Konfigurierbare Systemsoftware
Zusammenfassung des Wichtigsten

Sommersemester 2016

Dr. Daniel Lohmann

Autor:

- Christian Strate

Inhaltsverzeichnis

0 Organisatorisches

- Prüfungsanmeldung bei SST, nur 7 Vorlesungstermine und 1-2 Übungsaufgaben
- Prüfung 30 min

Note:

Bei dieser Zusammenfassung handelt es sich um eine inoffizielle Mitschrift von Studenten. Entsprechend sind weder Garantie auf Vollständigkeit noch auf Korrektheit gewährleistet.

1 Einführung

Anforderungen:

- Variabilität
  Menge erfüllter funktionaler Anforderungen
- Granularität
  Zielsetzung welche funktionalen Anforderungen erfüllt und nicht überfüllt werden.
  ↦ Passgenauigkeit

libc verfügt inzwischen über so viele interne Abhängigkeiten, dass das Rauspicken der einzelner Funktionen, die auch tatsächlich im eigenen Code verwendet werden, nahezu unmöglich ist (Beispiel printf(), diese ist per default bereits im Startup-Code enthalten (Fehlerausgaben)). Die Threadsicherheit ist maßgeblich Schuld an dem massiven Blow-Up, da Exit-Protokolle Daten auf dem Heap anlegen und dessen Fehlerbehandlung beim Fail zu dem printf-foo führen.

Experiment 5 (listing 1) funktioniert inzwischen, sind allerdings kein Teil der Standardschnittstellenbeschreibung und ändern sich gelegentlich.

```
1 echo '_start(){write(1, "Hello World\\n", 13);_exit(0);}' | gcc
    -xc - -Os -w -static -nostartfiles -o hello
```

Listing 1: Hello world mit write und ohne libc-Wulst (Startup-Code)

```
1 echo '_start(){write(4, 1, "Hello World\\n", 13);_exit(0);}' |
    gcc -xc - -Os -w -static -nostartfiles -o hello
```

Listing 2: Hello world mit syscall und ohne libc-Wulst (Startup-Code)
Wirft bei mir lustigerweise einen Segfault

- Bei der libc wurde also Granularität gegen Variabilität eingetauscht.
- Speicher in CPUs sind extrem teuer und steigt schnell an. Auch RAM ist nur begrenzt verfügbar.
- Customizing/Tailoring als Lösung: Modifizierung bestehender Systeme-Software, um eigene Bedürfnisse zu erfüllen. Dies schreit nach Granularität und Variabilität
- Software-Struktur ↦ Trade-Off reuse ⇔ coupling
- Software-Interface ↦ printf als god-mode
- Language and tool chain ↦ Compiler/Linker arbeiten auf Symbol-Granularität oder Object-Files und können nicht effektiv toten Code eliminieren

- Modellierung von Lösungen und sogar Problemen, deren Abhängigkeiten spezifiziert abgebildet werden. ⇒ Aspect (vgl. fig. 1)

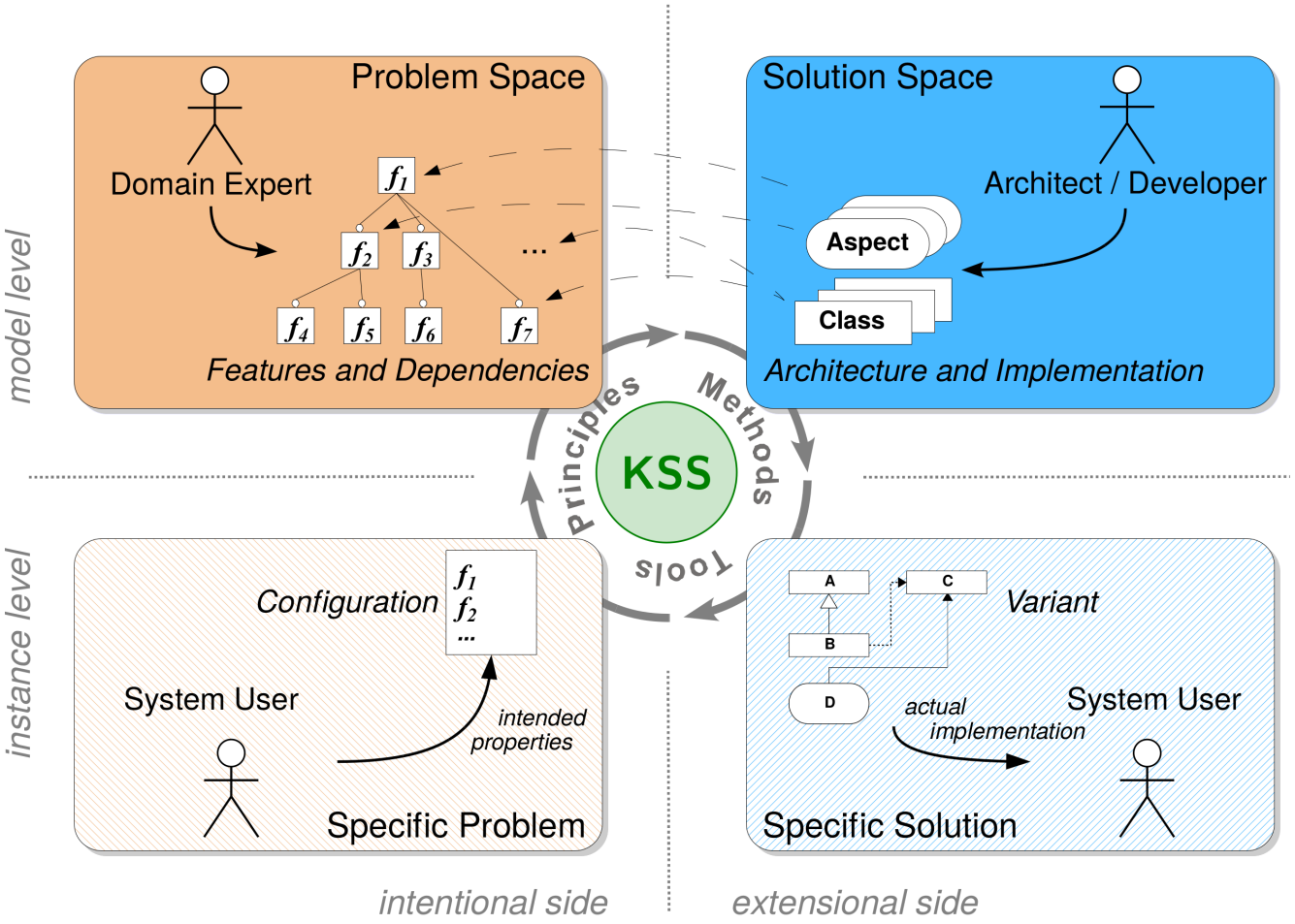


Abbildung 1: Product Line einer Konfigurierbare Software

2 Software Product Lines

Trend von Massenproduktion hin zur Spezialisierung dieser, aus einer Fülle von möglichen Features, gepaart mit automatisierter Produktion. Dadurch sind inzwischen unglaublich viele Features und Kombinationsmöglichkeiten vorhanden.

