Betriebssystemtechnik

Operating System Engineering (OSE)

Trennung der Belange bei der statischen Konfigurierung

Ergebnis der Problemanalyse

(siehe eCos Fallstudie)

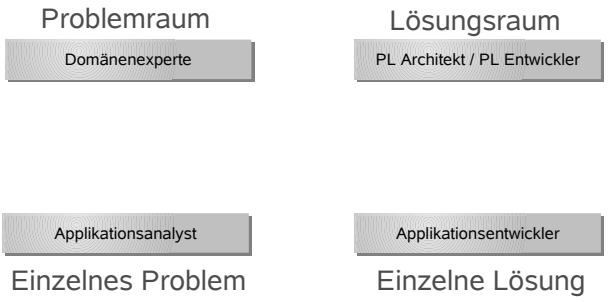
- Klassische Umsetzung der Konfigurierungsentscheidungen in den Komponenten mit Hilfe von #ifdef und Makros
  - Schutz vor ungewollten Ersetzungen nur durch strikte Namenskonvention
- **mangelnde Trennung der Belange**
  - viel Konfigurierungswissen ist im Quellcode verankert
  - quer schneidende Belange blähen die Funktionen auf
  - bedingte Übersetzung macht den Code schwer verständlich, zu warten und wiederzuverwenden

Lösungsansätze

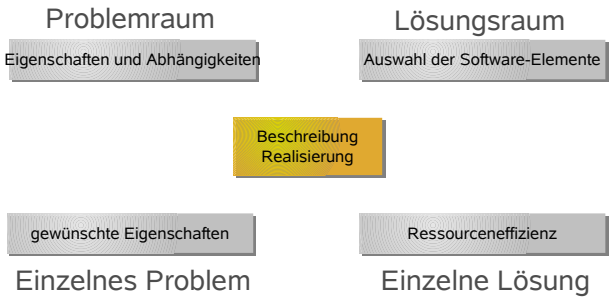
(durch statische Konfigurierungswerkzeuge)

- Variantenmanagement – z.B. **pure::variants** [1, 2]
  - zusätzliche (programmierbare) Ebene oberhalb der Komponenten
  - flexible Generierungs- und Transformationsmöglichkeiten
  - automatisierte Abbildung von Anforderungen (z.B. Merkmalselektion) auf Softwarevarianten
- *Frame* Prozessoren – z.B. **XVCL** [3]
  - Quellcode ist nur ein „Rahmen“, der je nach Konfiguration mit zusätzlichem Code angereichert werden kann
  - Ansatz ist unabhängig von der Programmiersprache
    - selbst Dokumentation lässt sich damit konfigurieren

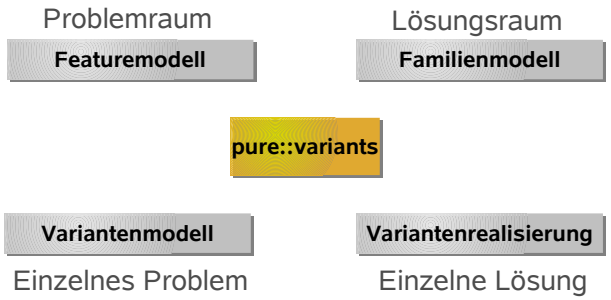
Rollenverteilung bei der PL-Entwicklung



Technische Herausforderung



Variantenmanagement mit p::v



p::v Modellgrundstruktur

- Alle Modelle besitzen gleiche Grundstruktur
  - **Element** ist Grundbaustein
  - Elemente können gerichtete Beziehungen zu anderen Elemente haben (**Relationen**)
  - Elemente können **Attribute** besitzen
  - An vielen Modellbestandteilen können **Restriktionen** die Gültigkeit einschränken
- Speicherung im XML-Format

p::v Element

- Jedes Element gehört zu einer Klasse: ps:feature, ps:component, ps:part, ps:source
- Jedes Element hat einen Typ, z.B. `ps:file`
- Typsystem ist frei konfigurierbar, p::v benutzt fast immer nur die Elementklassen
  - Ausnahme: Standardtransformation
- Standardinformationen
  - ID, Unique Name, Visible Name, Beschreibung
- Optional: Relationen, Restriktionen

p::v Elementrelationen

- Elementrelationen sind „1:n“ Verbindungen
- Relationen werden immer im Ausgangselement gespeichert
- Relationen besitzen einen Typ
- p::v definiert einige Relationen wie z.B. „ps:requires“ (und deren Semantik)
- Benutzer kann eigene Relationstypen einführen
- Jede Relation kann eine Beschreibung besitzen
- Eine Relation kann Restriktionen besitzen
  - Relation ist nur gültig, wenn Restriktion „wahr“ liefert

p::v Elementattribute (1)

