Type Inference in Dynamically Typed Feature Models

Leonardo Passos
(lpassos@gsd.uwaterloo.ca)

Generative Software Development Laboratory
University of Waterloo

November, 2011
Introduction
Introduction

- Feature models (FMs) capture the variability points of a software system.

- FMs consists of
  - Features: pieces of functionalities
  - Constraints: restrictions

- FMs are defined in terms of Feature Model Languages (FMLs).
## Introduction

- **Examples:**

<table>
<thead>
<tr>
<th>Project</th>
<th>FML</th>
<th>Component Description Language (CDL)</th>
<th>KConfig</th>
</tr>
</thead>
<tbody>
<tr>
<td>talktica Tiny JavaScript VM for Atmel AVR</td>
<td>ecos ReconOS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Introduction

• By loading a FM into a configurator, users configure and build variants of the system.

• In the configurator, users:
  • Select/Deselect features
  • Set feature values.

• The configurator enforces the constraints defined in the FM.
Introduction

eCos Configurator

Features

- File I/O operations via GDB
- Build Compiler sanity checking tests
- Common HAL tests
- FUJITSU architecture
  - FUJITSU CPU family
  - Linker script
  - Number of breakpoints supported by t32
- Use 'break' for breakpoints.
- FUJITSU MB93091 (FR-V 400) evaluation
  - Startup type
  - Number of communication channels
  - Debug serial port
  - Default console channel
  - Diagnostic serial port
  - Diagnostic serial port baud rate
  - GDB serial port baud rate
- Use LED bank for feedback on serial
- Real-time clock constants
  - Real-time clock numerator
  - Real-time clock denominator
  - Real-time clock period

Property

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL</td>
<td>ref/the-ecos-hardware-abstraction-layer.html</td>
</tr>
<tr>
<td>Macro</td>
<td>CYNUM_HAL_RTC_DENOMINATOR</td>
</tr>
<tr>
<td>File</td>
<td>unnamed2_install/include/pkgconf/hal_frv_frv400.</td>
</tr>
<tr>
<td>Value</td>
<td>100</td>
</tr>
<tr>
<td>Default</td>
<td>100</td>
</tr>
<tr>
<td>Flavor</td>
<td>data</td>
</tr>
<tr>
<td>DefaultValue</td>
<td>100</td>
</tr>
</tbody>
</table>
Introduction

• After configuration, a product (variation) is build.

• In code, features are C macros whose values are the ones provided in the configuration step:

  #define CYGNUM_HAL_RTC_DENOMINATOR 100
Introduction

• In dynamically typed FM languages (DTFML), the data value type of a feature is not always defined.

• Reasons include:
  • Ease of writing.
  • No castings when writing constraints
    - values are casted according to the context in which they appear.

• Example of DTFML: CDL
Introduction

• Disadvantages (from the user perspective):

  i. Type errors during configuration might lead to conflicts.

  ii. Type errors might not lead to conflicts, but will only be detected during build.

  iii. Type errors will not be detected in configuration nor build, and will lead to possible bugs in the resulting product.
Introduction

• Disadvantages (from the designer perspective):
  
i. No support for analysis:
    • Satisfiability checking
    • Dead feature detection
    • Common feature detection
    • False optional detection
    • Etc.

• Solvers require statically typed specifications.

Supported by solvers: Yices, Z3, Alloy, etc.
Introduction

• Disadvantages of using DTFMLs outweigh the advantages of statically typed FMLs.

• Then, why do we care about such languages, like CDL?
Introduction

• Most FML are restricted to Boolean features.

• CDL is a non-Boolean FML:
  • Data type value of features can be Booleans, numbers and strings.
  • Constraints are Boolean, number and string expressions over features.

• CDL is the most used non-Boolean FML in the open source community.
Introduction

- Our approach to overcome the disadvantages of CDL as a DTFML:
  
  An heuristic algorithm to statically compute all types of features in CDL models.

- We are the first to provide a fully automated solution to type detection.
Introduction

• Such algorithm allows:
  • Detection of type errors at the configuration step.
  • Enforcement of types in the existing model to prevent:
    - Build errors.
    - Bugs in the resulting product.
  • Analysis.
Organization

● Background on CDL

● Algorithm

● Experimental Results

● Conclusion
Background on CDL
Background on CDL

• A feature may have a data value and/or a boolean value.