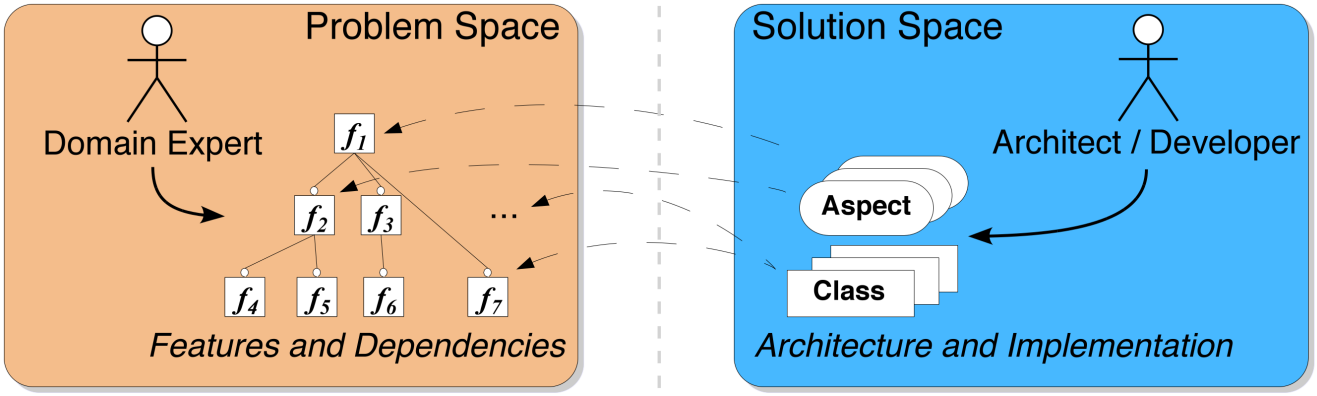


Abbildung 2: Herausforderungen einer Product Line

Identifizierung gewünschter Variabilität	Implementierung des variablen Codes
Darstellung, Spezifikation der gewünschten Variabilität	Verknüpfung der Variabilität mit dem Code

Product Line:

Eine *Gruppe von Produkten*, die eine gemeinsame Feature-Menge, die spezielle Bedürfnisse eines gewählten Sub-Marktes befriedigen, teilen.

Software Product Line (SPL):

Eine *Menge Software-intensiver Systeme*, die eine gemeinsam verwaltete Feature-Menge teilen, die die speziellen Bedürfnisse eines bestimmten Markt-Segments befriedigen. Die Motivation steckt in der Feature-Gleicheit bezüglich eines Markts.

Feature:

Unterscheidbare Konzept-Charakteristik, die für einige Stakeholder relevant ist.

Program Family:

Menge von Programmen, deren Gemeinsame Spezifikationen so immens sind, dass es ratsam ist die Gemeinsamkeiten zu betrachten, bevor die Unterschiede bestimmt werden.

- Programm Familie definiert durch die Gemeinsamkeiten von Programmen $\mapsto$ Lösungen

- SPL: definiert durch gemeinsame Anforderungen ↦ Probleme
- Programmfamilie wird verwendet um Produktlinien zu implementieren
- In aktueller Literatur wird Programmfamilie und Software Product Line synonym verwendet

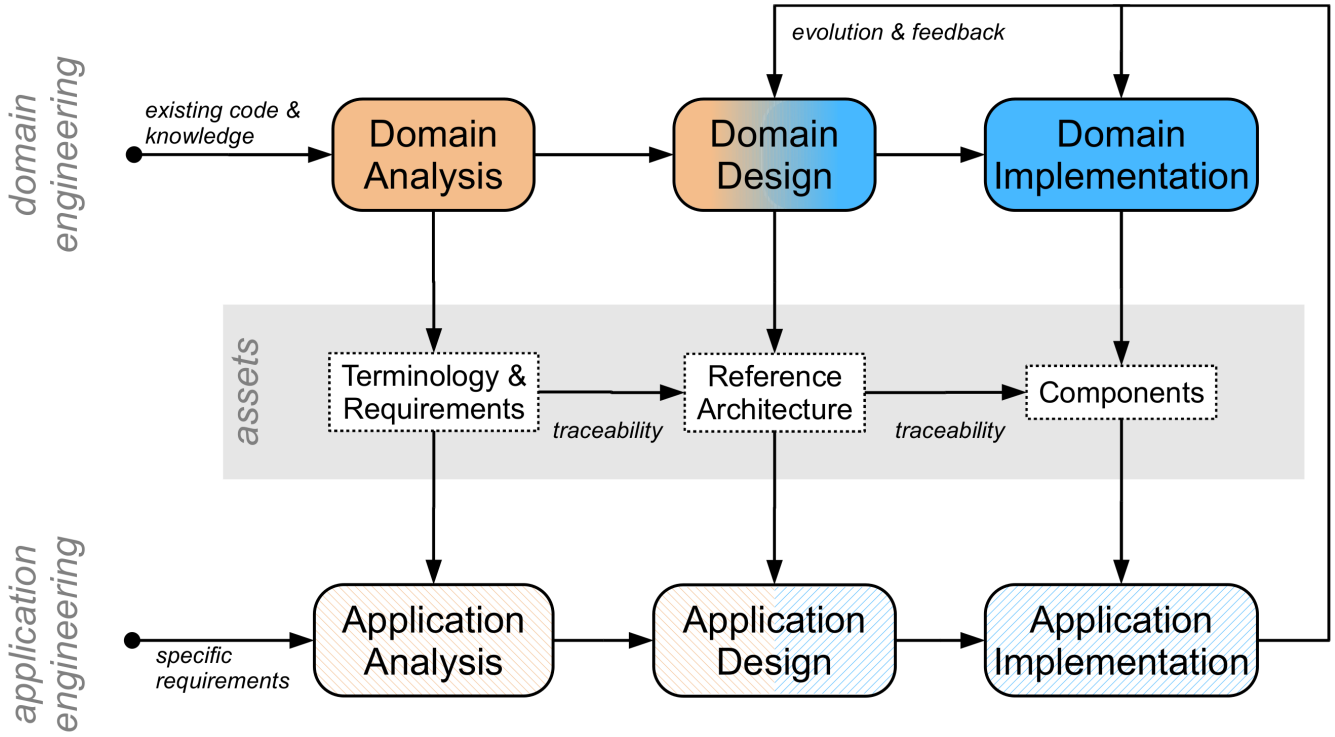


Abbildung 3: SPL: Development Reference Process
Probleme, Lösungen
erster linker Senkrechter Pfeil: Checkliste von Anforderungen - ein bereits bekannter Prozess
Mehrfachverwendung sinnvoll ⇒ Zurückführung aus Application Implementation nach oben

Entsprechend ist die Konfigurierbarkeit notwendig

Configurability:
Eigenschaft, die die vorab definierte Variabilität und Granularität durch System-Software mittels eines expliziten Konfigurations-Interface, angibt.

Keine Notwendigkeit Code anzufassen - Precompiler-Flags oder ein interaktiver Editor Ergebnis muss eine gültige Konfiguration sein. Deswegen wird der Zyklus auch nicht wie in dem Bild oben durchlaufen, sondern

Domain Model - 02-24ff:

Explizite Darstellung der geläufigen und variierenden Eigenschaften eines Systems in einer Domäne und die Abhängigkeiten unter den variablen Eigenschaften.