- Jedem Element kann eine beliebige Anzahl von Attributen zugeordnet sein.
- Ein Attribut ...
  - ist getypt (ps:string, ps:boolean, ...)
  - hat einen Namen
  - kann Restriktionen besitzen
- Der Attributwert ...
  - kann im Modell festgelegt werden (*fixed*)
  - kann in der Variantenbeschreibung gesetzt werden (*not fixed*)
- Beispiel: Merkmal Motor
  - $(\text{Attribut, Wert})_1$ = („Leistung [kW]“, 85)
  - $(\text{Attribut, Wert})_2$ = („Zylinder“, 4)

p::v Elementattribute (2)

- Attributwerte
  - Konstante
  - Berechnung (calculation)
- Jede Wertdefinition kann eine Restriktion besitzen
- Reihenfolge der Werte ergibt Berechnungsfolge
- Erste gültige Wertdefinition bestimmt Attributwert
- Ein Attribut eines ausgewählten Elements muss einen gültigen Wert haben
- „*Not fixed*“ Attribute können eine Defaultwertdefinition besitzen

p::v Restriktionen

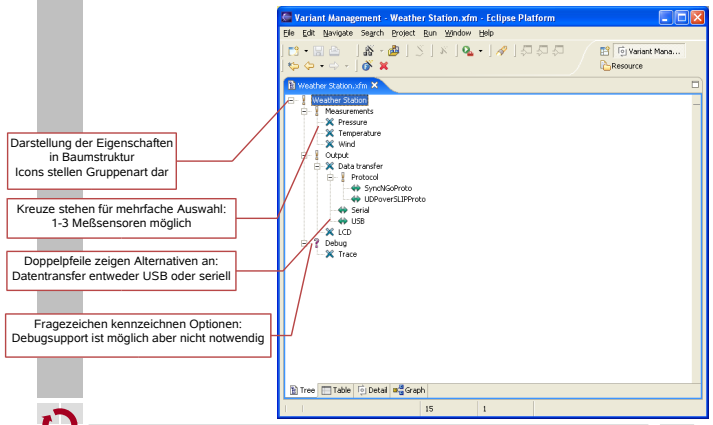
- Eine Restriktion schränkt die Gültigkeit/Verwendbarkeit des zugeordneten Bestandteils ein
- p::v verwendet eine an OCL angelehnte Notation, die von pvProlog ausgewertet wird
- Die genaue Semantik wird durch die Art des Bestandteils bestimmt
- Beispiel:

```
hasFeature('A') or not(hasFeature('B'))
```

p::v Featuremodell (1)

- Ein Featuremodell enthält nur Elemente der Klasse ps:feature
- Es gibt 4 Arten von Featuregruppen
  - notwendig (*ps:mandatory*) [n]
  - optional (*ps:optional*) [0-n]
  - alternativ (*ps:alternativ*) [1]
  - oder (*ps:or*) [1-n]*
- Jedes Feature kann von jeder Gruppenart maximal eine Gruppe besitzen

p::v Featuremodell (2)



p::v Featuremodell (3)

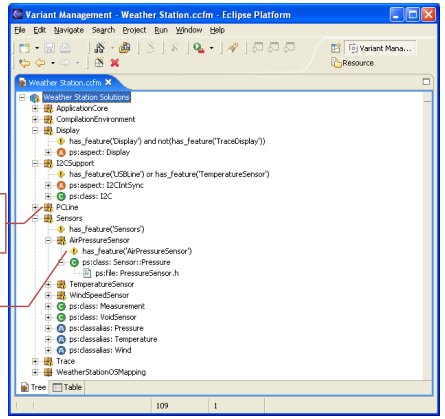
- Beziehungen zwischen Features (und auch anderen Modellelementen)
  - Gegenseitiger Ausschluß (*ps:conflicts, ps:discourages*)
  - Implikation (*ps:requires, ps:required_for, ps:recommends, ps:recommended_for*)
  - ... (erweiterbar)
- Semantik von 1:n Beziehung unterschiedlich
  - ps:conflicts und Co. : AND (Fehler, falls alle n selektiert sind)
  - ps:requires und Co. : OR (Fehler, falls keines der n selektiert ist)

p::v Familienmodell (1)

