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

Outline

- 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.

Natural Semantics

$$\frac{H_1 \vdash T_1 : R_1 \quad \ldots \quad H_n \vdash T_n : R_n}{H \vdash T : R} \quad \textbf{if} \ \ \text{<cond>}$$

- Hi are hypotheses (environments)
- Ti are terms (pieces of abstract syntax)
- Ri are results (types, run-time values, changed environments)
- Hj |- Tj : Rj 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:

$$v \in Int$$

expressions (abstract syntax):

$$
\begin{aligned}
e \in Exp ::= {} & v \\
& \mid e1 + e2 \\
& \mid e1 - e2 \\
& \mid e1 * e2 \\
& \mid e1 / e2 \\
& \mid -e
\end{aligned}
$$

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

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

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

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

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

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

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

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 *)
```

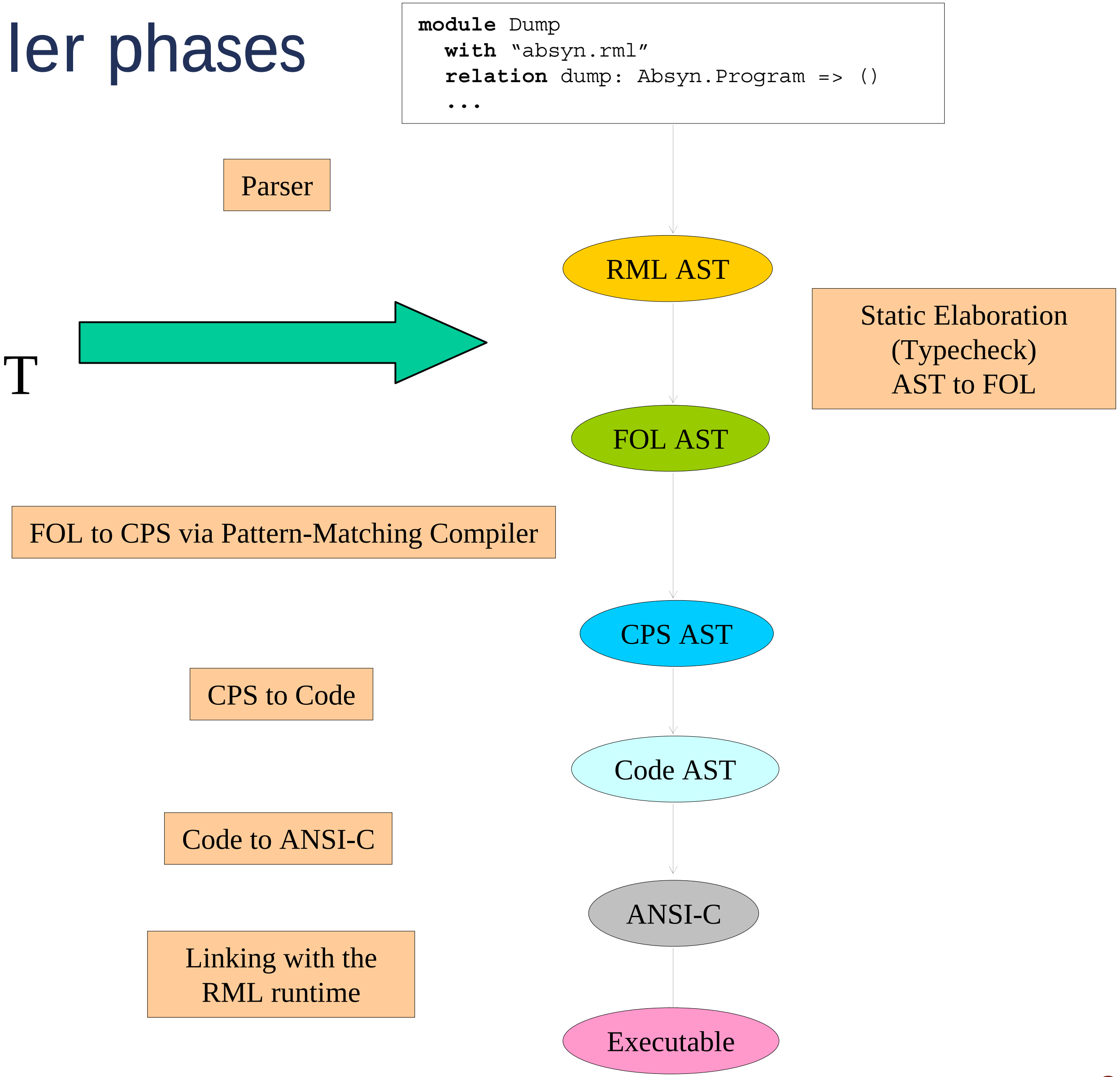
The Need for RML Debugging

- 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

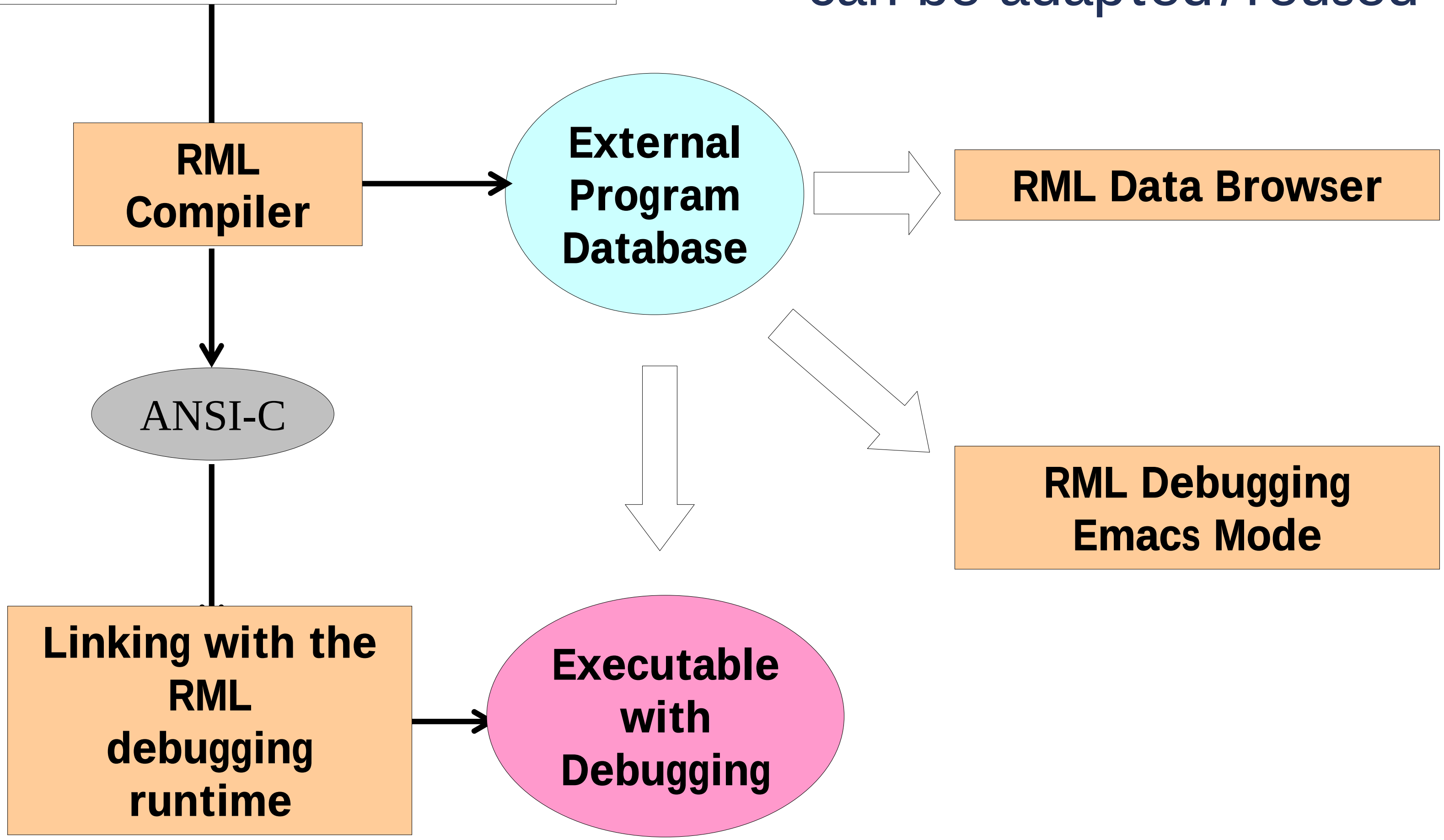


Instrumentation with debug nodes at the AST level

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 *)
```