Elemente eines Domänen-Modells:

- Domänen Definition - Spezifikation des Scopes einer Domäne durch aufzählen von Regeln und Beispielen
- Domänen Glossar - Vokabeln und Namen von Features und Konzepten
- Konzept-Modelle - beschreibt relevante Konzepte einer Domäne (formell, textuell, Syntax, Semantik)
- Feature-Modelle - beschreiben die geläufigen und variierenden Eigenschaften von Domänen-Mitgliedern (textuell, Feature Diagramme - fig. 4)

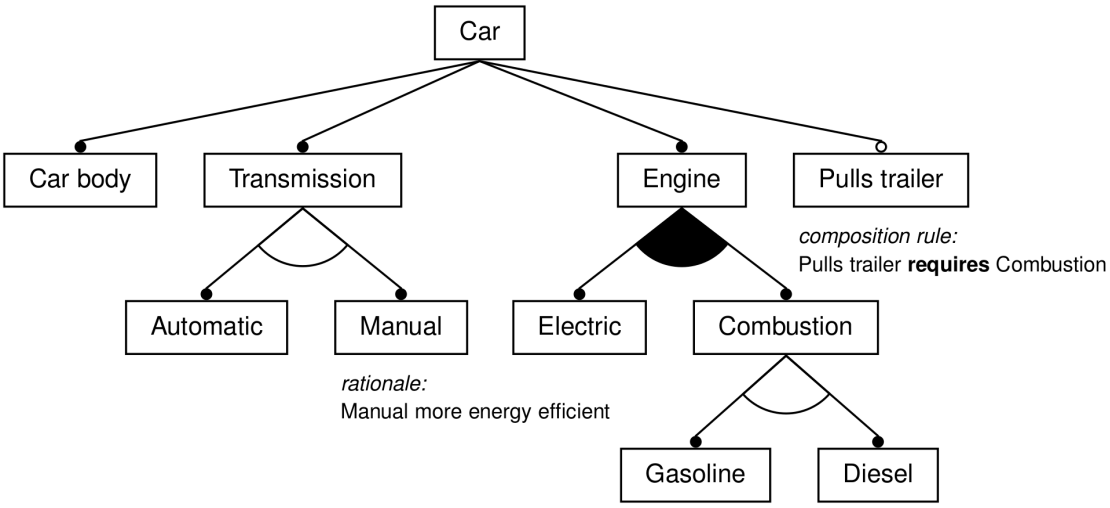


Abbildung 4: Feature Diagramm - Beispiel
Elementarer Bestandteil des Feature Modells, beinhaltet Konzepte, Features und Abhängigkeiten - Legende: fig. 5

Verschiedene Implementierungstechniken:

- Decompositional Approach
  - Unabhängigkeit von der Sprache (Preprocessor) ⇒ Flexibilität
  - Text-basierte Filterung (ungetypt)

  - Bedingte Compilation unter Verwendung des C-Präprozessors ist ein gängiger Ansatz, denn die Verwendung ist trivial und verursacht keinerlei Laufzeit-Overhead.
  - fig. 6

- Compositional Approach
  - durch die Schnittstellen ist die Compilierbarkeit eher gewährleistet, als beim Decompositional Approach
  - OOP, AOP, Templates - Sprach-basierte Compositionsmechanismus (getypt)
  - fig. 7

- Generative Approache
  - Generator generiert aus dem Domänenspezifischen Problem den entsprechenden Code. Flexibilität durch das spezifische Wissen im selbst gebauten Generator.
  - Metamodel-basierte Erzeugung von Komponenten (getypt)
  - MDD, C++ TMP, generators
  - fig. 8

Ziele der Implementierungs-Techniken:

- General
  - Separation of concerns (SoC)
  - Resourcen Sparsamkeit

- Operational
  - Granularität
    * Komponenten sollten feingranular gehalten sein.
    * jeder Bestandteil sollte notwendig oder aber nur einem einzelnen Feature zugeordnet sein
  - Economy
    * Der Zugriff auf Speicher und die Verwendung von Laufzeit-Teuren Sprach-Features sollte soweit wie nur möglich vermieden werden.
    * Soviel wie möglich zur Generationszeit linken.
  - Pluggability
    * Die Anpassung optionaler Features sollte nicht die Modifikation anderer Implementierungs-Teile erfordern
    * Feature-Werkzeuge sollten in der Lage sein sich selbst zu integrieren

  - Extensibility
    - Gleiches sollte für optionale Features gelten, die womöglich erst in zukünftigen Versionen der Product line verfügbar sind.

Evaluation Decompositional Approach - CPP:

- General
  - Separation of concerns (SoC) X
  - Resourcen Sparsamkeit ✓
- Operational
  - Granularität (✓)
    - Komponenten implementieren lediglich eine einzelne Funktionalität eines Features, enthalten jedoch Eingliederungs-Code für weitere optionale Features
  - Economy ✓
    - Sämtliche Features werden zur Compilezeit gebunden
  - Pluggability X
    - Hauptprogramm wird von integrierten Modulen beeinflusst, selbiges gilt für die Implementierung des Auslösers
  - Extensibility X
    - Weitere Auslöser und Sensoren erfordern die Erweiterung des Hauptprogramms

Evaluation Compositional Approach - OOP:

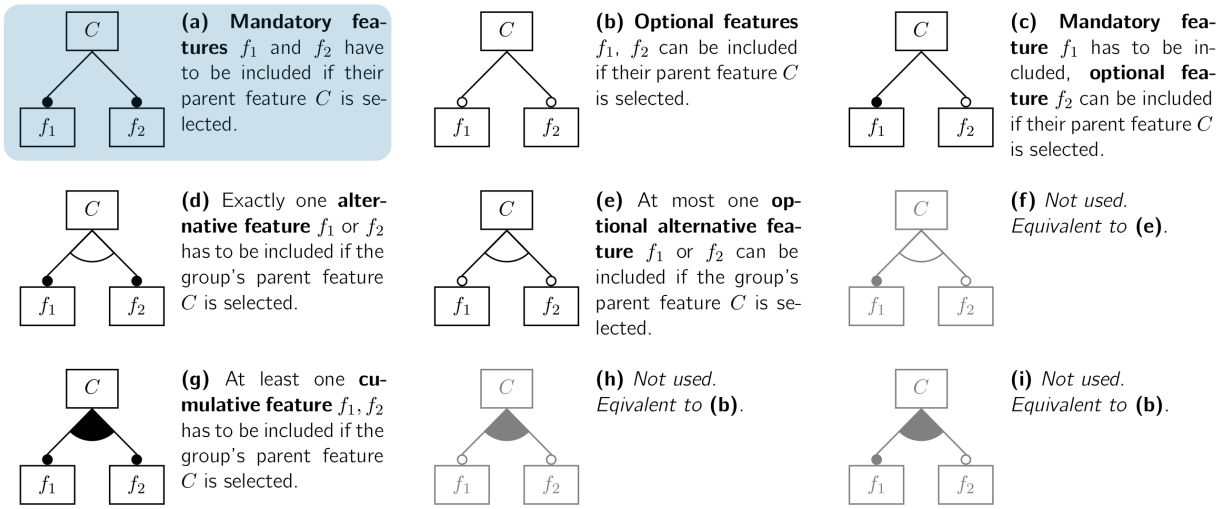
- General
  - Separation of concerns (SoC) ✓
  - Resourcen Sparsamkeit ?
- Operational
  - Granularität ✓
    - Jede Komponente ist entweder eine Basis-Klasse, oder implementiert lediglich eine einzelne Funktionalität eines Features
  - Economy (✓)
    - Eine Bindung zur Laufzeit und Laufzeit-Typ-Informationen nur dort verwendet, wo dies notwendig ist, um SoC zu erreichen
  - Pluggability ✓

    * Sensoren und Auslöser integrieren sich selbst als Design Patterns und globaler Instanzen
  - Extensibility ✓
    * Implementierung von Plug and Play von Sensoren und Auslösern

Analyse der Evaluation:

OOP - unglaublich teuer hinsichtlich Speicher weil v-table und Klassen-Deskriptoren. Aspekt-Orientierte Version hingegen ist nur minimal teurer, als die auf nativem C aufbauende (s. 02-42f). Für jede globale Variable wird eine Funktion erstellt, die den zugehörigen Konstruktor/Destruktor aufruft. Eine weitere Funktion die diese Funktionen aufruft. Dadurch ist hier keine Dead-Code-Eliminiation möglich, da Verweise auf diese Funktion existieren/erstellt wurden. Virtuelle Funktionsaufrufe können nicht inlined werden.

