Ciao-Konfigurationsystem – Kurzbeschreibung

Ralf Hackner

14. Mai 2010

Inhaltsverzeichnis

1 Feature-Modell

2 Familien-Modell

2.1 Transformation des Familienmodells

Die Transformation des Familienmodells erfolgt durch das Skript transform.pl Dieses erwartet folgende Aufrufparameter:

`-i` (input dir) Quell-Wurzelverzeichnis

`-o` (output dir) Ziel-Wurzelverzeichnis

`-a` Zielarchitektur z.B. `_tc`, `_avr`

`-f` Zu nutzende Feature-Auswahl (durch kconf erzeugte *.conf Datei)

2.2 Datenstrukturen

Das Familien-Modell stellt eine Baumstruktur von Komponenten dar, deren Blätter i.d.R. die zu verwaltenden Dateien sind. Die Komponenten-Hierarchie kann dabei eine beliebige Schachtelungstiefe annehmen. Die gesamte Struktur wird auf assoziative Perl-Hashes abgebildet, wobei jede Komponente durch eine eigene Hash-Map repräsentiert wird. Die Attribute einer Komponente werden

dabei als Schlüssel-Wert-Paare dargestellt. Kind-Komponenten werden durch Referenzen auf die entsprechende Komponente/Hash-Map in einer Liste eingefügt. Eine Referenz auf diese wird dem Schlüssel `comp` (component) zugeordnet.
Ein simples Familienmodell als Beispiel:

```
1  $cmp={
2          name=>'model_exaple',
3          comp=>[
4          {
5                  name=>'component_A',
6                  comp=>[
7                  {
8                          file=>'fileA.c'
9                  },
10                 {
11                         file=>'fileA.h'
12                 }
13                 ]
14         },
15         {
16                 name=>'component_B',
17                 comp=>[
18                 {
19                         file=>'fileB.c'
20                 },
21                 {
22                         file=>'fileB.h'
23                 }
24                 ]
25         }
26         ]
27 };
```

2.3 Elementare Attribute

Neben dem bereits oben beschriebenen Attribut `comp`, gibt es eine Reihe grundlegender Komponenten-Attribute:

`name` Eindeutiger Name der Komponente. Da im gesamten System nur ein globaler Namensraum vorgesehen ist, ist darauf zu achten, dass auch bei der Verwendung mehrerer Modelle keine Namenskollisionen entstehen. Namenlose Komponenten sind zulässig. Es wird jedoch empfohlen nur Blattknoten unbenannt zu lassen, sofern diese z.B das Attribut `file` enthalten.

`vname` (**v**isible **name**) Ausführlicher Name der Komponente. Dient vor allem der Übersicht. Spielt keine technische Rolle. Dieser Name muss nicht notwendigerweise eindeutig sein.

`classname` Klassenname ggF. mit entsprechendem namespace

`file` Name einer der Komponente zugeordneten Datei. Der Pfad zur Datei ist dabei nicht Bestandteil des Dateinamens (vgl. Pfadmechanismus). Sofern keine weiteren Attribute spezifiziert sind, die ein anderes Verhalten notwendig machen, wird bei einer Transformation die Standard-Aktion auf die Datei angewendet. In der Regel wird also eine Datei dieses Namens aus dem Quellverzeichnis an den Zielort kopiert. Das Auffinden und Ablegen der Datei im Verzeichnisbaum erfolgt durch einen gesonderten Mechanismus.

`srcfile` Sollte der Name der Quelldatei von dem der Zieldatei abweichen, kann mit `srcfile` ein abweichender Dateiname angegeben werden.

`files` Analog zu `file`, jedoch wird statt eines einzelnen Namens eine Referenz auf eine Liste von Namen angegeben. Alternativ kann auch ein regulärer Ausdruck(in Perl-Syntax) als String angegeben werden (analog zu grep -P). Berücksichtigt werden dabei alle Dateien die sich am Quellpfad der Komponente oder im korrespondierenden architekturspezifischen Verzeichnis befinden. Nicht mit `srcfile` kombinierbar.

```
$cmp={
        name=>'MyClass',
        files=>['MyClass.cc', 'MyClass.h'],
        classname=>'name::space::MyClass'
};
```

oder

```
$cmp={
        name=>'MyClass',
        files=>'MyClass.(cc|h)',
        classname=>'name::space::MyClass'
};
```

