

Debugging Natural Semantics Specifications

Adrian Pop and Peter Fritzson

Programming Environment Laboratory
Department of Computer and Information Science
Linköping University
2005-09-20

AADEBUG'2005, September 19-21,
Monterey, Canada, USA

- Introduction
 - Natural Semantics
 - Relational Meta-Language (RML)
- The debugging framework for RML
 - Framework overview
 - Emacs integration
 - Data value browser
- Demo
- Conclusions and Future Work

Natural Semantics and Relational Meta-Language

- Natural Semantics, a formalism widely used for specification of programming language aspects
 - type systems
 - static, dynamic and translational semantics
 - few implementations in real systems
- Relational Meta-Language (RML)
 - a system for generating efficient executable code from Natural Semantics specifications
 - fast learning curve, used in teaching and specification of languages such as: Java, Modelica, MiniML, Pascal, ..
 - *developed by Mikael Petterson*
 - *“Compiling Natural Semantics” PhD Linköping University 1996*
 - *also as Springer Lecture Notes in Computer Science (LNCS) vol. 1549 in 1999*
 - **previously no support for debugging.**

$$\frac{H_1 \vdash T_1 : R_1 \quad \dots \quad H_n \vdash T_n : R_n}{H \vdash T : R} \quad \text{if } \langle \text{cond} \rangle$$

- H_i are hypotheses (environments)
- T_i are terms (pieces of abstract syntax)
- R_i are results (types, run-time values, changed environments)
- $H_j \vdash T_j : R_j$ are sequents
- Premises or preconditions are above the line
- Conclusion is below the line
- Condition on the side if exists must be satisfied

Natural Semantics vs. Relational Meta-Language

RML has the same visual syntax as Natural Semantics

```
rule    <cond>
        RelName1(H1, T1) => R1 & ...
        RelNameN(Hn, Tn) => Rn &
        -----
        RelName(H, T)  => R
```

RML language properties

- Separation of input and output arguments/results
- Statically strongly typed
- Polymorphic type inference
- Efficient compilation of pattern-matching

Natural Semantics vs. Relational Meta-Language

Natural Semantics formalism

integers:

$v \in \text{Int}$

expressions (abstract syntax):

$e \in \text{Exp} ::= v$

| $e1 + e2$

| $e1 - e2$

| $e1 * e2$

| $e1 / e2$

| $-e$

Relational Meta-Language

module exp1:

(* Abstract syntax of language
Exp1 *)

datatype Exp = INTconst **of** int

| ADDop **of** Exp * Exp

| SUBop **of** Exp * Exp

| MULop **of** Exp * Exp

| DIVop **of** Exp * Exp

| NEGop **of** Exp

relation eval: Exp => int

end

Natural Semantics vs. Relational Meta-Language

Natural Semantics formalism

$$(1) \quad v \Rightarrow v$$

$$(2) \quad \frac{e1 \Rightarrow v1 \quad e2 \Rightarrow v2 \quad v1+v2 \Rightarrow v3}{e1+e2 \Rightarrow v3}$$

$$(3) \quad \frac{e1 \Rightarrow v1 \quad e2 \Rightarrow v2 \quad v1-v2 \Rightarrow v3}{e1-e2 \Rightarrow v3}$$

$$(4) \quad \frac{e1 \Rightarrow v1 \quad e2 \Rightarrow v2 \quad v1*v2 \Rightarrow v3}{e1*e2 \Rightarrow v3}$$

$$(5) \quad \frac{e1 \Rightarrow v1 \quad e2 \Rightarrow v2 \quad v1/v2 \Rightarrow v3}{e1/v2 \Rightarrow v3}$$

$$(6) \quad \frac{e \Rightarrow v \quad -v \Rightarrow vneg}{-e \Rightarrow vneg}$$

Relational Meta-Language

relation eval: Exp => int =

axiom eval(INTconst(ival)) => ival

rule eval(e1) => v1 &
eval(e2) => v2 & v1 + v2 => v3

eval(ADDop(e1, e2)) => v3

rule eval(e1) => v1 &
eval(e2) => v2 & v1 - v2 => v3

eval(SUBop(e1, e2)) => v3

rule eval(e1) => v1 &
eval(e2) => v2 & v1 * v2 => v3

eval(MULop(e1, e2)) => v3

rule eval(e1) => v1 &
eval(e2) => v2 & v1 / v2 => v3

eval(DIVop(e1, e2)) => v3

rule eval(e) => v & -v => vneg

eval(NEGop(e)) => vneg

end (* eval *)