CPP erfordert also geringere Hardware-Kosten, liefert jedoch keine SoC, wohingegen OOP diese zu lasten teurer Hardware-Anforderungen liefern kann.



- die ausgefüllten Kreise signalisieren ein verbindliches Feature:
  a) $\mathcal{V} = \{(C, f_1, f_2)\}$
- die holen Kreise signalisieren ein optionales Feature:
  b) $\mathcal{V} = \{(C), (C, f_1), (C, f_2), (C, f_1, f_2)\}$
- diese können Kombiniert werden:
  c) $\mathcal{V} = \{(C, f_1), (C, f_1, f_2)\}$
- der hole Bogen signalisiert eine Gruppe von Alternativen Features (höchstens):
  d) $\mathcal{V} = \{(C, f_1), (C, f_2)\}$
  e) $\mathcal{V} = \{(C), (C, f_1), (C, f_2)\}$
- der ausgefüllte Bogen signalisiert eine Gruppe von zusammengehörigen Features (mindestens):
  g) $\mathcal{V} = \{(C, f_1), (C, f_2), (C, f_1, f_2)\}$

Abbildung 5: Feature Diagramm - Sprachbeschreibung

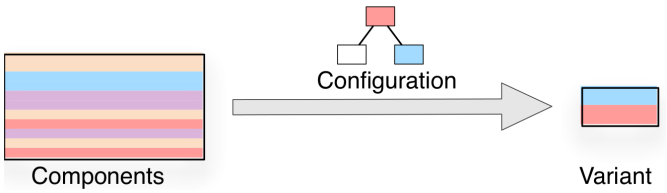


Abbildung 6: Decompositional approaches

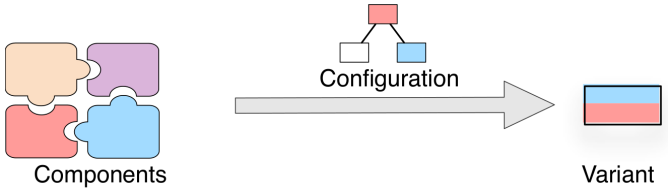


Abbildung 7: Compositional approaches

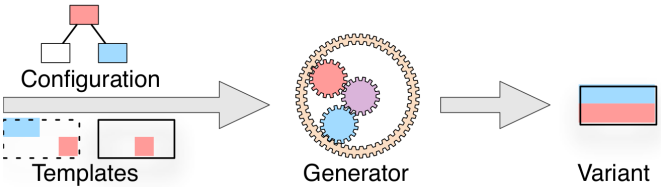


Abbildung 8: Generative approaches

3 Aspect-Oriented-Programming (AOP)

Idee:

Es soll der modulare Code von den einzufügenden Orten getrennt werden. Der AspectC++-Code wird vom ac++-Weber in reguläres C++ übersetzt, indem die Regeln angewendet werden und vor/nach bestimmten Instruktionen weitere hinzugefügt werden. - Beispiele 3-12, 3-27f

- Join Points ↦ wo
  - Stellen, innerhalb der statischen Strukturen oder dynamischen Kontrollflüssen
  - Erklärend durch Pointcut-Ausdrücke
- Advice ↦ was
  - zusätzliche Elemente (Mitglieder, ...), um einen Join Point einzuführen
  - zusätzliches Verhalten (Code) das an den Join Points eingeblendet wird

Probleme ohne AOP:

- Überschwemmung des tatsächlichen Codes mit Features, die sich über den gesamten Code verteilen
- Entsprechend ist der Code schwer zu lesen und schreiben, da vieles simultan geschieht
- Die Konfigurierbarkeit zur Compilezeit ist u.U. schwierig

Syntax:

- Aspekte werden eingeleitet vom Keyword *aspect* und dem Aspekt-Namen. Dieser Aspekt verhält sich syntaktisch wie eine Klasse (Klassenmember, Konstruktoren, ...), auf die Member können innerhalb der advices zugegriffen werden. Aspekte werden auch später bei der Codeumwandlung in Klassen umgewandelt.
- Es werden Regeln eingeführt, nach/vor diesen Instruktionen werden jene Instruktionen eingewoben. - listing 3
- Man kann zu den aufrufenden Instanzen eine Variable binden, die in diesem Fall vom Typ util:Queue sein soll/muss: - listing 4
- Advices sind anfügbar an: Funktionen, Methoden, Konstruktoren, Klassen, Structs

```
1 advice execution(<Instruktion nach der eingefügt werden soll
      als Pattern>) : after(){
2     <einzuwebender Code>
3 }
```

Listing 3: Einfügen nach Code - Regel, nach welchem Code welcher Code einzuweben ist

```
1 advice execution(<Instruktion nach der eingefügt werden soll
      als Pattern>) && that(queue) : after(util::Queue& queue){
2     <einzuwebender Code>
3 }
```

Listing 4: Variablen binden und Typ Checking - die Aufrufende Instanz (Variable) wird gebunden (*that* an *queue*, wobei *that* die Kontextvariable ist), die dann innerhalb des einzuwebenden Codes verwendet werden kann. Der Typ der Variable wird auf *Queue&* spezifiziert.

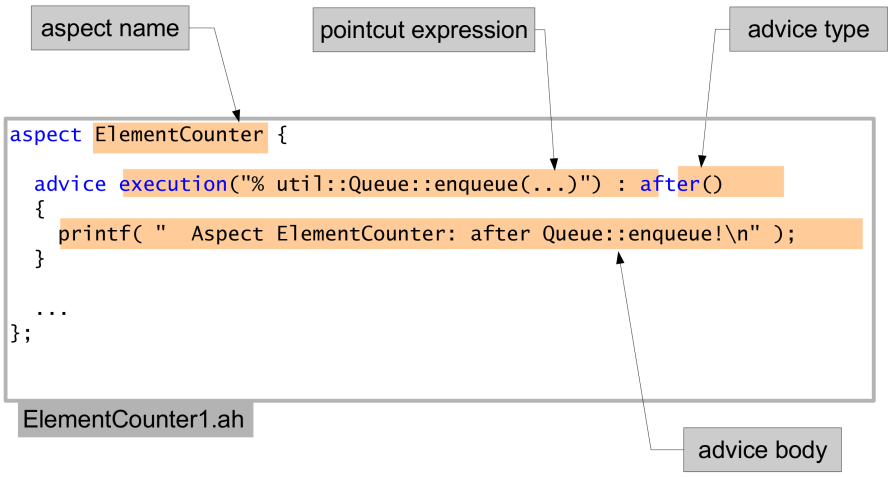


Abbildung 9: Verschiedene Sytax-Elemente

Joinpoint:

Markiert eine Stelle, um Anweisungen zu geben. Sie werden von Pointcut-Ausdrücken vorgegeben, die wiederum eine Menge von Joinpoints beschreiben.