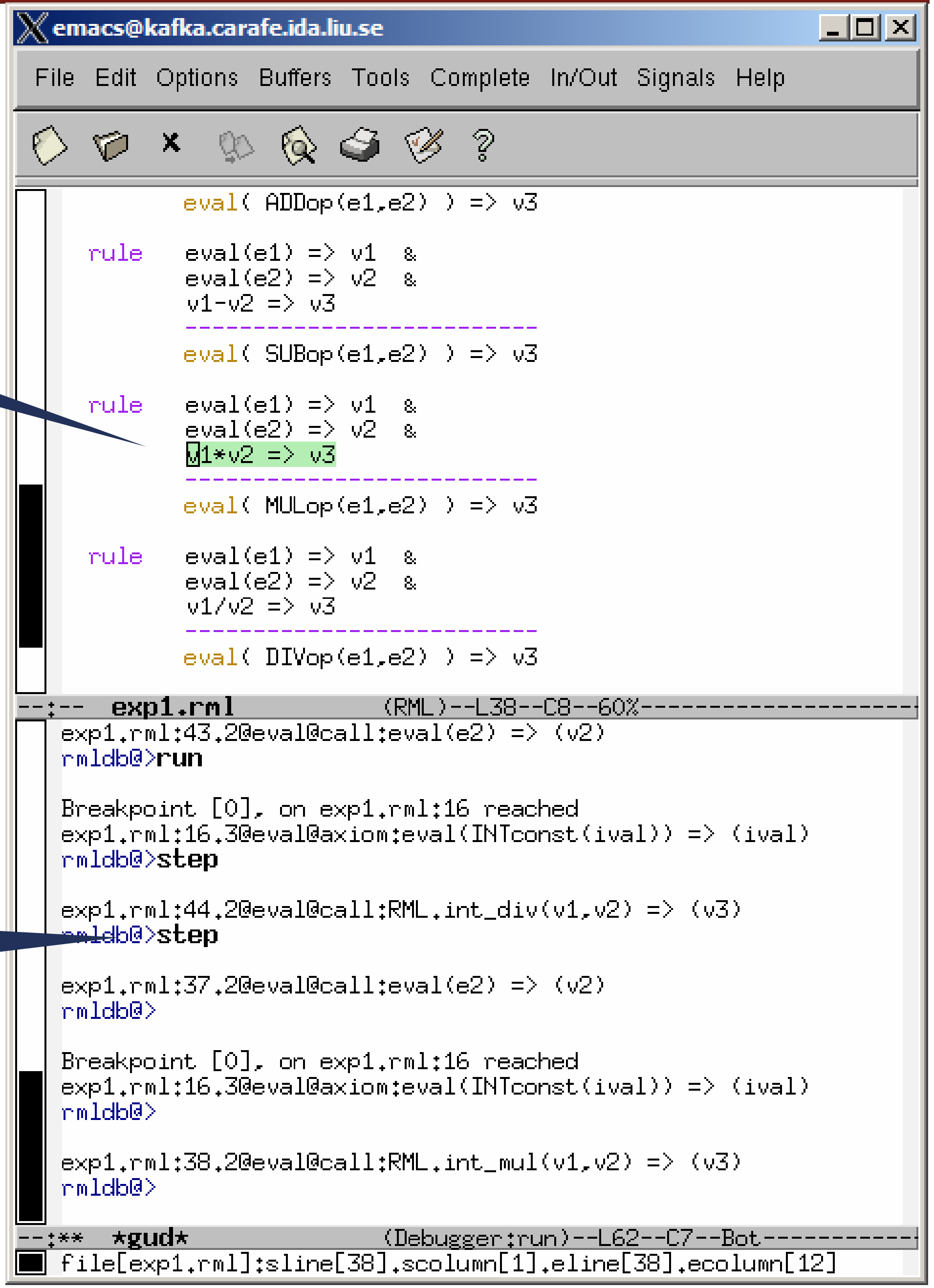
```
(* Evaluation semantics of Exp1 *)
relation eval: Exp => int  =
axiom eval(INTconst(ival)) => ival

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