- Darstellung eines Lösungsraums
- Hierarchische Gruppierung von Elementklassen
  - Komponenten (*component*) enthalten Teile (*part*)
  - Teile enthalten weitere Teile und/oder Quellen (*source*)
- Teile und Quellen sind getypt
  - Typen werden in der Transformation ausgewertet
  - Typ gibt notwendige und optionale Attribute vor
- Restriktion ist Vorbedingung für Auswahl

p::v Familienmodell (2)

Die Familienmodelle erfassen den Aufbau der Lösung von der abstrakten variablen Architektur bis hin zu den notwendigen Dateien. Neben der hier gezeigten Möglichkeit, die Informationen direkt in pure::variants zu bearbeiten, kann auch eine Kopplung mit anderen Modellierungswerkzeugen erfolgen.



Die Icons stellen den Typ des Modellelements dar. Hier eine Komponente. Diese Typen sind frei definierbar (entsprechend der Architektur).

Diese Regeln steuern die Auswahl von Lösungselementen. Hier soll die Komponente „AirPressureSensors" nur dann zur Lösung gehören, wenn das entsprechende Feature gleichen Namens gewählt wird.

p::v Konfigurationsraum (1)

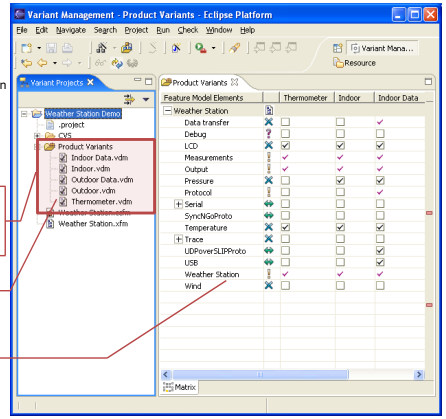
(*configuration* space)

Zusammenstellung von Modellen für die gemeinsame Konfiguration

- Feature- und Familienmodelle können in mehreren Konfigurationsräumen verwendet werden
- speichert die Parameter für die (optionale) Transformation
- enthält beliebig viele Variantenbeschreibungen

p::v Konfigurationsraum (2)

In pure::variants werden Produktvariabilitäten (Featuremodelle) und Lösungsarchitekturen (Familienmodelle) flexibel zu Produktlinien kombiniert. Eine Produktlinie kann aus beliebig vielen Modellen bestehen. Der „Configuration Space" verwaltet diese Informationen und fasst die Varianten einer Produktlinie zusammen.



Der Configuration Space „Product Variants" fasst die Variantendefinitionen für die Produkte zusammen.

Jedes Modell repräsentiert eine Variante

Diese Variantenmatrix ermöglicht einen Überblick über Unterschiede und Gemeinsamkeiten der Produktvarianten

p::v Variantenbeschreibung (1)

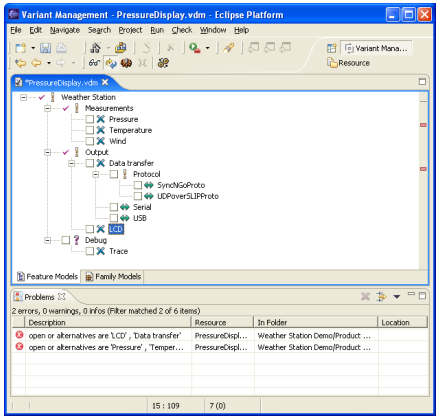
(*variant description model*)

definiert eine Variante aus den Möglichkeiten eines Konfigurationsraums

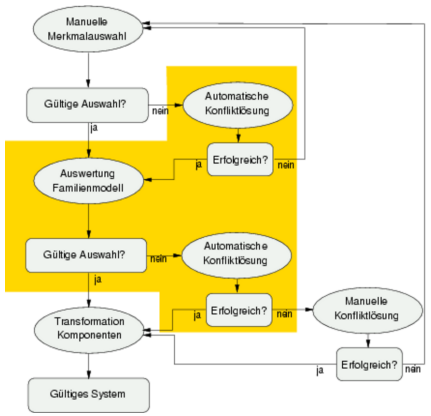
- enthält
  - alle gewählten Features
  - alle durch Anwender spezifizierten Attributewerte

p::v Variantenbeschreibung (2)