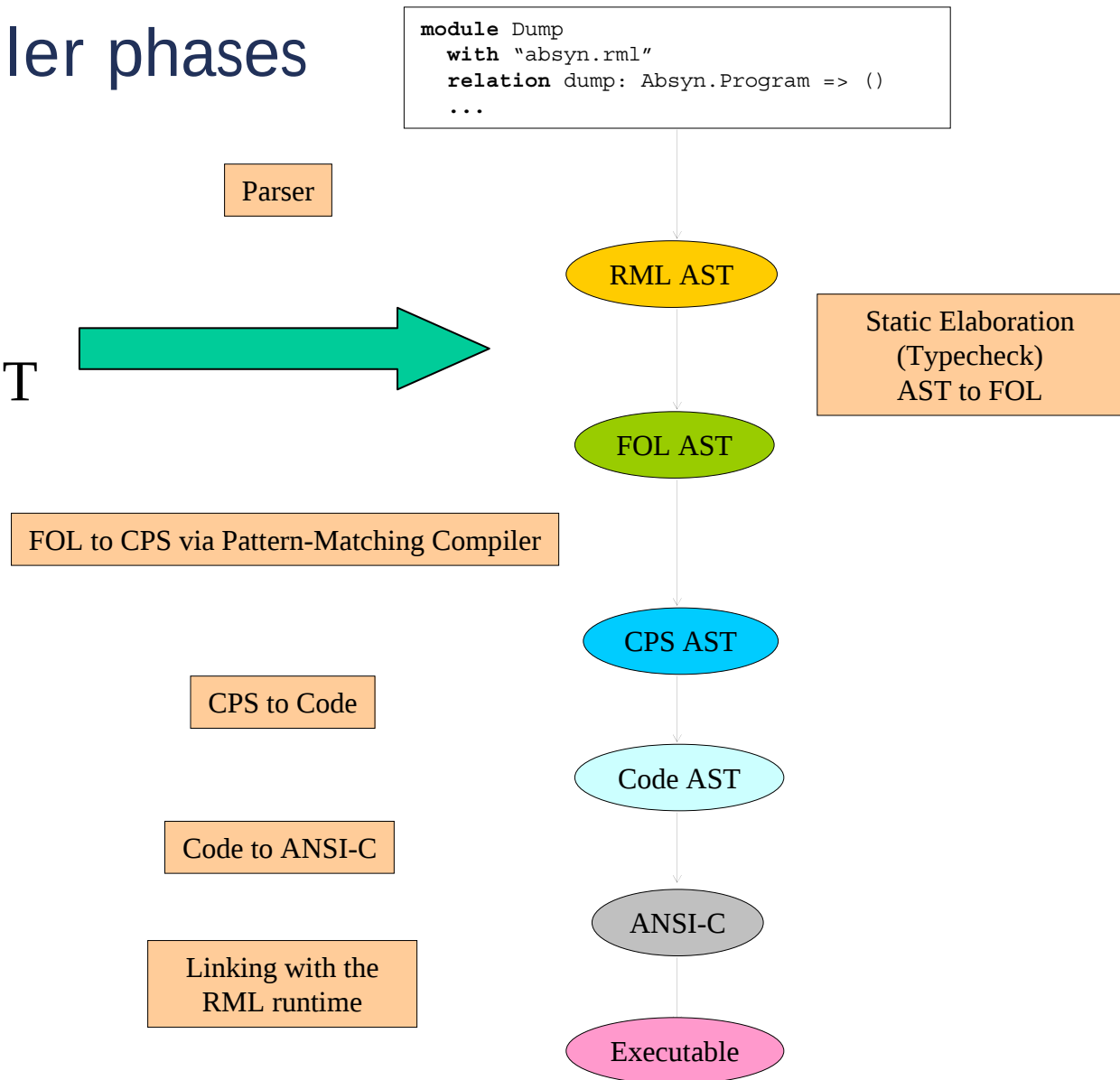
The Need for RML Debugging

- Facilitate language learning
 - run, stop and inspect features
- Large specifications are hard to debug
 - Example: The OpenModelica compiler for Modelica
 - **43** packages
 - **57083** lines of code
 - **4054** functions
 - **132** data structures
 - Also Java 1.2 specification ~18000 lines generating a bytecode compiler

The RML Debugging framework

- The RML compiler phases

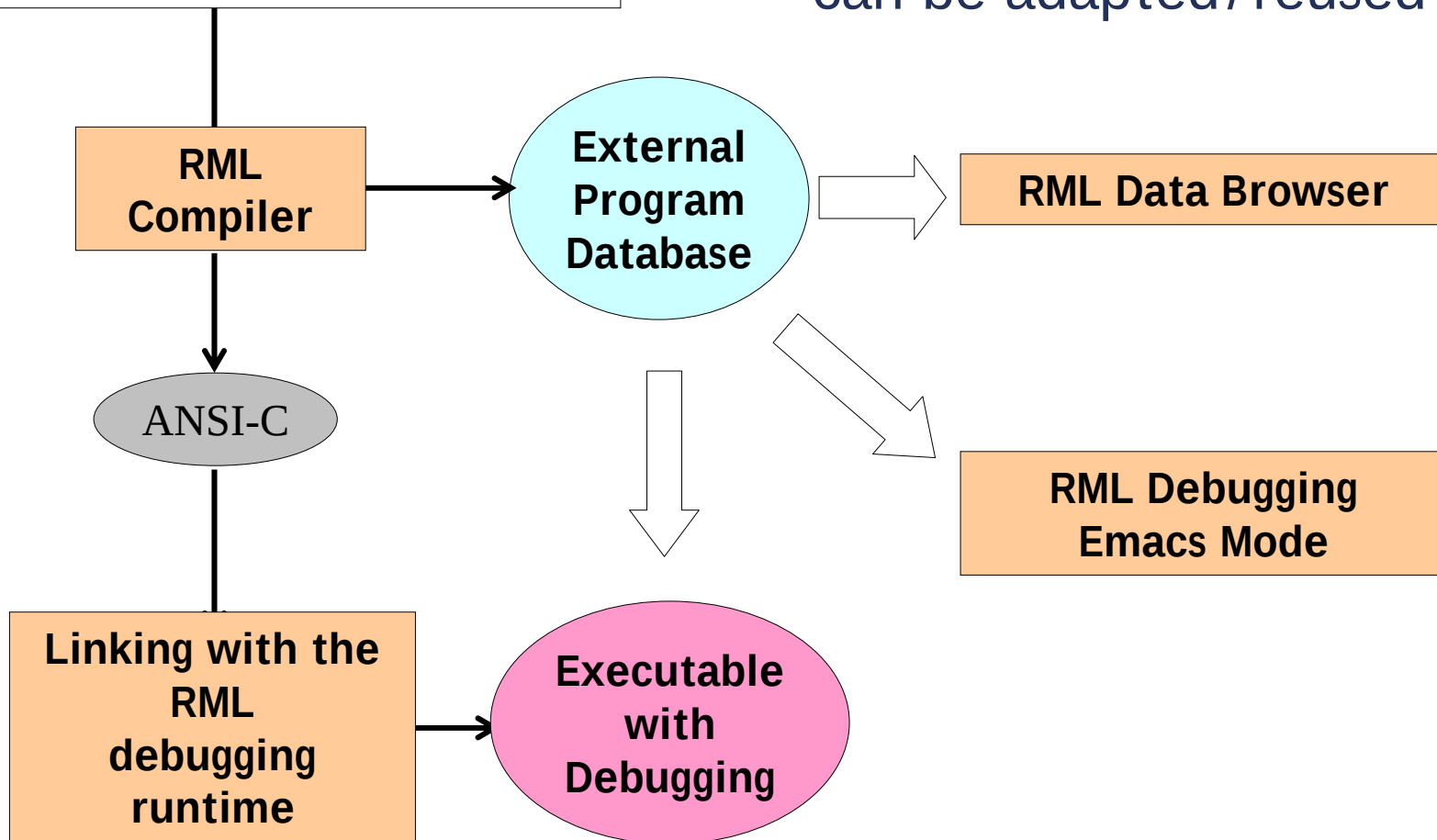
Instrumentation with
debug nodes at the AST
level