• A data value has no explicit type.

• A boolean value indicates whether a feature is selected or not.
Background on CDL

- Option feature (flavor bool)
- Option feature (flavor data)
- Component feature (flavor none)
- Package feature (flavor booldata)
- Option feature (flavor booldata)
Background on CDL

- Example of a features definitions:

```c
option CYGNUM_FALLBACK_MALLOC_POOL_SIZE{
    display "Size of the fallback..."
    legal_values [32 to 0x7fffffff ]
    default_value [16384 ]
    flavor data
}
...
option CYGPKG_MEMALLOC_TESTS{
    display "Tests"
    calculated ["tests/dlmalloc..."
}
...
option CYGBLD_ISO_CTYPE_HEADER{
    display "Ctype implementation header"
    flavor booldata
}
```
Background on CDL

- Interfaces: non-configurable features, whose flavor is either bool, data or booldata.
- Control how many features of a given type can occur.

```c
interface CYGINT_KERNEL_SCHEDULER{
    display "Number of schedulers in this configuration"
    requires [1 == CYGINT_KERNEL_SCHEDULER ]
}
...
option CYGSEM_KERNEL_SCHED_BITMAP{
    display "Bitmap scheduler"
    implements [CYGINT_KERNEL_SCHEDULER ]
    ...
}```
Background on CDL

- **Constraint types:**
  - **Legal values:** constrains the feature's value
    - Constraint is a list of expressions.
  - **Default value:** sets an initial value
    - Constraint is one expression.
  - **Calculated:** defines a constant/computed value
    - Cannot be set by the user.
    - Constraint is one expression.
Background on CDL

• Constraint types:
  
  • Active-if: constraint whether a feature is active or not.
    - Active: value is accessible.
    - Inactive: value is not accessible (= 0).
    - Constraint is one expression required to produce a boolean.

  • Requires: checks whether a feature satisfies the constraint.
    - Value is accessible regardless of the constraint being SAT.
    - If unsatisfied, conflict is reported.
    - Constraint is one expression required to produce a boolean.
Background on CDL

• A reference to a feature $f$ has the following semantics in the configurator:

```java
if $f$ is selected and it is active then
    return the value of the feature
else
    return 0 // can be converted // to any type
```

• We are interested in determining the type of the value of the feature.
CDL Background

- In CDL, the types of some features can be easily determined:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Data Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>Version of the package</td>
<td>String</td>
</tr>
<tr>
<td>Interface (flavor bool)</td>
<td>True, if at least one active feature implements it</td>
<td>Boolean</td>
</tr>
<tr>
<td>Interface (flavor data/booldata)</td>
<td>Number of active features implementing it</td>
<td>Number</td>
</tr>
<tr>
<td>Flavor none</td>
<td>True</td>
<td>Boolean</td>
</tr>
<tr>
<td>Flavor bool</td>
<td>True</td>
<td>Boolean</td>
</tr>
<tr>
<td>Legal values with constants of the same type T.</td>
<td>Any value in the defined range.</td>
<td>T</td>
</tr>
<tr>
<td>Default value*</td>
<td>The value $v$ in the constraint.</td>
<td>Type(v)</td>
</tr>
</tbody>
</table>
CDL Background

- Constraint expressions:
  - Arithmetic
  - Relational
  - Conditional (cond ? pass : fail)
  - Logical
  - Function call (is_loaded, is_active, is_substr, etc)
  - String concatenation
Algorithm
Algorithm

• Work on a forest of ASTs.

• Nodes in each AST store two fields:
  • Required type: what is the node's type required to be.
  • Type: the node's actual type.

• Type precedence: S > N > B, where
  • S: String, N: Number, B:Boolean
Algorithm

- Open AST nodes: nodes whose type have not been defined yet.

- Open features:
  - A feature with currently no known or,
  - A feature that depends on another open feature and has current type different from string.