Die Variantenerstellung erfolgt durch Auswahl der gewünschten Feature im Variantenmodell. pure::variants prüft die Auswahl interaktiv und löst oder meldet auftretende Probleme.



p::v Modellauswertung

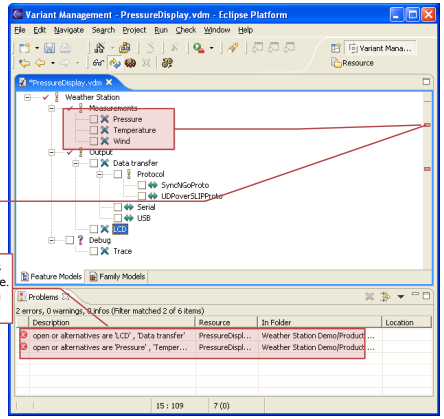


p::v Variantenerstellung

Die Variantenerstellung erfolgt durch Auswahl der gewünschten Feature im Variantenmodell. pure::variants prüft die Auswahl interaktiv und löst oder meldet auftretende Probleme.

Durch einen Doppelklick wird zur Problemstelle navigiert.

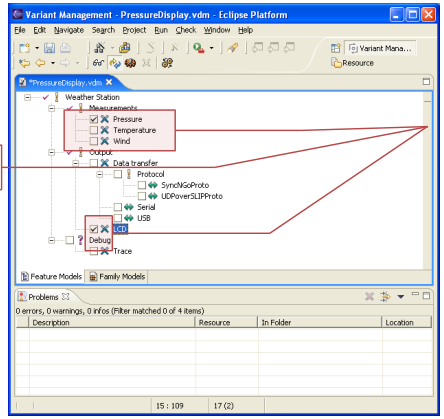
Die Auswertung eines neuen Variantenmodells ergibt hier 2 vom Anwender zu lösende Probleme. In den beiden Mehrfachauswahlen wurde noch nichts ausgewählt.



p::v Variantenerstellung

Die Variantenerstellung erfolgt durch Auswahl der gewünschten Feature im Variantenmodell. pure::variants prüft die Auswahl interaktiv und löst oder meldet auftretende Probleme.

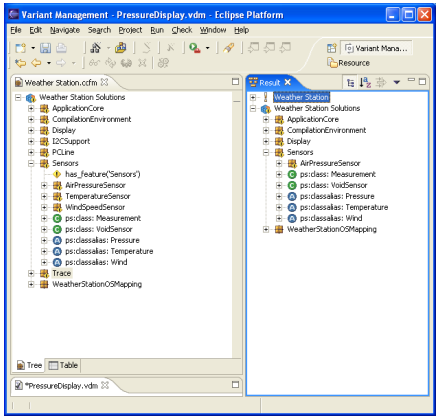
Die Auswahl von mindestens einem Feature (Pressure, LCD) löste das Problem.



p::v Resultatsansicht

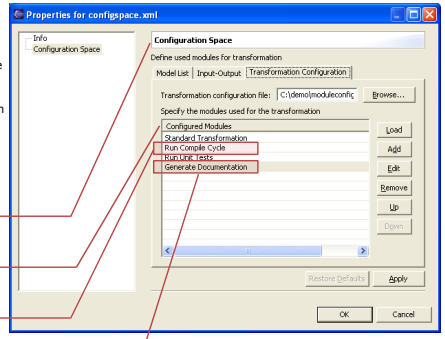
Die Resultatsicht (Result view) stellt die errechnete Lösung in (konfigurierbarer) Form dar als Baum oder Tabelle dar.

Export über das Kontextmenü möglich.



p::v Variantentransformation

Neben den Exportmöglichkeiten in verschiedene Formate wie HTML, CSV/Excel oder XML bietet pure::variants einen Mechanismus mit dem aus Variantenmodellen die konkrete Lösung erzeugt wird. Dieser Transformationsmechanismus kann durch die Anwender beliebig erweitert und angepasst werden, sollten die Möglichkeiten der mitgelieferten Module nicht ausreichen.