The RML Debugging framework

- Portable debugging framework based on code instrumentation and a small runtime interface; can be adapted/reused

```
module Dump
  with "absyn.rml"
  relation dump: Absyn.Program =>
  ()
  ...
```



Debugger Implementation - Instrumentation

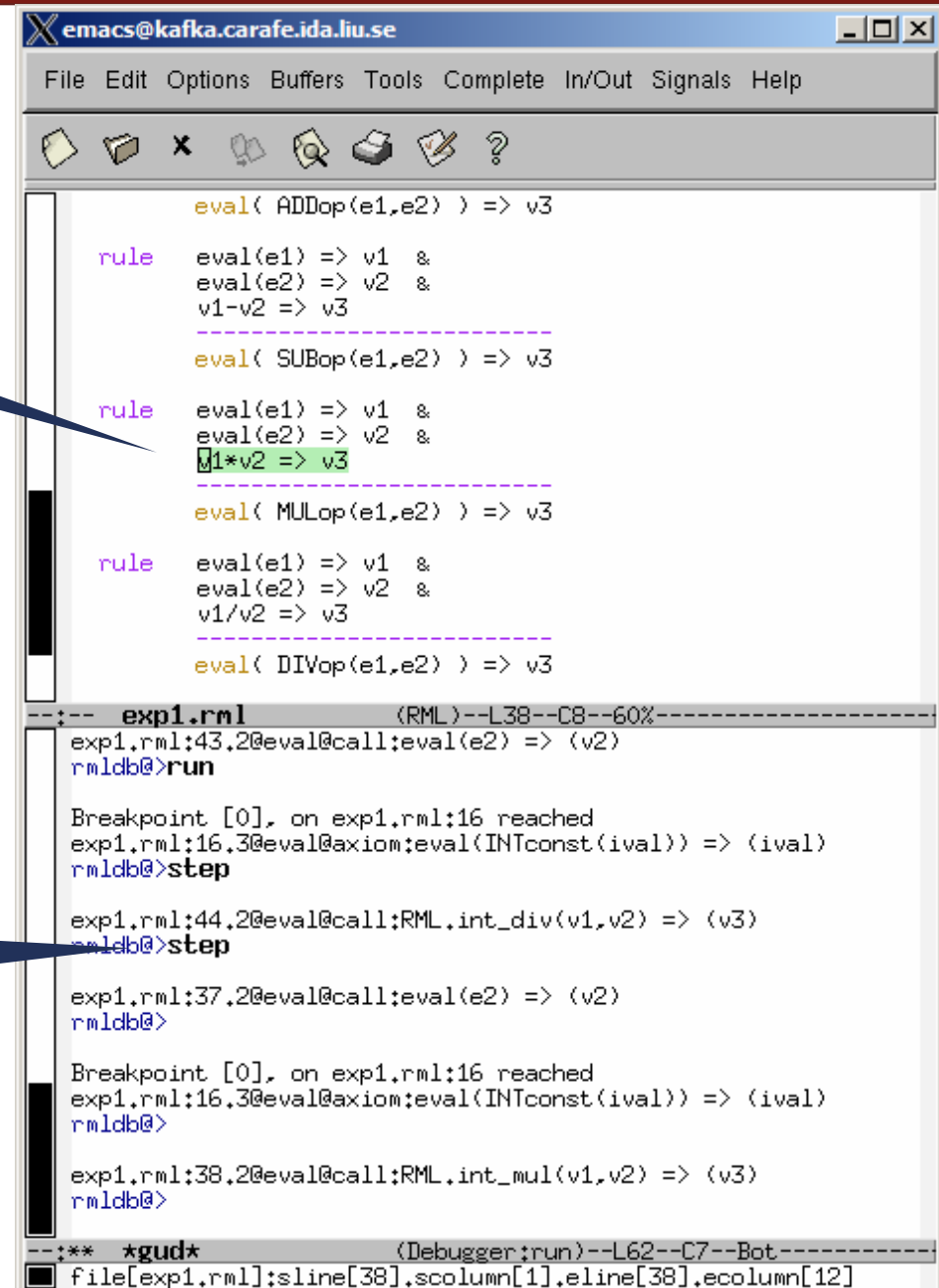
```
(* Evaluation semantics of Exp1 *)  
relation eval: Exp => int =  
axiom eval(INTconst(ival)) => ival  
  
rule  eval(e1) => v1 &  
      eval(e2) => v2 &  
      v1 + v2 => v3  
      -----  
      eval( ADDop(e1, e2) ) => v3  
  
...  
end (* eval *)
```

```
(* Evaluation semantics of Exp1 *)  
relation eval: Exp => int =  
axiom eval(INTconst(ival)) => ival  
  
rule  RML.debug_push_in01("e1",e1) &  
      RML.debug(...) &  
      eval(e1) => (v1) &  
      RML.debug_push_out01("v1",v1) &  
      RML.debug_push_in01("e2",e2) &  
      RML.debug(...) => () &  
      eval(e2) => (v2) &  
      RML.debug_push_out01("v2",v2) &  
      RML.debug_push_in02("v1",v1,"v2",v2) &  
      RML.debug(...) &  
      RML.int_add(v1,v2) => (v3)  
      -----  
      eval(ADDop(e1,e2)) => (v3)  
  
...  
end (* eval *)
```