```plaintext
option C {
    ...
    legal_values [ A B ]
}
```
Proposed Solution

• Fixed types of AST nodes:

<table>
<thead>
<tr>
<th>Expression Type</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>Number</td>
</tr>
<tr>
<td>Logical</td>
<td>Boolean</td>
</tr>
<tr>
<td>Function call</td>
<td>Boolean</td>
</tr>
<tr>
<td>Relational</td>
<td>Boolean</td>
</tr>
<tr>
<td>Numeric literal</td>
<td>Number</td>
</tr>
<tr>
<td>Concatenation</td>
<td>String</td>
</tr>
<tr>
<td>String literal</td>
<td>String</td>
</tr>
</tbody>
</table>
Proposed Solution

• Algorithm has 3 phases:
  
i. Build AST forest, type graph and set initial feature types
    • Packages and interfaces
    • Flavors none and bool
    • Default values* (ignoring 0 and “”)
    • Legal values made of constants

One AST per constraint.
Proposed Solution

• Algorithm has 3 phases:
  
  ii. Do until a fixed point is reached:
    
    – Propagate required and actual types in all trees.
    – Set feature types based on the context in which they are used.

  iii. Set remaining features to be string and propagate actual and required types where needed until a fixed point is reached.
Proposed Solution

• Example:

```python
option A {
    flavor data
}

option B {
    flavor data
}

option C {
    requires[A == 20 ? B : 10) ||
    is_substr(B, “xx”)
    legal_values [ A B ]
    flavor data
}
```
Phase One

• **Build AST forest**, type graph and set initial feature types:

```cpp
option A {
    flavor data
}

option B {
    flavor data
}

option C {
    requires[A == 20 ? B : 10) || is_substr(B, “xx”)]
    legal_values [ A B ]
    flavor data
}
```
Phase One

- Build AST forest, **type graph** and set initial feature types:

  ```
  option A {
    flavor data
  }

  option B {
    flavor data
  }

  option C {
    requires[A == 20 ? B : 10] ||
    is_substr(B, "xx")
    legal_values [ A B ]
    flavor data
  }
  ```
Phase One

• Build AST forest, type graph and set initial feature types:

```plaintext
option A {
    flavor data
}

option B {
    flavor data
}

option C {
    requires[A == 20 ? B : 10] ||
       is_substr(B, “xx”)]
    legal_values [ A B ]
    flavor data
}
```

No feature types are given.
Phase Two

- **Propagate types in forest** and then propagate context:

```
         R=B, T=B
        /   \      
       Or     R=?, T=?
               /  \   
              is_substr R=?, T=B
                     /  \   
                    B   “xx”
                    \    
                    R=?, T=?
                   /    
                  R=?, T=S
                /     
               R=B, T=B
              /     
             Or     R=?, T=?
                /  \   
               Eq     R=?, T=?
              /     R=?, T=S
             C      B
            /     /   \  
           R=?, T=?  R=?, T=?
             R=?, T=?
Phase Two

- Propagate types in forest and then propagate context:

```
R=B, T=B

Cond

R=?, T=?
R=?, T=B

R=?, T=N

R=?, T=?

Eq

B

20

R=?, T=?
A

R=?, T=N

R=?, T=?

Or

is_substr

R=?, T=B

B

"xx"

R=?, T=?

R=?, T=S

Or

Eq

C

R=?, T=?
A

R=?, T=S

Eq

C

R=?, T?.
B

R=?, T=?
```

```
Phase Two

- Propagate types in forest and then propagate context:
Phase Two

- Propagate types in forest and then propagate context:
Phase Two

- Propagate types in forest and then propagate context:
Phase Two

- Propagate types in forest and then propagate context:

```
Or
  ▲                  ▲
  ▼                  ▼
Cond                  is_substr
  ▲                  ▲
  ▼                  ▼
  ▲                  ▲
  ▼                  ▼
  ▲                  ▲
  ▼                  ▼
Eq                  Eq
  ▲                  ▲
  ▼                  ▼
  ▲                  ▲
  ▼                  ▼
  ▲                  ▲
  ▼                  ▼
  ▲                  ▲
  ▼                  ▼
A                  B
R=B, T=B           R=B, T=B
R=B, T=I           R=S, T=?
R=B, T=20          R=?, T=S
R=?, T=?           R=?, T=?
R=?, T=20          R=?, T=?
R=?, T=I           R=?, T=?
R=?, T=20          R=?, T=I
R=?, T=20          R=?, T=20
R=?, T=I           R=?, T=I
R=? S, T=I         R=?, T=I
R=?, T=S           R=? S, T=I
R=?, T=S           R=? S, T=I
```

- **Propagate types in forest and then propagate context:**
Phase Two

- **Propagate types in forest** and then propagate context:
Phase Two

- **Propagate types in forest** and then propagate context:

```
Or
  \( R=B, T=B \)
  \( R=B, T=? \)
  \( R=B, T=N \)
  \( \text{Cond} \)
    \( B \)
    \( 10 \)
  Or
    \( R=B, T=B \)
    \( \text{is_substr} \)
      \( B \)
      \( "xx" \)
    \( R=S, T=? \)
    \( R=S, T=S \)
  Or
    \( R=B, T=B \)
    \( \text{Eq} \)
      \( \text{R=?}, T=? \)
      \( \text{C} \)
      \( \text{A} \)
    \( \text{Eq} \)
      \( \text{R=?}, T=B \)
      \( \text{C} \)
      \( \text{B} \)
      \( \text{R=?}, T=? \)
    \( \text{R=?}, T=? \)
    \( \text{R=?}, T=? \)
  \( R=B, T=B \)
```
Phase Two

- Propagate types in forest and then propagate context:
Phase Two

- Propagate types in forest and then propagate context:
Phase Two

- Propagate types in forest and then propagate context:

```
<table>
<thead>
<tr>
<th></th>
<th>R=B, T=B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eq</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>R=B, T=?=</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eq</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>&quot;xx&quot;</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>R=S, T=?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eq</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>R=S, T=S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eq</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>R=?, T=?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eq</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>R=?, T=?</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>R=?, T=?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eq</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>
```
Phase Two

• Propagate types in forest and then propagate context:

Choose candidates = \{A, B\}
Phase Two

- Propagate types in forest and then **propagate context**:

```plaintext
        R=B, T=B
       /    \
      /      \    
     R=B, T=B  \
    /      \    
   /  Eq   10  \
  /      \    
 R=? , T= ?  
    
        R=B, T=?
       /    \
      /      \    
     R=B, T=N  \
    /      \    
   /  Eq   R=?  \
  /      \    
 R=? , T=?  
    
        R=S, T=?
       /      \
      /        \    
     B          \
    /      \    
   /  is_substr  \    
  /      \      \    
 R=B, T=B  “xx”  R=S, T=S
```

```
        R=B, T=B
       /    \
      /      \    
     /  Eq   R=?  \
    /      \    
   /  Eq   R=B, T=B  \
  /      \    
 R=? , T= ?  R=? , T=?
```
Phase Two

- Propagate types in forest and then propagate context:
Phase Two

- Propagate types in forest and then **propagate context**:

```
Cond
\|   \|   \|
R=B, T=B  Eq  R=B, T=S
R=?, T=?  B   R=B, T=N
       10

Or

is_substr
\|   \|
R=B, T=B  B
"xx"

Or

Eq
\|   \|
R=B, T=B  Eq
R=B, T=B

R=B, T=S

R=?, T=S
C

A

R=?, T=?

C

R=?, T=S

B

R=?, T=S

R=?, T=S
```

R=B, T=S

R=?, T=N

R=?, T=S
Phase Two

- Propagate types in forest and then propagate context:
Phase Two

- Propagate types in forest and then propagate context:

```
Or

Cond

<table>
<thead>
<tr>
<th>R=B, T=B</th>
<th>B</th>
<th>10</th>
</tr>
</thead>
</table>

Eq

<table>
<thead>
<tr>
<th>R=B, T=S</th>
<th>R=B, T=N</th>
</tr>
</thead>
</table>

is_substr

| R=B, T=B | B | “xx” |

Or

```

```
Eq

<table>
<thead>
<tr>
<th>R=B, T=S</th>
<th>R=S, T=S</th>
</tr>
</thead>
</table>

Eq

<table>
<thead>
<tr>
<th>R=B, T=B</th>
<th>R=B, T=B</th>
</tr>
</thead>
</table>

```

