere relevante Anfragearten (*Use Cases*) identifiziert werden.

Um im Rahmen des Datenmanagements von automatisierter Produktlinienkonfigurierung noch stärker profitieren zu können, sollten feingranular konfigurierbare DB-Infrastruktursoftwareproduktlinien in Zukunft bereits mit dem Wissen *entworfen* werden, dass zur Konfigurierung der Bedarf an Merkmalen aus der Anwendung abgeleitet werden kann. Hierbei muss ein vernünftiger Kompromiss gefunden werden zwischen einer für den Entwickler der Anwendung angenehm benutzbaren API der Infrastruktur einerseits und der Ermöglichung einer hohen Rate an Merkmalselektionsentscheidungen andererseits.

Literatur

[ALRS05] Sven Apel, Thomas Leich, Marko Rosenmüller und Gunter 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, September 2005.

[ÅLS+03] Rickard A. Åberg, Julia L. Lawall, Mario Südholt, Gilles Muller und Anne-Françoise Le 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)*, Seiten 196–204, Montreal, Canada, Marz 2003. IEEE.

[Bat04] Don Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *26th Int. Conf. on Software Engineering (ICSE '04)*, Seiten 702–703. IEEE, 2004.

[CE00] Krysztof Czarnecki und Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* AW, Mai 2000.

[CES86] E. M. Clarke, E. A. Emerson und A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 2(8):244–263, 1986.

[Frö01] A. Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, August 2001.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak und S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Bericht, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, November 1990.

[KP96] Rainer Koschke und Erhard Plödereder. Ansätze des Programmverstehens. In Franz Lehner, Hrsg., *Softwarewartung und Reengineering*, Seiten 159–176. Gabler Edition Wissenschaft, Deutscher Universitätsverlag, 1996. The language of this article is German.

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. AW, 2002.

[LSSP06] Daniel Lohmann, Olaf Spinczyk und Wolfgang Schröder-Preikschat. Lean and Efficient System Software Product Lines: Where Aspects Beat Objects. In Awais Rashid und Mehmet Aksit, Hrsg., *Transactions on AOSD II*, number 4242 in LNCS, Seiten 227–255. Springer, 2006.

[LST+06] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk und Wolfgang Schröder-Preikschat. A Quantitative Analysis of Aspects in the eCos Kernel. In *EuroSys 2006 Conference (EuroSys '06)*, Seiten 191–204. ACM, April 2006.

[Ora06] Oracle Corporation. Oracle Berkeley DB Documentation. http://www.oracle.com/technology/documentation/berkeley-db/db/index.html, 2006.

[PLM06a] Yoann Padioleau, Julia L. Lawall und Gilles Muller. SmPL: A Domain-Specific Language for Specifying Collateral Evolutions in Linux Device Drivers. In *International ERCIM Workshop on Software Evolution*, 2006.

[PLM06b] Yoann Padioleau, Julia L. Lawall und Gilles Muller. Understanding Collateral Evolution in Linux Device Drivers. In *EuroSys 2006 Conference (EuroSys '06)*, Seiten 59–71. ACM, April 2006.