3. Übung zur Vorlesung Software-Produktlinien – Präprozessoren

TECHNISCHE UNIVERSITÄT DARMSTADT

Lösungsvorschläge

Lösung 1

- a) Lösung:

```
// Feature-Modell:
#if  (  defined (BEV)  &&
     (  defined (CO)   || defined (TEA) || defined (CAP)) &&
     (! defined (CAP)  || defined (CO)))
#define VALIDCONFIG
#endif

// Konfiguration:
#define BEV 1
#define TEA 1
```

  b) Lösung:

```
// Feature-Modell:
#define FM(BEV,CO,TEA,CAP)
        (BEV && (CO || TEA || CAP) && (!CAP || CO))

// Konfiguration:
int BEV = 1
int CO  = 0
int TEA = 1
int CAP = 0
```

- a) Lösung:

```
#ifdef i
  #define TYPE int
#elif f
  #define TYPE float
#endif

#ifdef min
  #define FUNC <
#elif max
  #define FUNC >
```

```
#endif

TYPE optimum (TYPE x, TYPE y){
    if (x FUNC y)
      return x;
    else
      return y;
}
```

b) Lösungsvorschlag:

- Funktionsvariabilität kodieren durch Umformung:

```
if (x < y)
  return [min:x,max:y];
else
  return [min:y,max:x];
```

Anwendung der Distributionsregel:

```
if (x < y)
  [min: return x;, max: return y;]
else
  [min: return y;, max: return x;]
```

und Variability Encoding durch `if`-Statement:

```
if (x < y) {
  if (min)
    return x;
  else
    return y;
}
else {
  if (min)
    return y;
  else
    return x;
}
```

- Schnittstellenvariabilität kodieren durch Duplikation der Funktion:

```
int optimum__i (int x, int y) {
  ...
}
float optimum__f (float x, float y) {
  ...
}
```

und `if`-Statement an jeder Aufrufstelle:

```
if(i)
  c = optimum__i(a,b);
else
```

```
c = optimum__f(a,b);
```

- Lösungen:
  a) `foo = X; // Annahme: X wurde zuvor nicht definiert`
  `bar = 4;`
  b) `bar = 1020;`
  Erläuterungen:
  – Kommentare `/*...*/` werden vor dem Aufruf des CPP entfernt.
  – Newline-Befehle (Backslash) werden vor dem Aufruf des CPP entfernt und in einer Zeile zusammengefasst.
  c) `((1) < (2) ? (1) : (2))`
  `((((1) < (2) ? (1) : (2))) < (2) ? (((1) < (2) ? (1) : (2))) : (2))`
  d)
```
struct command commands[] = {
  { "quit", quit_command },
  { "help", help_command },
}
```
  Erläuterungen:
  – Argument Stringification: `#NAME`
  – Argument Concatenation: `NAME ## TEXT`
  e) `(2*(1))`
  f) `x = foo(1,2);`
  g) `(4 + foo)`
  Erläuterung: rekursive Makro-Definitionen werden nicht weiter expandiert.