```
Eq

<table>
<thead>
<tr>
<th>R=S, T=S</th>
<th>R=S, T=S</th>
</tr>
</thead>
</table>

```

```
R=S, T=S
```

```
R=?, T=？
```

```
R=S, T=？
```

```
R=?, T=？
```

```
R=S, T=？
```

```
R=?, T=？
```

```
R=S, T=？
```

```
R=?, T=？
```

```
R=S, T=？
```

```
R=?, T=？
```

```
R=S, T=？
```

```
R=?, T=？
```

```
R=S, T=？
```

```
R=?, T=？
```

```
R=S, T=？
```

```
R=?, T=？
```

```
R=S, T=？
```

```
R=?, T=？
```

```
R=S, T=？
```

```
R=?, T=？
```
Phase Two

- **Propagate types in forest** and then propagate context:

```
R=B, T=B

Or

Cond

B

10

R=B, T=S

R=B, T=N

is_substr

R=B, T=B

“xx”

B

R=S, T=S

R=S, T=S

Or

Eq

A

20

R=?, T=?

R=?, T=N

R=B, T=B

R=S, T=S

R=S, T=T

Eq

C

A

B

R=S, T=S

R=S, T=T

R=S, T=?”

R=S, T=S

R=S, T=?”
```
Phase Two

- Propagate types in forest and then propagate context:

Choose candidates = \{A\}
Phase Two

- Propagate types in forest and then propagate context:

```
Or
  Cond
  |     10
  |     B
  |    Eq
  |   A 20
  | R=?, T=?
  | R=?, T=N
  | R=B, T=B

Or
  is_substr
    B
    “xx”
    R=B, T=B
    R=S, T=S
    R=S, T=S

Or
  Eq
  R=B, T=B
    Eq
    R=B, T=B
    Eq
    R=S, T=S
    R=S, T=S
    R=S, T=S
    R=S, T=S
```

R=S, T=S
Phase Three

Set remaining features to be string and propagate actual and required types where needed.
Phase Three

• Set remaining features to be string and propagate actual and required types where needed.
In reality, the type graph is a disconnected graph, where each component is the topological sorting of each SCC in the type dependency graph.

```
option A {
  legal_values [B ? C : D]
  flavor data
}

option B { flavor data }

option C {
  requires [D > 0]
  default[B ? A : D]
  flavor data
}

option D {
  flavor data
}
```
Experiments
Experiments

- Real world projects (119 models)
  - eCos: 116 models
  - Redboot4lpc: 1 model
  - PSAS: 1 model
  - Talktic: 1 model
Experiments

Loaded vs Unloaded

98.8

1.2

Loaded
Unloaded
Experiments

Inference Need (Loaded Features)

- Given: 62.6
- Need to be inferred: 37.4
Experiments

Inference (Loaded Features)

- Given: 62.6%
- Inferred by context (2nd phase): 18.4%
- No context (taken as string - 3rd phase): 19.0%
Experiments

Inference Moment (Unloaded Features)

By context (1st phase)
By string choosing (2nd phase)
Experiments

- Possible errors during configuration
  - Min: 0; Max: 7; Median: 0

- Detection:
  - Numeric features in arithmetic expressions, but not constrained by legal_values nor calculated.
  - String features used in arithmetic expressions.
Experiments

• No detection of build errors or bugs:
  • Designers, however, can rely on the discovered types to enforce the range of feature values (legal_values).

• Informal feedback on correctness:
  • cdI2clafer developer (direct consumer)

• TODO: measure precision
  • Depend on volunteers to avoid any biased results.
Related Work

- Range fixes (Yingfei, submission to ICSE 2012)
  - Adhoc support for casting.
  - Semi-automatic detection of types
    - users must annotate model
  - No indication of possible errors.
Conclusion

- DTFMLs need type inference:
  - Avoid configuration and build errors.
  - Avoid bugs.
  - Allow analysis.

- We presented an algorithm to detect features types
  - Captures possible configuration errors.
  - Build errors and bugs can be achieved by enforcing discovered types (legal_values).
  - Allow reasoning of models.
Thanks for Listening

Questions?