Eine Transformationskonfiguration wird einem Configuration Space zugeordnet.

Die Module werden in der angegebenen Reihenfolge ausgeführt.

Die Einbindung externe Werkzeuge ist einfach und schnell möglich.

Mitgelieferte Module erlauben zum Beispiel die Erzeugung variantenspezifischer Dokumentation direkt aus Microsoft Word Dokumenten.

p::v Standardtransformationen

... erlauben im Wesentlichen:

- das Kopieren von Dateien in den Lösungsbaum
- das Zusammenstellen von Dateien aus Fragmenten
- das Erstellen von Links (nicht unter Windows)
- das Generieren von „Flag-Files“

```
#ifndef __flag_XXX
#define __flag_XXX
#define XXX 1
#endif // __flag_XXX
```

- das Generieren von „Class Aliases“

```
#ifndef __alias_YYY
#define __alias_YYY
#include “ZZZ.h“
typedef ZZZ YYY;
#endif // __alias_YYY
```


pure::variants - Zusammenfassung

- p::v ist ein als Eclipse-*Plugin* realisiertes kommerzielles Werkzeug zum Variantenmanagement
- p::v unterstützt ...
  - die Erfassung von Domänenwissen
  - die Beschreibung des Lösungsraums (die „Plattform“)
  - die Beschreibung konkreter Applikationsanforderungen
  - die Kontrolle der Lösung durch ein Resultatsmodell
- zur Auswertung von Attributen, Relationen, Restriktionen, u.s.w. steht dem Entwickler die volle Leistungsfähigkeit von Prolog zur Verfügung
- die Generierung der Lösung basiert auf einer einfachen „Standardtransformation“, die aber erweitert und um weitere Transformationen ergänzt werden kann

pure::variants erlaubt das automatische Generieren von Applikationen anhand von Merkmalselektionen.

XVCL – ein *Frame* Prozessor

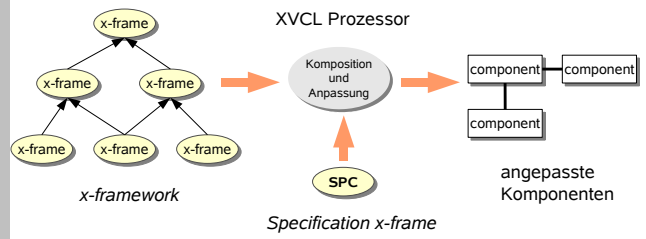
- *XML-based Variant Configuration Language*
- Was ist ein *frame?*

> *„When one encounters a new situation (or makes a substantial change in one's view of a problem) one selects from memory a structure called a **frame**. This is a remembered framework to be adapted to fit reality by changing details as necessary.“*

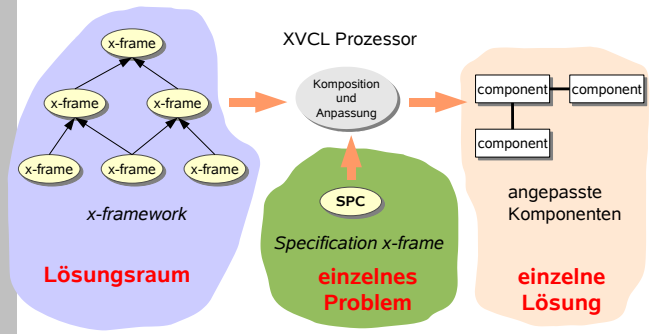
M. Minsky
A Framework for Representing Knowledge
1975

