

Lab on Model-Driven Software Development

Creating a Model-Based Programming Language

The lab guides students through the process of designing and realizing an own model-based programming language, writing a compiler to an intermediate language and creating an execution environment revolving around the language's interpreter.

Technische Universität Braunschweig | Institute of Software Engineering and Automotive Informatics

Dr.-Ing. Christoph Seidl | c.seidl@tu-braunschweig.de | Phone +49 (0) 531 391-2296

1. Model-Based Programming Language (MPL)

MPL is a Turing complete programming language with variables, calculations, control flow statements and operations. MPL is developed in four revisions with increasing complexity, where the first one is developed in a class-room tutorial and the other three in homework assignments. The resulting metamodel and textual syntax are used to generate an editor.

The Model-Based Programming Language (MPL) is a Simple GPL

```

Program Test
Variables a, b, c := 3.
If (c > 2) Then
  a := 3.
Else
  a := 2.
End.
b := (a + 4) * c.
Trace(b).
End.
    
```

- Contains variables, assignments, expressions, procedures, functions
- Keywords and built-in operations start with a capital letter
- Lines end with a dot.
- Integer as only (implicit) type
- Assignment with :=
- Comparison for equality with =
- Comparison for inequality with <>

The Generated Textual Editor Integrates into the Eclipse IDE

- Syntax highlighting
- Automatic background parsing
- Outline view shows model (abstract syntax)

2. Model-Based Intermediate Language (MIL)

MIL features an instruction set similar to that of processors with a total of 19 instructions, which are easy to execute. MIL is developed in four revisions that align with those of MPL. In class-room tutorials and homework assignments, students design a metamodel and textual syntax for MIL while learning the semantics of each instruction.

The Intermediate Language Emulates Operations Performed by a Processor

```

//Example
lod 10
sto a
lod 0
sto f
lod a
cal Fac
sto f
jpc EndProgram
Fac:
sto a2
EndProgram:

//If-Then Statement
//Condition
lod a2
lod 0
eq
jpc EndIf
lod 1
ret
EndIf:
//...
EndProgram:
    
```

- Simple instruction set
- Integer as only (implicit) type
- Boolean values synthesized
 - a = 0 == false
 - a <> 0 == true
- Sequential execution (except for jump/call/return instructions)
- Simple interpreter
- Hybrid stack/register machine
- Compiler MPL to MIL (see later)

3. Execution Environment

Instead of executing MPL directly, it is compiled to MIL by model-to-model transformation. Students devise and implement appropriate translations from MPL constructs to MIL instructions. The execution environment for MIL is a hybrid stack/register machine students implement in Java. The implementation is relatively straight forward, which makes it portable so that MIL instructions and, thus, MPL code can be executed on different platforms when necessary.

Compiling MPL Means Encoding in MIL

```

Program Example
Variables a := 10, f.
f := Fac(a).
End.

Function Fac(a)
If (a = 0) Then
Return 1.
End.
Return a * Fac(a - 1).
End.

//Example
//If-Then Statement
//Condition
lod 10
sto a
lod 0
sto f
eq
jpc EndIf
lod a
cal Fac
sto f
jpc EndProgram
Fac:
sto a2
EndProgram:
    
```

MIL Programs are Executed in a Hybrid Stack/Register Machine

Key	Value
f	4
a	3
b	7

Metamodeling
Constraints
Textual Syntaxes
Model Transformation
Model Interpreter
IDE Integration

The Conditional Jump Instruction Jumps when Top of Stack is Zero

4. Extensions to Language Infrastructure

Students follow up on the guided tasks in free exercises where they can choose from a number of potential topics to extend and refine the language infrastructure, e.g., introducing a type system to MPL, creating a graphical programming language that compiles to MIL, writing a debugger for MIL or compiling MIL to a binary format with an interpreter for embedded devices. To complete the lab, students have to realize at least one free exercise topic.

Type System for MPL

```

Program Example
Variables a : Integer := 10, f : Real.
f := Fac(a).
End.

Function Fac(a) : Integer
Variables s : String := "Hello",
        b : Boolean.
If (a = 0) Then
Return 1.
End.
Return a * Fac(a - 1).
End.
    
```

- Types: Integer, Real (floating point), Boolean, String
- Respective literals, e.g., 1.2, true, "Test"
- Determine which types are compatible with one another, e.g., 1.2 may be used as Integer as well
- Ensure that assignments and comparisons are only between compatible types

MIL for Embedded Devices

- Create compact binary byte code files
- In Java, ...
 - ... devise a binary byte code (instructions as op codes)
 - ... create a compiler to the binary byte code
- In another programming language (e.g., C++, Ruby), ...
 - ... create a parser for the binary byte code
 - ... write an interpreter for the binary byte code with identical semantics as the Java MIL interpreter