Debugger Functionality (1)

Breakpoints

Stepping and Running



```
emacs@kafka.carafe.ida.liu.se
File Edit Options Buffers Tools Complete In/Out Signals Help

eval( ADDop(e1,e2) ) => v3

rule
  eval(e1) => v1 &
  eval(e2) => v2 &
  v1-v2 => v3
-----
eval( SUBop(e1,e2) ) => v3

rule
  eval(e1) => v1 &
  eval(e2) => v2 &
  v1*v2 => v3
-----
eval( MULop(e1,e2) ) => v3

rule
  eval(e1) => v1 &
  eval(e2) => v2 &
  v1/v2 => v3
-----
eval( DIVop(e1,e2) ) => v3

--:-- exp1.rml (RML)--L38--C8--60%-----
exp1.rml:43.2@eval@call:eval(e2) => (v2)
rmlldb>run

Breakpoint [0], on exp1.rml:16 reached
exp1.rml:16.3@eval@axiom:eval(INTconst(ival)) => (ival)
rmlldb>step

exp1.rml:44.2@eval@call:RML.int_div(v1,v2) => (v3)
rmlldb>step

exp1.rml:37.2@eval@call:eval(e2) => (v2)
rmlldb>

Breakpoint [0], on exp1.rml:16 reached
exp1.rml:16.3@eval@axiom:eval(INTconst(ival)) => (ival)
rmlldb>

exp1.rml:38.2@eval@call:RML.int_mul(v1,v2) => (v3)
rmlldb>

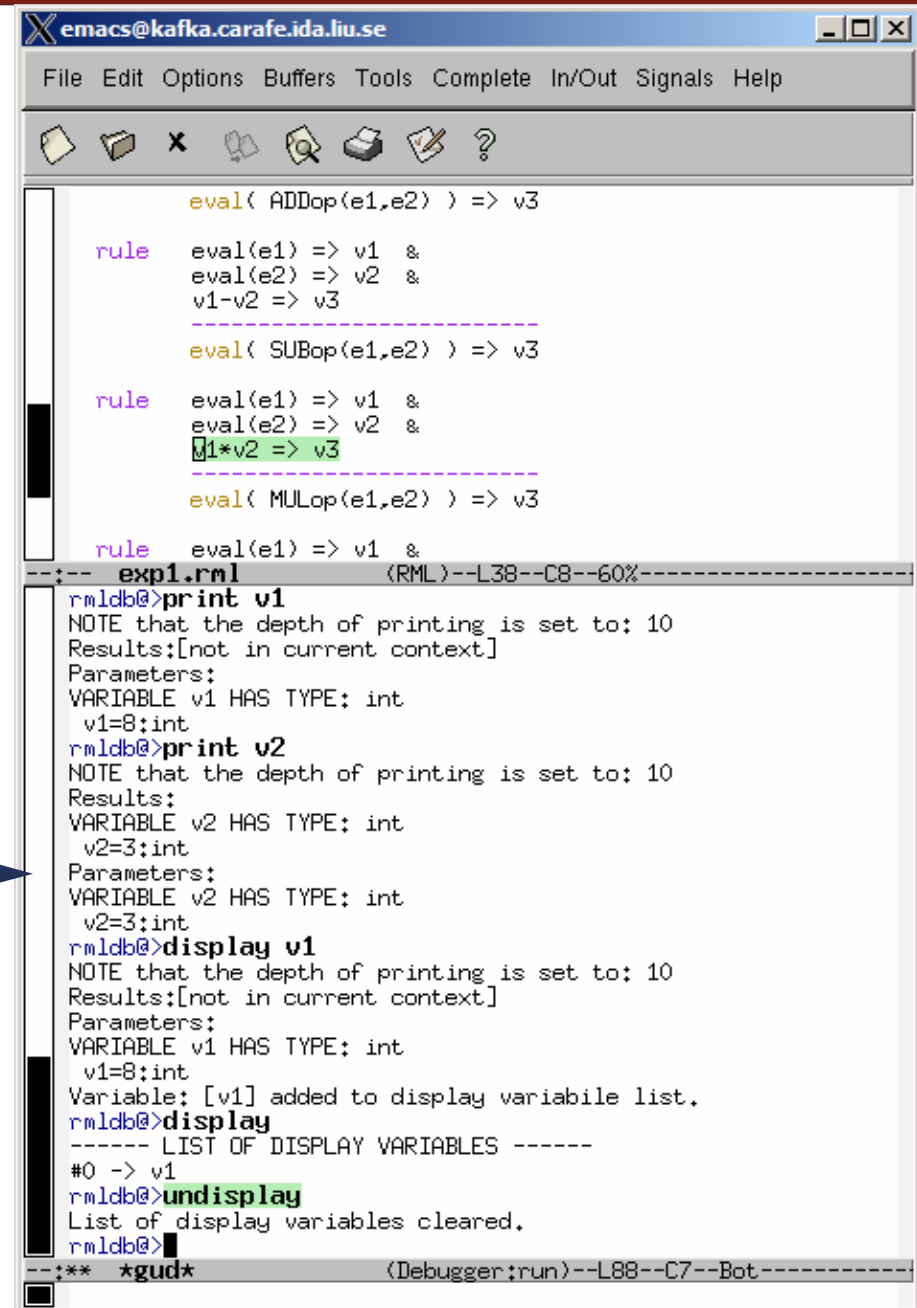
--:** *gud* (Debugger:run)--L62--C7--Bot-----
file[exp1.rml]:sline[38].scolumn[1].eline[38].ecolumn[12]
```