- XVCL arbeitet mit *frames* in Form von Textdateien

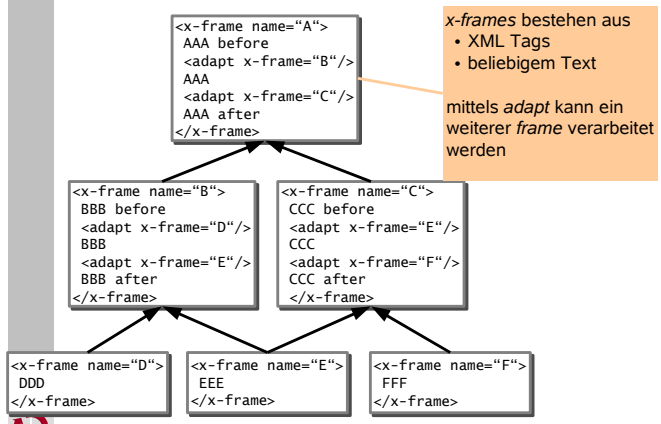
XVCL – Grundbegriffe



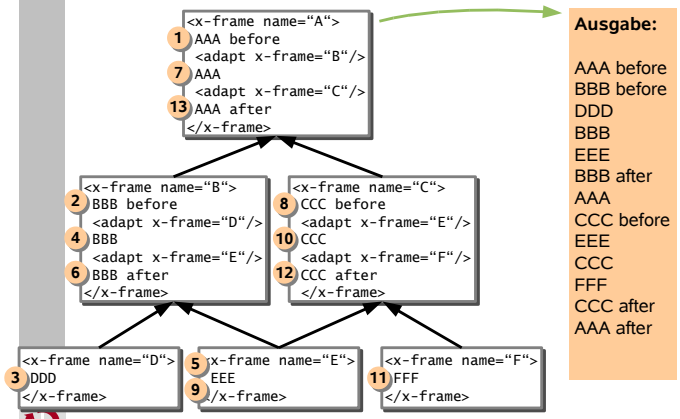
XVCL – Grundbegriffe



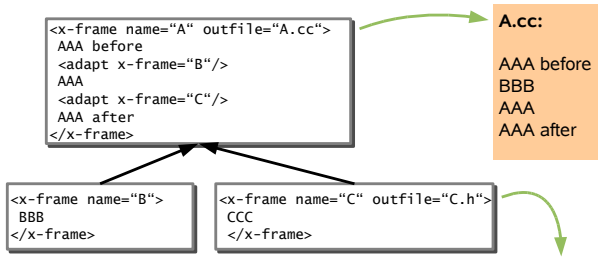
XVCL – Verarbeitung (1)



XVCL – Verarbeitung (2)

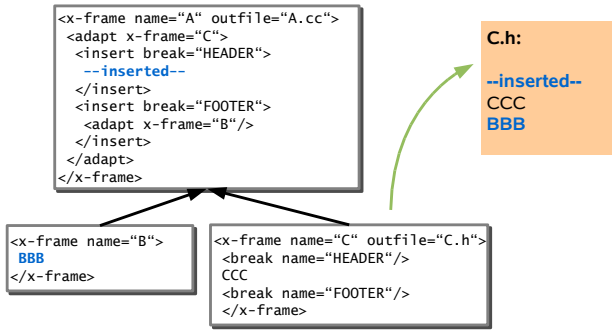


XVCL – Dateiselektion



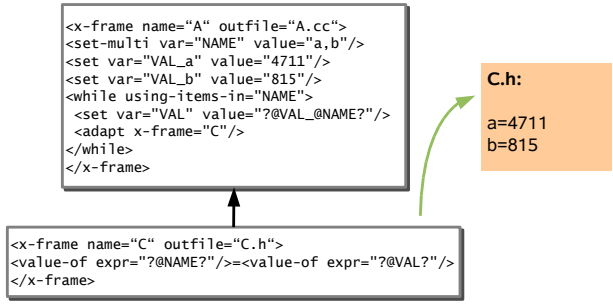
- normalerweise wirkt *adapt* wie #include
  - Komposition aus (angepassten) Codefragmenten
- in Verbindung mit *outfile* kann der SPC bestimmen, welche Dateien generiert werden

XVCL – Anpassung von *Frames* (1)



- durch *break* und *insert* (*-before*, *-after*) kann an vordefinierten Stellen durch übergeordnete *Frames* Text eingefügt werden

XVCL – Anpassung von *Frames* (2)



- Variablen, Schleifen und Bedingungen erlauben flexible Textgenerierung

XVCL – Beispiel I/O-Library (1)

```
<x-frame name="OStream">
#ifndef __OStream_h__
#define __OStream_h__

#include "OStreamDock.h"
<break name="STREAM_INCLUDE"/>

class OStream : public OStreamDock {
<break name="STREAM_ATTRIBUTE"/>
public:
  OStream&amp;amp; operator &amp;lt;&amp;lt; (char) {...}
  OStream&amp;amp; operator &amp;lt;&amp;lt; (OStream&amp;amp; (*f) (OStream&amp;amp;));
<break name="STREAM_OP"/>
};

OStream&amp;amp; endl (OStream&amp;amp; os) {...}
<break name="STREAM_MANIP"/>

#endif // __OStream_h__
</x-frame>
```