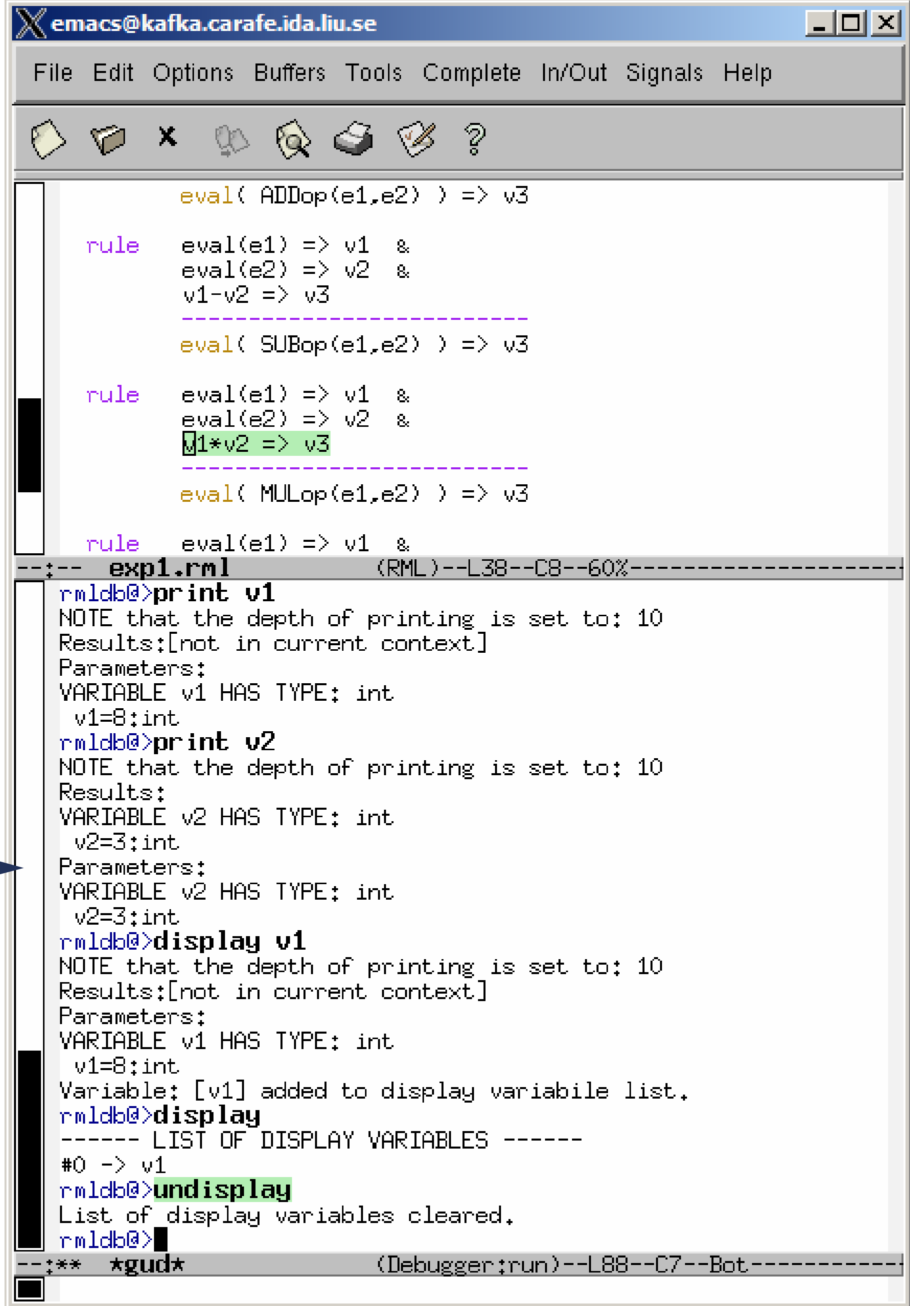
Stepping and Running



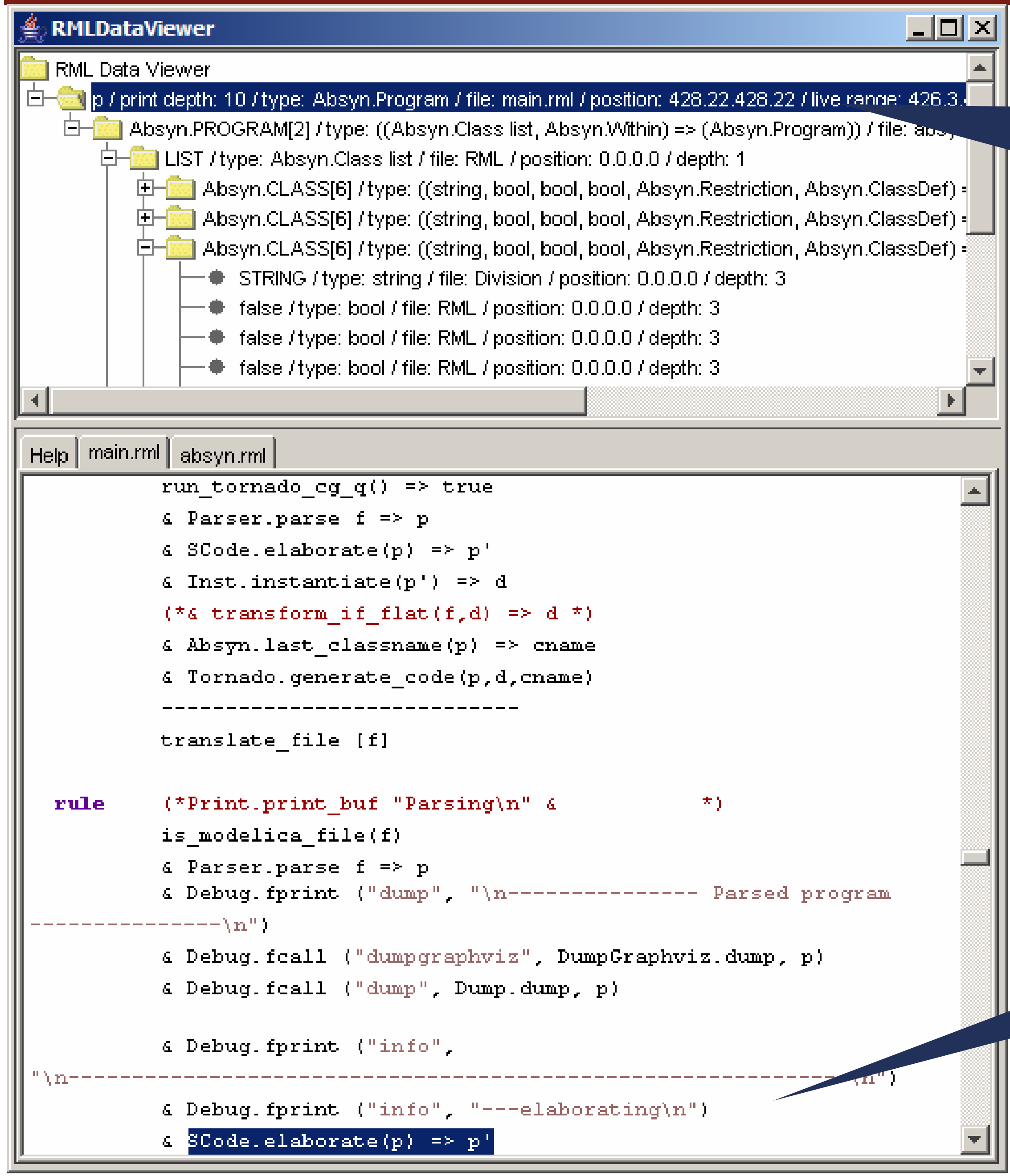
Debugger Functionality (2)