- Code - Ein Punkt im Kontrollfluss
- Name - eine benannte C++-Programm-Entität (Identifikator), also eine Klasse, eine Funktion, eine Methode, ein Typ, ein Namespace

Pointcut Expressions:

Pointcut Funktionen können mithilfe der Logischen Operatoren &&, bzw. || zu einer Pointcut Expression kombiniert werden. Expressions werden erzeugt aus

- match expression
  - Unterstützt Wildcards
  - Wird auf C++-Programm-Entitäten ge*match*t, die zu Joinpoints werden
  - Beispiel "% util::queue::enqueue(...)"
- pointcut functions $\mapsto$ code Joinpoints
  - Ausführung - Alle Knoten des Kontrollflusses, in denen eine Funktion ausgeführt wird
  - Aufruf - Alle Knoten des Kontrollflusses, in denen eine Funktion aufgerufen wird
  - Beispiele: execution(...), call(...), that(...)

within-keyword:

within("% main(...)") entspricht
Alle Aufrufe, die statisch bestimmbar, innerhalb des Namespaces, von der main aufgerufen werden. Dies schließt nicht zwingend unerreichbare aus. Falls diese ausgeschlossen werden sollen, so kann man *cflow* verwenden, um sicherzustellen, dass der Kontrollfluss berücksichtigt wird. Hierbei wird der Call-Stack zur Laufzeit betrachtet.

args-keyword:

Bindet eine Kontextvariable an das erste Argument des Typs:
advice execution(<original code>) && args(<kontextvar>) : before(<Type>)

result-keyword:

Bindet eine Kontextvariable an das Ergebnis des angegebenen Typs:
advice execution(<original code>) && result(<kontextvar>) : after(<Type>)

advices:

- before
  lesen/modifizieren von Parameter vor dem originalen Code
- after
  lesen/modifizieren von Rückgabewerten nach dem originalen Code
- around
  ersetzen von ursprünglichem Code, der originale Code kann jedoch explizit aufgerufen werden mit $tjp->proceed()$

```
1  //execution joinpoints
2  class ClassA{
3  public:
4     void foo(){
5        //advice execution("void classA::foo()") : before()
6        ...
7        //advice execution("void classA::foo()") : after()
8     }
9  };
10
11 //call joinpoints
12 int main(){
13    ...
14    Class A a;
15    //advice call("void classA::foo()") : before()
16    a.foo();
17    //advice call("void classA::foo()") : after()
18    ...
19 }
```

Listing 5: Before/After Advice mit Execution und Call Joinpoints - Beachte, dass das selbe Pattern je nach Joinpoint auf etwas anderes matcht

Aspect-Eigenschaften:

- Aspecte sind friend
- Aspecte können von anderen Aspecten erben, wobei Methoden und Pointcuts überschreibbar sind
- advice() sind z.Z. nicht überschreibbar, da kein Name
- Pointcuts können pure virtual sein, der zugehörige Aspect ist dann ein abstract aspect
- Aspect-Definition kann keine Templates entgegen nehmen, d.h. Funktionen müssen pure virtual sein, diese können aber zur Compile-Zeit aufgelöst werden, sonst baut es gar nicht erst. D.h. es ist nicht arschlahm.
- order-advice kann für eine Pointcut-expression die einzelnen Aspekte ordnen. Per default erzeugt der Weber eine Ordnung, die beliebig aber fest ist.
- Auflösung zweier sich gegenseitig beeinflussender Aspekte ist ein neuer Aspekt, der die Auflösung vornimmt

In diesem Foliensatz befinden sich noch viele weitere Beispiele und fortgeschrittene Keywords, die ich hier nicht mit aufgenommen habe, da ich es für sinnvoller halte die Folien durchzugehen

4 Aspect-Aware Design, CiAO

Struktur AOP vs. OOP:

Bei einem OOP-Ansatz, muss die main die aufzurufenden Funktionalitäten kennen, um diese aufrufen zu können. Bei AOP hingegen geben die Aspekte womöglich Anweisungen an die main, entsprechend kann der Aspect die main kennen, muss es aber nicht. Der Kontrollfluss wird also entgegengesetzt der Direktion umgesetzt. ("I make you call me"). Die Aspecte können jedoch auch ohne Wissen über das umgebene Programm verwendet werden, indem Pointcuts verwendet werden, da die Expressions auf die Joinpoints matchen.

Ziele des CiAO (CiAO is Aspect-Oriented):

CiAO verfolgt eingebettete Systeme mit gezielter Anwendung von Aspekten.

- architektonische Konfigurierbarkeit (Synchronisation, Schutz, Interaktion)
- Effizienz, allgemeine Konfigurierbarkeit, Portabilität
- Strikte Entkoplung der Strategie von den Implementierungs-Mechanismen

Richtlinen des Aspect-gewahren Entwicklung:

- *Design Principles* $\mapsto$ *Development Idioms*
- freie Kopplung durch anweisungsbasierte Bindung
  - Aspekte sollen sich in alle statischen und dynamischen Systemkomponenten einklinken können.
  - Aspekte sollen die Komponentenbindung und deren Instantiierung, sowie Zeitpunkt und Reihenfolge der Initialisierungen übernehmen.
- Sichtbare Transitionen durch explizite Joinpoints
  - Aspekte sollen sich in alle möglichen Kontrollflüsse einklinken können.
  - Sämtliche Transitionen in das System hinein, aus diesem heraus und innerhalb des Systems sollen von Aspecten beeinflussbar sein.
  - Dafür müssen die Transitionen auf dem Joinpoint-Level als statisch auswertbar und eindeutige Joinpoints dargestellt werden.
- minimale Erweiterung durch Erweiterungs-Scheiben
  - Aspecte sollen sämtliche Features, die vom System bereitgestellt werden, auf feiner Granularitätsstufe erweitern können.
  - System-Komponenten und -Abstraktionen sollten feingranular, wenige und durch Aspekte erweiterbar sein.

Aspects and Classes:

Eine Sache wird nur dann als Klasse und nicht als Aspect abgebildet, wenn es ein erkennbares instanziierbares Konzept von CiAO ist (sonst ist es ein Aspect). Also wenn es:

- Eine System-Komponente ist
  - Intern zu Gunsten von CiAO instantiiert (Scheduler, Alarm Manager, Verwaltung des Kernel-Zustands, ...)
- Eine System-Abstraktion ist
  - Zugunsten des Users als Objekt instantiiert (Task, Event, Ressource, ...)