Debugger Functionality (2)

- Additional functionality
 - viewing status information
 - printing backtrace information (stack trace)
 - printing call chain
 - setting debugger defaults
 - getting help

Examining data

- printing variables
- sending variables to an external browser



The screenshot shows the Emacs debugger interface. The top window displays RML code with rules for ADDop, SUBop, and MULop. The bottom window shows the debugger's internal state and commands. A blue callout bubble points to the 'Examining data' section of the slide.

```
emacs@kafka.carafe.ida.liu.se
File Edit Options Buffers Tools Complete In/Out Signals Help

eval( ADDop(e1,e2) ) => v3

rule
  eval(e1) => v1 &
  eval(e2) => v2 &
  v1-v2 => v3
-----
eval( SUBop(e1,e2) ) => v3

rule
  eval(e1) => v1 &
  eval(e2) => v2 &
  v1*v2 => v3
-----
eval( MULop(e1,e2) ) => v3

rule
  eval(e1) => v1 &

--:-- exp1.rml (RML)--L38--C8--60%-----

rmlldb>print v1
NOTE that the depth of printing is set to: 10
Results:[not in current context]
Parameters:
VARIABLE v1 HAS TYPE: int
v1=8:int
rmlldb>print v2
NOTE that the depth of printing is set to: 10
Results:
VARIABLE v2 HAS TYPE: int
v2=3:int
Parameters:
VARIABLE v2 HAS TYPE: int
v2=3:int
rmlldb>display v1
NOTE that the depth of printing is set to: 10
Results:[not in current context]
Parameters:
VARIABLE v1 HAS TYPE: int
v1=8:int
Variable: [v1] added to display variable list.
rmlldb>display
----- LIST OF DISPLAY VARIABLES -----
#0 -> v1
rmlldb>undisplay
List of display variables cleared.
rmlldb>

--:** *gud* (Debugger:run)--L88--C7--Bot-----
```

Browser for RML Data Structures (1)

The screenshot shows the RMLDataViewer application. The top pane displays a tree view of RML data structures. The tree is rooted at 'RML Data Viewer' and contains a folder 'p / print depth: 10 / type: Absyn.Program / file: main.rml / position: 428.22.428.22 / live range: 426.3.428.22'. This folder contains a folder 'Absyn.PROGRAM[2] / type: ((Absyn.Class list, Absyn.Within) => (Absyn.Program)) / file: absyn.rml'. This folder contains a folder 'LIST / type: Absyn.Class list / file: RML / position: 0.0.0.0 / depth: 1'. This folder contains three folders 'Absyn.CLASS[6] / type: ((string, bool, bool, bool, Absyn.Restriction, Absyn.ClassDef) => (Absyn.Class)) / file: RML / position: 0.0.0.0 / depth: 2'. The first 'Absyn.CLASS[6]' folder contains four items: 'STRING / type: string / file: Division / position: 0.0.0.0 / depth: 3', 'false / type: bool / file: RML / position: 0.0.0.0 / depth: 3', 'false / type: bool / file: RML / position: 0.0.0.0 / depth: 3', and 'false / type: bool / file: RML / position: 0.0.0.0 / depth: 3'. The bottom pane shows a code editor with the following code:

```
run_tornado_cg_q() => true
& Parser.parse f => p
& SCode.elaborate(p) => p'
& Inst.instantiate(p') => d
(*& transform_if_flat(f,d) => d *)
& Absyn.last_classname(p) => cname
& Tornado.generate_code(p,d,cname)
-----
translate_file [f]

rule (*Print.print_buf "Parsing\n" &          *)
is_modelica_file(f)
& Parser.parse f => p
& Debug.fprint ("dump", "\n----- Parsed program
-----\n")
& Debug.fcall ("dumpgraphviz", DumpGraphviz.dump, p)
& Debug.fcall ("dump", Dump.dump, p)

& Debug.fprint ("info",
"\n-----\n")
& Debug.fprint ("info", "---elaborating\n")
& SCode.elaborate(p) => p'
```

The code editor has tabs for 'Help', 'main.rml', and 'absyn.rml'. The 'absyn.rml' tab is active. The code is color-coded: keywords are in blue, identifiers are in black, and strings are in red. The line '& SCode.elaborate(p) => p'' is highlighted in blue, indicating the current execution point.