2.4 Der Pfad-Mechanismus

Die Pfade der Dateien werden grundsätzlich als relativ zu einem zu spezifizierenden Wurzelverzeichnis betrachtet. Im Falle von Kopiervorgängen werden ein separates Quell- und Zielwurzelverzeichnis betrachtet. Die Pfade innerhalb dieser beiden Verzeichnisse werden als identisch betrachtet, sofern nichts anderes explizit angegeben ist. Sofern eine Komponente nicht selber Modifikationen am Pfad vornimmt, erbt sie den Pfad der ihrer Eltern-Komponente. Mit folgenden Attributen kann der Pfad einer Komponente beeinflusst werden:

`dir` Wechselt in das angegebene Verzeichnis relativ zum dem Skript übergebenen Wurzelverzeichnis. Das Wurzelverzeichnis selbst kann dabei sowohl als `'/'` oder `''` angegeben werden. Abschließende Pfadseparatoren sind optional.

`subdir` Wechselt in das angegebene Verzeichnis relativ zum Pfad der Elternkomponente

`srcdir` Quellpfad der Komponente bezüglich des Quellwurzelverzeichnisses, sofern dieser vom relativen Zielpfad abweicht.

2.5 Von Features abhängige Komponenten

Über das Attribut `depends` kann die Auswahl von Komponenten eingeschränkt werden. Als Wert kann hier eine beliebige Perl -Befehlszeile (ohne abschließenden Strichpunkt) angegeben werden. Wird dieser zu `0` evaluiert, werden die Komponente und ihre Kinder nicht ausgewählt. Um Querverweise auf das Feature-Modell zu ermöglichen, werden über den Autoload-Mechanismus von Perl parameterlose Funktionen mit den Namen der Feature bereitgestellt. Diese liefern als Rückgabewert, abhängig davon ob das Feature ausgewählt ist, `1` oder `0` zurück. Da die Funktionen nicht explizit deklariert wurden, müssen sie entsprechend der Perl-Syntax z.B. durch ein `&` kenntlich gemacht werden.

```
$cmp={
        name=>'MyModul',
        file=>'MyModul.h',
        comp=>[
        {
                name=>'FastImpl',
                depends=>'&hurry_up',
                file=> 'MyModul.c',
                srcfile=>'MyModul_fast.c'

        },
        {
                name=>'SlowImpl',
                depends=>'not &hurry_up',
                file=> 'MyModul.c',
                srcfile=>'MyModul_slow.c'
        },
        {
                file=>'optimizable.c',
                depends=>'&hurry_up and &compiler_can_optimize'
        }
        ]
};
```

2.6 Generatorskripten

Mittels des Attributes `generate` können Dateien beliebigen Inhaltes erzeugt werden. Hierzu wird ein Perl-Ausdruck als Wert angegeben. Dieser wird ausgewertet, und der Rückgabewert in die Datei geschrieben. Alle eigenen Funktionen die dazu benötigt werden, sollten in der Datei `generators.pl` abgelegt werden.

Dabei können Generatorskripte über folgende Variablen und Funktionen auf den Kontext des Familien- und Featuremodells zugreifen:

3 Applikations-Konfiguration

3.1 Erstellen des Feature-Modells

Um eine Instantiierung der Komponenten zu ermöglichen, muss zunächst eine „Profil“-Datei angelegt werden. Diese enthält Auflistungen der Namen aller notwendigen Instanzen. Auf Basis dieser Datei wird mit dem Aufruf `app_create_model.pl <profildatei>` ein neues Feature-Modell `app.fm` erzeugt. Beispiel für Eine Profildatei:

```
$apps={
        isrs=>["NetworkISR","myISR1","myISR2"],
        tasks=>["T1", "T2"],
        components=>["A","B"],
        msgPorts=>["MyMsgPort","MyMsgPortRe"],
        netPorts=>["MyNetPort","TestNetPort"],
        apps=>["App0","App1"],
        appmodes=>["myAppMode"],
        events=>["NetworkHardwareEvent","PacketWaitingEvent"],
}
```

3.2 Transformation

Durch den Aufruf `app_transform.pl <.config-Datei>` wird das Modell anschließend zur Erzeugung einer xml-Datei genutzt.