Rollen von Aspects and Classes:

drei übliche Aspect-Rollen

- Extension aspect
  Erweitert einige System-Komponenten oder System-Abstraktion um weitere Funktionalitäten
- Policy aspect
  klebt System-Abstraktion oder Komponenten zusammen, um einige CiAO Kernel-Policy zu implementieren
- Upcall aspect
  Binde von höheren Schichten definierte Verhalten an Events, die von niedrigeren Schichten des Systems erzeugt werden

Bindung von Joinpoints:

- Advice-basierte Bindung (Verfügbarkeit der korrekten Joinpoints)
  - statisch evaluierbar für sämtliche semantisch relevanten System-Transitionen
- Feingranulare Komponenten-Struktur (Viele Implizite Joinpoints)
  - Anzahl und Präzision der Semantik ist oft Implementierungsabhängig
  - Aspects benötigen Wissen
- Wichte Transitionen aus technischen Gründen nicht als Joinpoints verfügbar
  - Zielcode kann instabil sein (Kontextwechsel), wodurch keine Beratung erforderlich ist
  - Zielcode kann in Assembly geschrieben werden, wodurch Transitionen nicht als Joinpoints sichtbar sind

Lösungsansatz Bindung von Joinpoints - Explizite Joinpoints:

- Leere Inline-Methoden, die einzig die Funktionalität haben, dass Aspects gegen diese binden können
- Explizit durch Komponenten oder andere Aspects ausgelöst
- Upcall Joinpoints
  - Repräsentieren System-interne Events, die auf einer höheren Ebene zu bearbeiten sind
  - Exceptions, Signale, Interrupts
  - Interne Events, System-Initialisierung, Betreten eines Idle-Zustands
- Transitions Joinpoints
  - Repräsentieren wichtige Kontrollfluss-Transitionen innerhalb des Kernels
  - Level-Transitionen: $user \rightarrow kernel, user \rightarrow interrupt$
  - Kontext-Transition: $threadA \rightarrow threadB$

5 Generative Programming, Sloth

5.1 OSEK

OSEK OS:

Komplett statisch konfigurierte Betriebssystemfamilie: OSEK OS, OSEKtime, AUTOSAR OS

Verschiedene Kontrollflüsse:

- Task
  Software-Triggered Kontrollfluss (statisch prioritätsbasiertes Scheduling)
  - Basic Task (BT)
    run-to-completion Task mit strikter stackbasierter Aktivierung und Beendigung
  - Extended Task (ET)
    Kann Ausführung suspenden und wiederaufnehmen (Koroutinen)
- ISR
  Hardware-Triggered Kontrollfluss (Hardware-spezifiziertes Scheduling)
  - Cat 1 ISR (ISR1)
    Läuft unterhalb des Kernels und darf keine System-Services aufrufen (Prolog, ohne Epilog)
  - Cat 2 ISR (ISR2)
    Mit dem Kernel synchronisiert, darf System-Services aufrufen (Epilog, ohne Prolog)
- Hook
  OS-Triggered Signal/Exception Handler
  - ErrorHook - Im Fall eines Syscall-Fehlers aufgerufen
  - StartupHook - zur System-Boot-Zeit aufgerufen
  - ...

Koordination und Synchronisation:

- Ressource
  Gegenseitiger Ausschluss zwischen wohldefinierten Taskmengen
  - Stackbasiertes priority ceiling protocol
  - Verdrängung wird blockiert, da der vordefinierte RES_SCHED die höchste Priorität hat.
  - Taskmenge enthält möglicherweise ISR2

- Event
  Condition Variable, auf die ETs möglicherweise blockieren
  - Teil des Task-Kontextes
- Alarm
  Asynchroner Trigger durch einen HW/SW Counter
  - führt möglicherweise einen Callback durch, aktiviert einen Task oder sendet ein Event bei Fristversäumung

Vordefinierte Maßschneiderbarkeit durch folgende Konformitätsklassen:

- BCC1
  - Nur Basic-Tasks
  - Begrenzt auf eine Aktivierungsanfrage pro Task
  - Jeder Priorität ist lediglich ein Task zugeordnet
  - Entsprechend haben verschiedene Tasks auch verschiedene Prioritäten
- BCC2
  - Wie BCC1, aber zusätzlich:
  - Können einzelnen Prioritäten nun mehrere Tasks zugeordnet sein
  - Mehrere Anfragen der Task-Aktivierung sind nun erlaubt
- ECC1
  - Wie BCC1, aber zusätzlich:
  - Erweiterte Tasks
- ECC2
  - Wie BCC2, aber:
  - Es gibt erweiterte Tasks und
  - Mehrere Anfragen der Task-Aktivierung sind weiterhin nur den Basic-Tasks erlaubt

Verhalten von Basic-Tasks:

Basic-Tasks verhalten sich ähnlich wie IRQ-Handler auf einem System mit verschiedenen IRQ-Prioritätsebenen. Das Dispatching erfolgt prioritätsbasiert und unter run-to-completion-Semantik. Da die Ausführung LIFO ist, können sämtliche Kontrollflüsse auf einem einzelnen, geteilten Stack ausgeführt werden. Die Idee ist nun also die Tasks als ISRs zu dispatchen und die Hardware das Scheduling übernehmen zu lassen (SLOTH)

5.2 SLOTH

Idee:

Die Idee ist Threads als Interrupt Handler zu betrachten, die synchrone Thread-Aktivierung ist IRQ. Dadurch kann das Interrupt-Subsystem das Scheduling und Dispatching übernehmen. Das ist auch auf Prioritätsbasierte Echtzeitsysteme anwendbar. Die Vorteile sind: Es ist ein kleines System, mit schnellem Kernel und einer vereinheitlichten Kontrollfluss-Abstraktion. Es gibt keine Readliste, sondern alles läuft über Interrupts. Und Sloth ist tatsächlich wesentlich schneller als OSEK. Die Benchmark Ergebnisse auf 5-20 sind durchaus beachtlich.

Sloth-Eigenschaften:

- Kernel-Implementierung in weniger als 200 LoC und weniger als 700 Bytes
- eine Kontrollfluss-Abstraktion für Tasks ISR1/ISR2, Callbacks
- Es ist irrelevant wie etwas ausgelöst wurde, das wird sich entsprechend auch nicht gemerkt
- Vereinheitlichter Prioritätsraum für Tasks und ISRs, keine Ratenmonotone Prioritätsumkehr
- Straight-Forward Synchronisation durch die Veränderung der CPU-Prioritäten
  - Ressourcen und ISRs verfügen über Ceiling Priority
  - Nichtverdrängende Abschnitte mit RES_SCHEDULER, der über die höchste Task-Priorität verfügt
  - Kernel-Synchronisation mit höchster Task-Priorität

Limitierende Eigenschaften:

- Jeder Priorität kann nur ein einzelner Task zugeordnet werden (Ausführungsreihenfolge muss identisch zu der Aktivierungsreihenfolge sein)
- Keine erweiterten Tasks, das ist mit einem Stackbasierten IRQ Ausführungsmodell nicht möglich
- Keine Sicherheit (Memory-Protection), ist nicht möglich, wenn alles als IRQ-Handler läuft

5.3 Sleepy SLOTH

Motivation und Herausvorderung:

Es sollen erweiterte blockierende Tasks mit eigenen Stacks unterstützt werden, ohne dabei die Vorteile von SLOTH einzubüßen, indem Threads als ISRs laufen.

	Activation Event	Sched./Disp.	Semantics
ISRs	HW	by HW	RTC
Threads	SW	by OS	Blocking
SLOTH	HW or SW	by HW	RTC
Sleepy SLOTH	HW or SW	by HW	RTC or Blocking

Tabelle 1: Kontrollflüsse in Embedded Systems

Die Herausforderung besteht in dem mangelnden Support der IRQ-Controller bzgl. der Suspendierung und Reaktivierung von ISRs.