Variable value
inspection

Current Execution
Point

Browser for RML Data Structures (2)

The screenshot shows the RMLDataViewer application. The top pane displays a tree view of the RML data structure. The root node is 'RML Data Viewer', which contains a folder 'p' with a file 'main.rml'. This file contains an 'Absyn.PROGRAM[2]' node, which in turn contains a 'LIST' node. The 'LIST' node contains three 'Absyn.CLASS[6]' nodes. The bottom pane shows the code editor with the following content:

```
(** Within statements *)
datatype Within = WITHIN of Path | TOP

(** - Classes *)
(** A class definition consists of a name, a flag to indicate if this *)
(** class is declared as 'partial', the declared class restriction, *)
(** and the body of the declaration. *)
datatype Class = CLASS of Ident (* Name *)
                * bool          (* Partial *)
                * bool          (* Final *)
                * bool          (* Encapsulated *)
                * Restriction    (* Restriction *)
                * ClassDef       (* Body *)

(** The 'ClassDef' type contains the definition part of a class *)
(** declaration. The definition is either explicit, with a list of *)
(** parts ('public', 'protected', 'equationc' and 'algorithm'), or it *)
(** is a definition derived from another class or an enumeration type. *)
(** For a derived type, the type contains the name of the derived class and
an optional *)
```

Data structure
browsing

Data structure
definition

- Conclusions
 - debugging framework for Natural Semantics
 - based on source code instrumentation
 - Emacs integration
 - data value browser
- Future Work
 - debugging enhancements
 - Eclipse IDE integration
 - integration with static equation debugging, or with dynamic algorithmic debugging

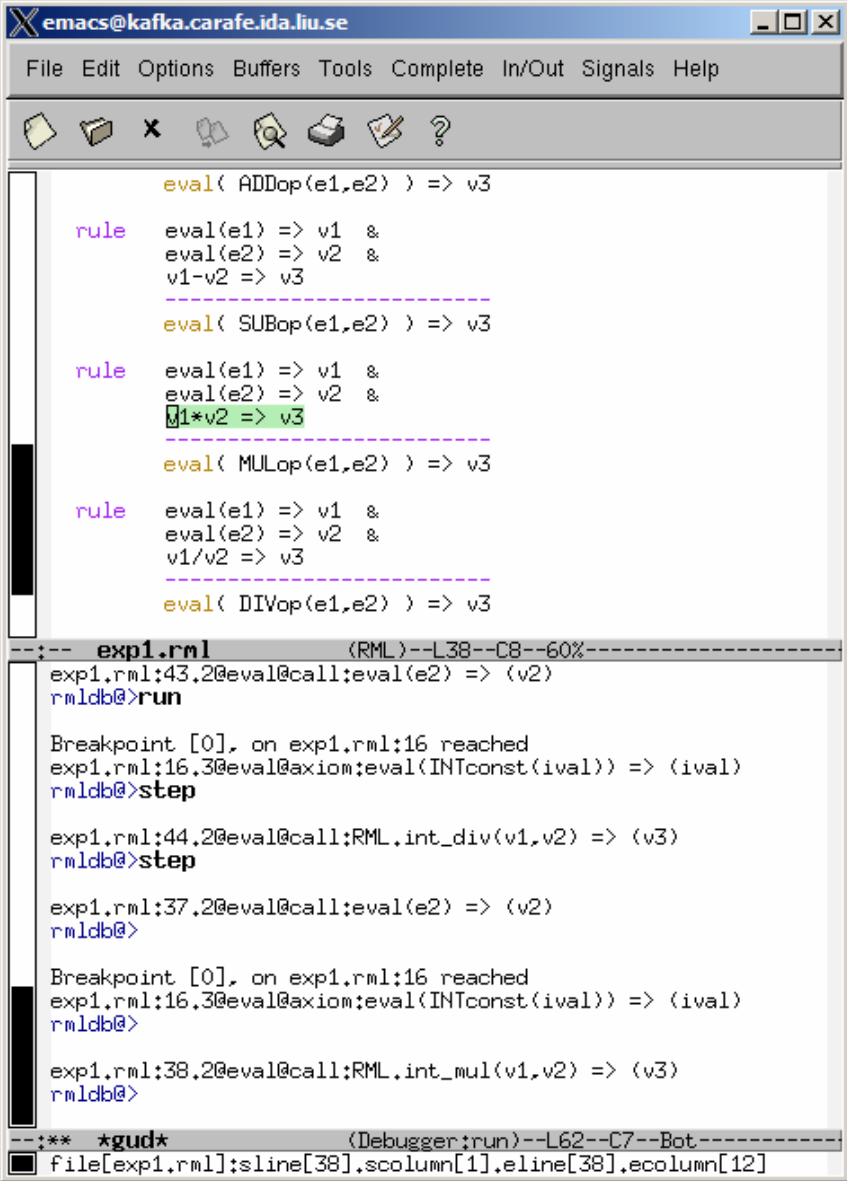
Demo

Thank you!
Questions?