- die OStream Klasse enthält lediglich das Grundgerüst und Markierungen an potentiellen Variationspunkten

XVCL – Beispiel I/O-Library (2)

```
<x-frame name="IntegerOStream">
 <select option="SECTION">
  <option value="INCLUDE">
   #include "IntegerOutput.h"
  </option>
  <option value="ATTRIBUTE">
   IntegerOutput iout;
  </option>
  <option value="OP">
   OStream &amp;amp; operator &amp;lt;&amp;lt; (MaxInt i) { ... }
  </option>
 </select>
</x-frame>
```

- getrennte *x-frames* beschreiben die Erweiterungen für die verschiedenen unterstützen Datentypen (*slices*)

XVCL – Beispiel I/O-Library (2)

```
<x-frame name="IntegerOStream">
 <select option="SECTION">
  <option value="INCLUDE">
   #include "IntegerOutput.h"
  </option>
  <option
   Intege
  </optio
  <option
   OStrea
  </optio
 </select
</x-frame
```

```
<x-frame name="PointerOStream">
 <select option="SECTION">
  <option value="INCLUDE">
   #include "PointerOutput.h"
  </option>
  <option value="OP">
   OStream &amp;amp; operator &amp;lt;&amp;lt; (void *p)
{ ... }
  </option>
 </select>
</x-frame>
```

...

- getrennte x-frames beschreiben die Erweiterungen für die verschiedenen unterstützen Datentypen (*slices*)

XVCL – Beispiel I/O-Library (3)

```
<x-frame name="ConfigOStream" outfile="OStream.h">
 <adapt x-frame="OStream">
  <insert break="STREAM_INCLUDE">
   <set var="SECTION" value="INCLUDE"/>
   <while using-items-in="TYPES">
    <adapt x-frame="?@TYPES?"/>
   </while>
  </insert>
  <insert break="STREAM_ATTRIBUTE">
   <set var="SECTION" value="ATTRIBUTE"/>
   <while using-items-in="TYPES">
    <adapt x-frame="?@TYPES?"/>
   </while>
  </insert>
  ... weitere inserts für Operationen und Manipulatoren
 </adapt>
</x-frame>
```

- ein weiterer *x-frame* fügt die OStream Klasse entsprechend der konfigurierten Typen in TYPES zusammen

XVCL – Beispiel I/O-Library (4)

```
<x-frame name="Test1">
 <set-multi var="TYPES" value="IntegerOStream,PointerOStream"/>
 <adapt x-frame="ConfigOStream"/>
</x-frame>
```

- der SPC beschreibt eine konkrete Konfiguration in kompakter Weise

XVCL - Zusammenfassung

- **Pro**
  - Konfigurationswissen und Programmcode kann getrennt werden
  - selektives Generieren von Dateien möglich
  - Quellcodeorganisation kann unabhängig von Modularisierungs-techniken der Programmiersprache (z.B. Klassen) erfolgen
  - programmiersprachenunabhängiger Ansatz
- **Contra**
  - keine explizite Repräsentation des Problemraums
  - XML Syntax und *Escape*-Zeichen im Quelltext, z.B. &
  - fehlende Verbindung zwischen XVCL Quelltext und generiertem Quelltext problematisch bei Fehlermeldungen des Übersetzers

Ausblick

- Untersuchung verschiedener Techniken zur Umsetzung von Variabilität in der Implementierung der Komponenten
  - programmiersprachenbasierte Lösungen
    - Aspekte
    - Objekte
    - *Templates*
    - *Mixin Layers*

Literatur

[1] pure-systems GmbH. *Variantenmanagement mit pure-variants*. Technical White Paper, http://www.pure-systems.com/.

[2] pure-systems GmbH. *pure::variants Eclipse Plugin. pure-variants*. Manual, http://www.pure-systems.com/.

[3] National University of Singapore. *XML-based Variant Configuration Language (XVCL)*. 2004.