Dispatchung und Rescheduling:

Extended Task $\mapsto$ verdrängbar $\mapsto$ eigener Stack. Die Sicherung des Kontextes des Verdrängten muss durch denjenigen erfolgen, der den Prozessor erhält. Setzen der CPU auf Priorität 0 gibt einem die Info welcher Task der höchstpriore ist und an den kann man dann die Ausführungsrechte überreichen. Denn der einst verdrängte muss nicht der moment höchstpriore lauffähige Task sein.

- Task-Prolog: Tauscht den Stack, sofern notwendig
  - Ein Basic-Task unterlässt den Stackwechsel
  - Beim Starten des Jobs wird der Stack initialisiert
  - Beim Fortsetzen des Jobs wird der Stack wiederhergestellt
- Task-Beendigung: Task mit der nächsthöheren Priorität muss laufen
  - gibt die CPU ab, indem die Priorität auf 0 gesetzt wird
  - Der Stackwechsel wird vom Prolog des nächsten Tasks durchgeführt
- Task-Blocking: Entnehme einen Task aus der Readyliste
  - Die IRQ-Quelle des Tasks wird deaktiviert
  - gibt die CPU ab, indem die Priorität auf 0 gesetzt wird
- Task-Unblocking: Setze einen Task auf die Readyliste
  - Reaktivierung der IRQ-Quelle eines Tasks
  - Neuauslösung der IRQ-Quelle eines Tasks, indem das Pending-Bit gesetzt wird

5.4 Safer SLOTH

Motivation und Herausforderung:

Zustände und Privilegien sollen isoliert werden, ohne dabei die Vorteile von SLOTH einzubüßen, indem Threads als ISRs laufen.

Die Herausforderung besteht darin ISRs im Supervisor-Modus mit vollen Privilegien laufen zu lassen. Also müssen Kernel und Anwendung voneinander getrennt werden und einige Design-Prinzipien von SLOTH überarbeitet werden.

Ziel ist Safety, nicht Security. Geschützt werden Daten, kein Code. Vertikal: Schutz von Kernel-Zustand und MPU-Konfiguration. Horizontal: Anwendungen oder Tasks voneinander Isolieren. Es wird also vor dummen Programmieren, nicht vor bösartigen Angreifern geschützt.

Protection Modes - Laufzeitvergleich 05-40f:

- unsafe
  - Orignales Sloth OS, ohne Isolation
- MPU
  - aktive MPU, Tasks werden mit Supervisor-Privilegien ausgeführt
  - Vertikale Isolation ist in Post-Validation zugesichert
- MPU+traps
  - Vertikale Isolation wird durch Hardware-Privilegien-Ebenen zugesichert
  - System-Services fordern Kernel-Privilegien an, indem sie Syscall-Mechanismen verwenden

Traps als Problem:

Sloth wird durch Compiler-Optimierungen extrem begünstigt

- System-Service-Calls können inlined werden
- Dead-Code-Elimination
- Constant-Propagation
- Traps verbieten solche Optimierungen ein Stück weit (Umsortieren von Code, der möglicherweise zu Traps führt ist häufig nicht erlaubt)

Die Lösung ist eine Kombination der Modes MPU und MPU+traps, also ein neuer Protection Mode (MPU+itraps), indem Traps inlined werden.

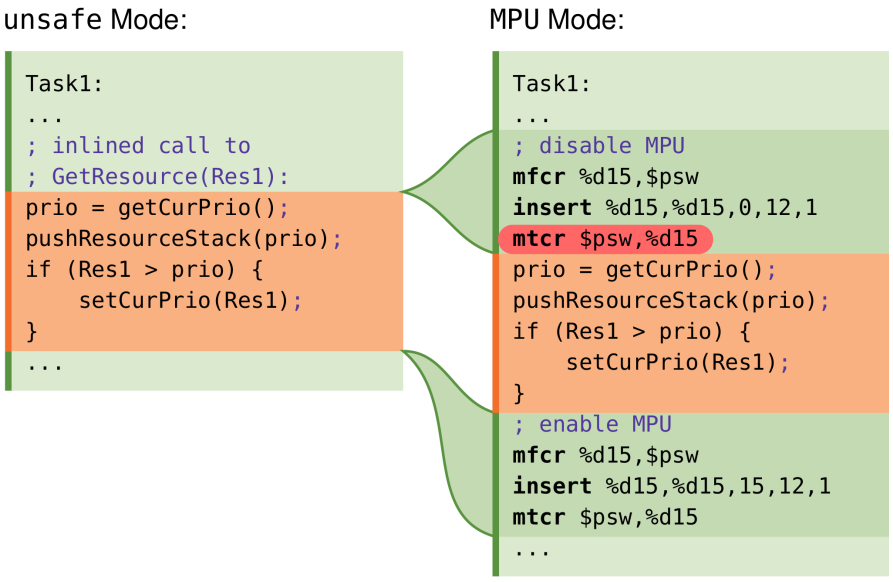


Abbildung 10: MPU mode in Safer Sloth

User mode, Supervisor mode, System service implementation

Trap in den Kern nicht inlinebar, deswegen wird das in Sloth mit einer vierten Implementierung realisiert. Hier wird die Anwendung im User und nicht mehr Supervisor modus ausgeführt. Springt also nach dem Trap direkt zurück, ohne den Modus (in grün und zurück) zu wechseln, sondern fährt direkt im Kernel-modus weiter. Also lässt sich das alles inlinen und vom Compiler optimieren.

5.5 SLOTH on time

Idee:

- Arrays von Hardware Timern werden verwendet, um Scheduling Tables zu implementieren
- Sehr interessant: Prioritätsumkehr ist im SLOTH on Time ist nicht gegeben, weil die Tasks alle im selben Adressraum laufen und RES_SCHEDULER die höchste Priorität hat. Deswegen ist Prioritätsumkehr per Design nicht möglich.

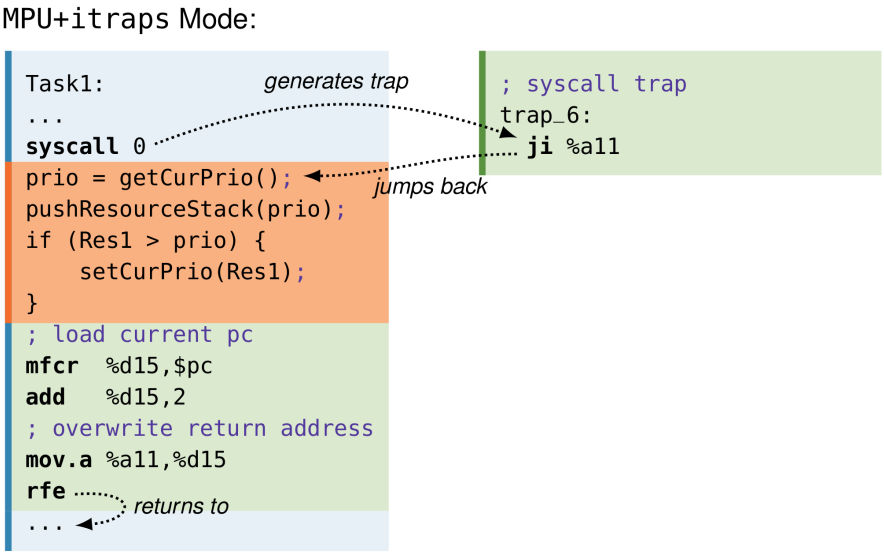


Abbildung 11: Inline Traps in Safer Sloth
User mode, Supervisor mode, System service implementation

6 Variability in the Large, VAMOS

Ecos verfügt über einen sehr platten Konfigurationsbaum, das lässt sich für Dinge ausnutzen und muss es auch. (06-06)

Linux Configuration und Generation Process:

Es erfolgt eine Auflösung von Konfigurationen durch Variabilitätsauswahl und Variablitäts-Termination bei jeder Ebene bis zum letztendlichen Source-Code.