<http://www.ida.liu.se/~pelab/rml>

Debugger Emacs integration

- Emacs integration within GUD mode
- Classical debugging facilities
 - breakpoints
 - stack trace
 - data inspection



The screenshot shows the Emacs editor window titled 'emacs@kafka.carafe.ida.liu.se'. The menu bar includes File, Edit, Options, Buffers, Tools, Complete, In/Out, Signals, and Help. The toolbar contains icons for file operations and search. The main text area displays RML code with several 'eval' and 'rule' blocks. The bottom panel shows the GUD debugger interface with a command history and execution log.

```
eval( ADDop(e1,e2) ) => v3

rule
  eval(e1) => v1 &
  eval(e2) => v2 &
  v1-v2 => v3
-----
eval( SUBop(e1,e2) ) => v3

rule
  eval(e1) => v1 &
  eval(e2) => v2 &
  1*v2 => v3
-----
eval( MULop(e1,e2) ) => v3

rule
  eval(e1) => v1 &
  eval(e2) => v2 &
  v1/v2 => v3
-----
eval( DIVop(e1,e2) ) => v3

--:-- exp1.rml (RML)--L38--C8--60%-----
exp1.rml:43,2@eval@call:eval(e2) => (v2)
rmlldb>run

Breakpoint [0], on exp1.rml:16 reached
exp1.rml:16,3@eval@axiom:eval(INTconst(ival)) => (ival)
rmlldb>step

exp1.rml:44,2@eval@call:RML.int_div(v1,v2) => (v3)
rmlldb>step

exp1.rml:37,2@eval@call:eval(e2) => (v2)
rmlldb>

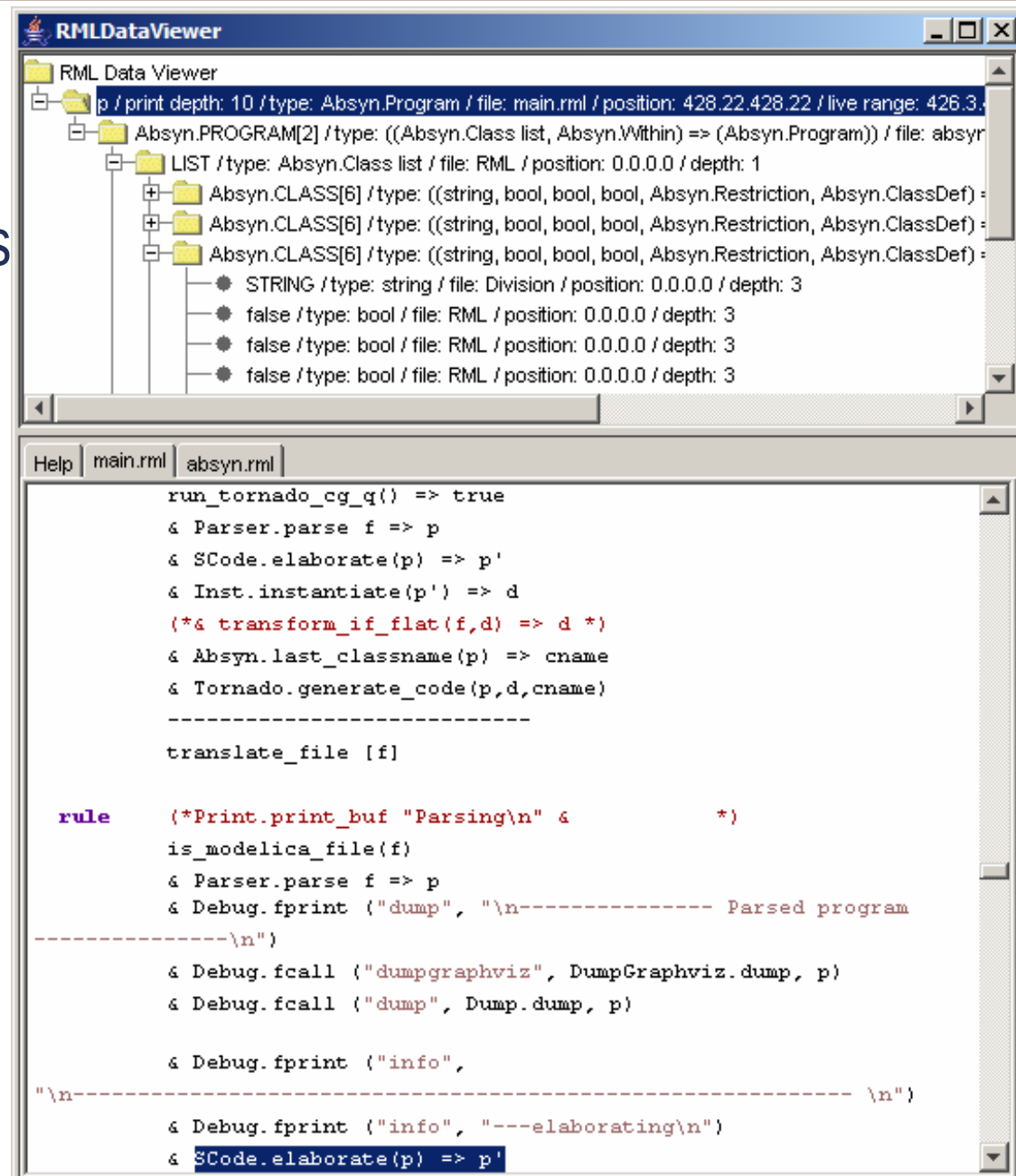
Breakpoint [0], on exp1.rml:16 reached
exp1.rml:16,3@eval@axiom:eval(INTconst(ival)) => (ival)
rmlldb>

exp1.rml:38,2@eval@call:RML.int_mul(v1,v2) => (v3)
rmlldb>

--:** *gud* (Debugger:run)--L62--C7--Bot-----
file[exp1.rml]:sline[38].scolumn[1].eline[38].ecolumn[12]
```