1. Konfiguration mithilfe eines KCONFIG-Frontends, z.B. visuell aus einem Feature Diagram
2. Compilation einer Datei-Teilmenge
3. Auswahl einer Untermenge der darin enthaltenen CPP-Blöcke
4. Linking des Kernels und der ladbaren Kernel-Module

- KBUILD $\mapsto$ Grobgranulare Konfiguration (Zentrale Proklamation der Konfigurierbarkeit)
- CPP, GCC, LD, MAKE $\mapsto$ Feingranulare Konfiguration (Verteilte Implementierung der Konfigurierbarkeit)
- Inzwischen wird versucht diese Konfiguration von Preprocessor zu gcc-language level ausgelagert. Der Vorteil ist hier findet auch Typprüfung statt.

Konfigurationskonsistenz als Herausforderung:

Neben den symbolische Constraints, also den ausgewählten Konfiguration, bestehen außerdem logische Constraints, also Abhängigkeiten von Defines. Dieser Code bedingt diesen und benötigt wiederum jenen, etc.. Unterschiede können in der Implementierung bestehen. Das resultiert in untote und tote Blöcke (also nur scheinbar konfigurierbare Blöcke, die entweder immer oder aber nie ausgewählt werden können. Solche (un)tote Blöcke sind ein Hinweis auf Fehler in Konfigurationen, also ein Konfigurationsdefekt. Also z.B. eine fehlerhafte Bezeichnung bei Präprozessor-Variablen.

Konfigurationsdefekt:

Existenz von (un)toten Blöcken. Dies ist nicht zwingend ein falsches Verhalten, auch nicht zwingend ein Bug. Womöglich ist Code ganz bewusster tot.

Lösung: Undertaker soll toten ifdef-code finden

Struktur der Variatbilität:

- ein Großteil der ifdefs ist Options-Abhängig und nicht Werte-Abhängig. (ca 93%)
- Häufig wird nur der x86-Teil von Linux betrachtet
- x86, mit alle Konfigurationen aktiviert decken nur ca 47% des Codes aller möglichen Konfigurationen ab.
- Konfiguration der Konfig 2/3 Build-System (KBUILD) und nur 17% ist tatsächlich auf Präprozessor-Ebene. D.h. auf der Ebene ist keine so richtige Aussage möglich.
- Ein Großteil der Kernel-Konfigurationen sind Treiber und foo (Hardware-Features). Der Software-Anstieg ist marginal im Verhältnis zum HW-Bezogenen.
  Maßgeblich befeuert durch Smartphones (ARM) und das wird wohl entsprechend so weiter gehen

Dh. man möchte eigentlich KBUILD untersuchen, nicht nur CPP. Makefiles sind aber schwierig zu parsen. Also ist die Idee das Buildsystem zu fragen: Für jede Konfiguration wird jede Datei gefragt: "Wirst du gebaut?" Dadurch lässt sich das ganze für ca 95% aller Dateien bestimmen, ob das der Fall ist. Der zusätzliche Einfluss dieser grobgranularen Dinge hat relativ wenige neue Fehler gefunden. Auf dieser grobgranularen Ebene werden wohl weniger Fehler gemacht.

Parser wurde dann doch gebaut und der funktioniert sehr viel besser und schneller als alles andere.

Commit-Prüfung: Jede Code-Zeile sollte vom Compiler in Zusammenhang mit Code-relevanten Abschnitten gesehen worden sein. Dafür gibt es VAMPYR. Wenn das Werkzeug zu viele false positivs liefert, wird es nicht verwendet und ist weniger Wert als eines mit false negatives, da es dann niemand verwenden würde.

Idee: alle spezifische Anwendungszenarien konstruieren. Dann ein Mapping auf die entsprechenden Ifdef-Blocks (bitmap) und den Kernel entsprechend anpassen. Also wird gemessen welche Code-Teile verwendet werden. Anscheinend reichen tatsächlich einige Minuten Laufzeit und das liefert sogar weniger CVE-Funktions-Einträge. Weiterhin wird es vermieden Geräte zu starten, die nicht benötigt werden.
blacklist: ftrace
whitelist: bootstrap

7 Conclusion, Summary

OSEK hat den Vorteil gegenüber gängigen Embedded-OS (eCOS, ..), dass alles statisch bekannt sein muss und nichts erst zur Laufzeit instantiiert werden muss. Es gibt keinen Grund die Prozesse/Threads zur Laufzeit zu erzeugen.

robustness (SDC - signed data corruption) - und ist bei ERIKA deutlich besser als eCOS. Bei eCOS werden viele verkettete Listen verwendet, auf dessen Zeiger sind deutlich empfindlicher auf Bitflips, als es Arrays sind, die in ERIKA verwendet werden können.

Hardware-Centric Operating-System Design:

> Direkte 1:1 Abbildung von Application-Foo auf die Hardware, dies ist möglich, da alles statisch bekannt ist. (Das wurde bei Sloth so gemacht). Gängige OS-Designs multiplexen die Hardware auf verschiedene Tasks.

Checksums und fehlerkorrigierender Code verschlimmern den falschen Zustand häufig, weil die Ausführung der Checks häufig lange dauert und dadurch die Zeit, in der Fehler passieren können erhöht.

dOsek:

- Hardware Memory Protection
- Inlining vieler Funktionen, um es zu vermeiden, dass Rücksprungadressen notwendig sind (Pointer)
- Arithmetic encoding of the kernel path - ist extrem teuer $\mathcal{O}(n)$ - weswegen das auch sehr viel langsamer ist als vergleichbares.
- Extreme Minimierung von SDCs Faktor $10^9 \rightarrow 10^4$, dafür aber auch eine Codegröße von Faktor 10 und Syscall-Latenz von Faktor 1.5

07-16 bis 07-22 nochmal anschauen

Entsprechendes Ziel: seltener schedulen 7-18 Globaler CFG, statische Analyse der Prioritäten von Tasks. Nach Möglichkeit für jeden Syscall überprüfen welcher Task gemäß der Prioritäten als nächstes laufen muss. Dh es muss eine CFG-Analyse erfolgen. Problematisch sind dabei trotzdem Interrupts. D.h. man muss hier beachten wie häufig und wann Interrupts auftreten können, diese sind nicht statisch bekannt und auch nicht unbedingt wie dessen Behandlung aussieht.

8 Abbildungsverzeichnis

Abbildungsverzeichnis

9 Tabellenverzeichnis

Tabellenverzeichnis

10 Listingverzeichnis

List of Listings

Liste der noch zu erledigenden Punkte