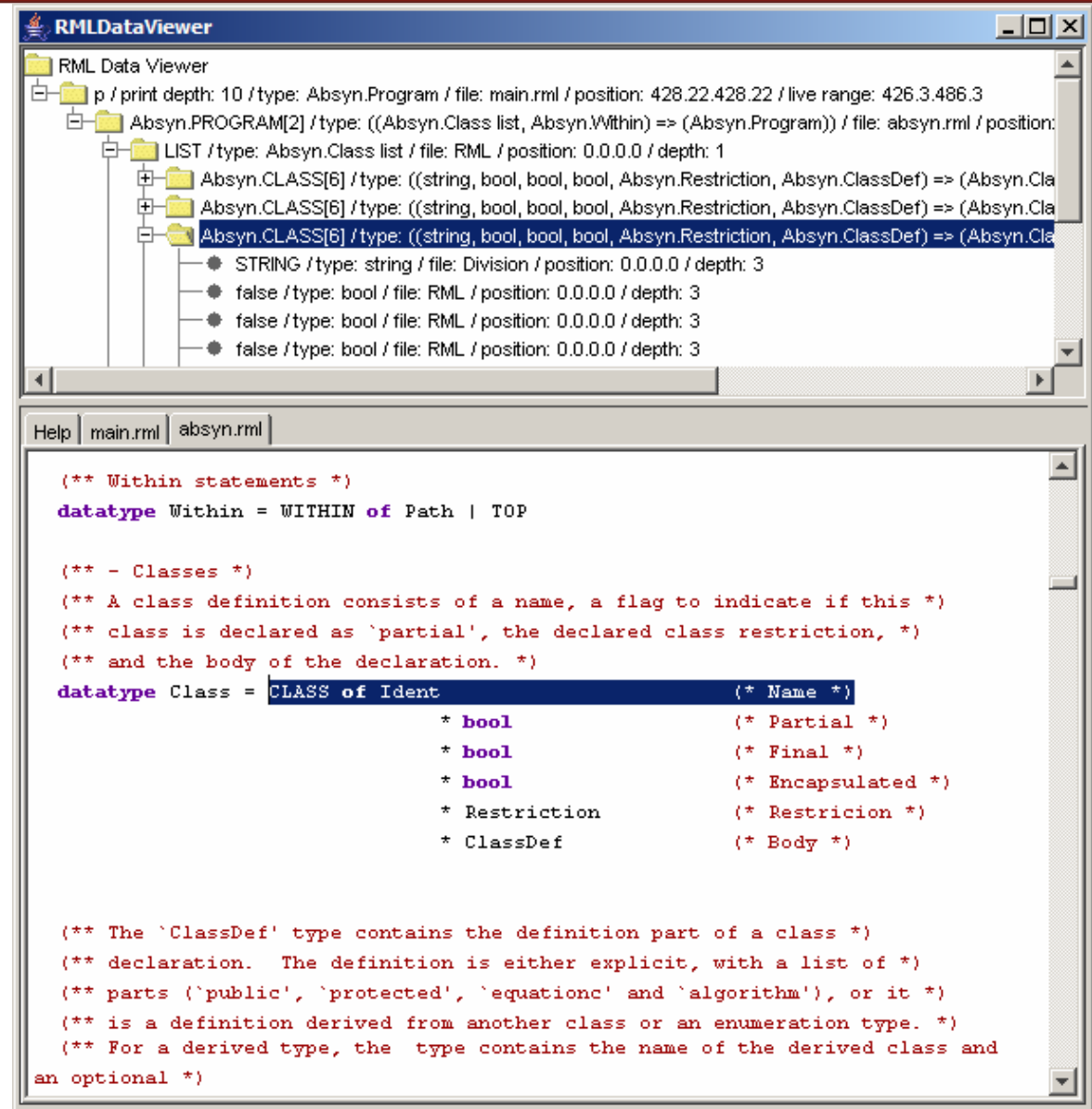
Data Value Browser

- Helps the user understand complex datatypes
- Shows the execution point



Data Value Browser

- Helps the user understand complex datatypes
- Presents datatype definitions



Ideas for Presentation

- Natural Semantics/Structured Operational Semantics common for specification of types systems, programming languages
- RML is a language with efficient implementation of NS, compiling to C
- The current work is first (to our knowledge) debugger for compiled NS
- Portable debugging framework based on code instrumentation and a small interface; can be adapted/reused
- Automatic mapping from data term to position in program where data was created
- Emphasis on practical debugging of large specifications
- Future work: Integration with static equation debugging, or with dynamic algorithmic debugging

RML example: the Exp language

■ Abstract syntax

```
datatype Exp = INTconst of int
            | PLUSop of Exp * Exp
            | SUBop of Exp * Exp
            | MULop of Exp * Exp
            | DIVop of Exp * Exp
            | NEGop of Exp
```

Exp: 10 * 12/3