- 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



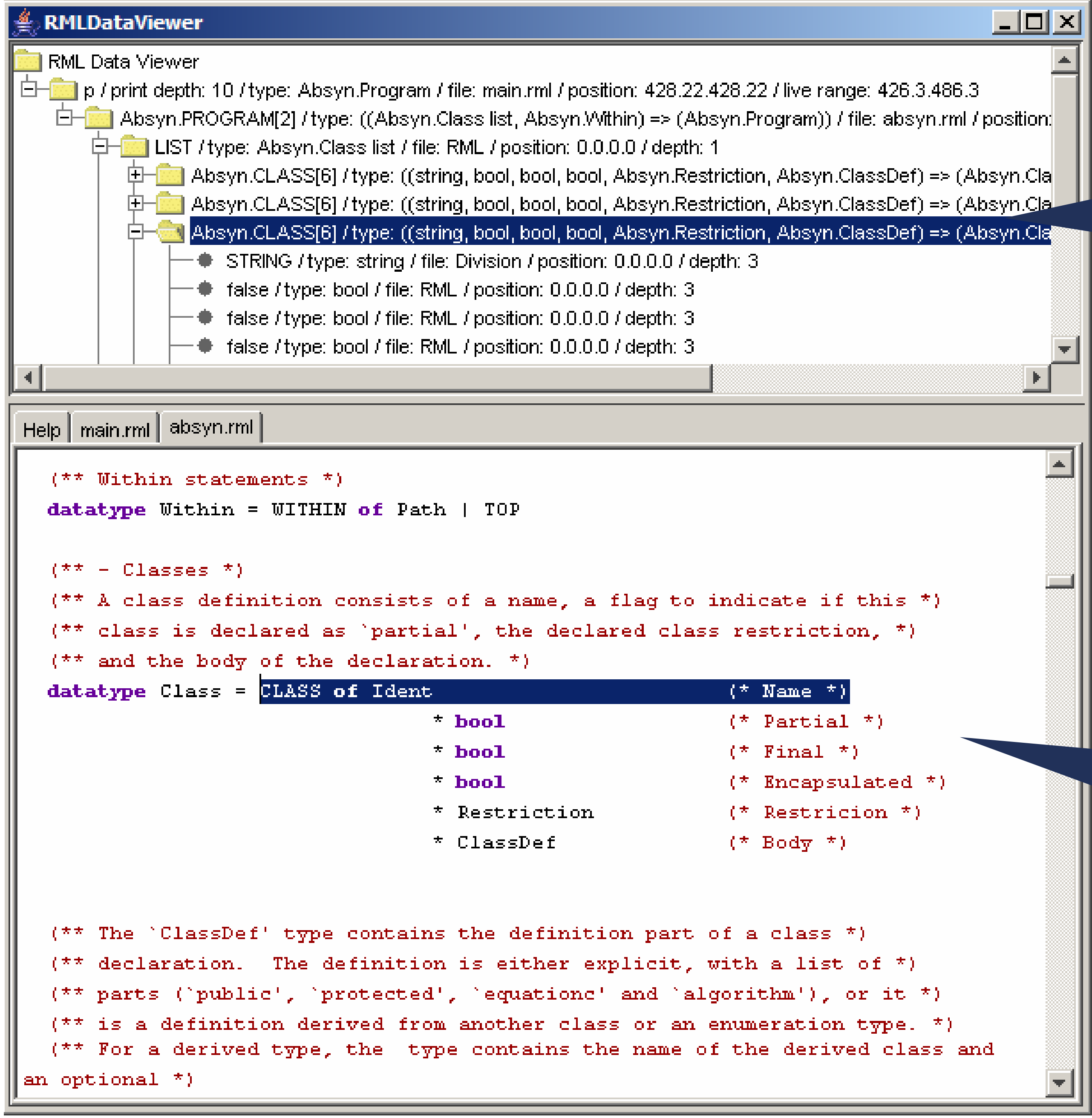
Browser for RML Data Structures (1)



Variable value inspection

Current Execution Point

Browser for RML Data Structures (2)



Data structure browsing

Data structure definition

Conclusions and Future work

- 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

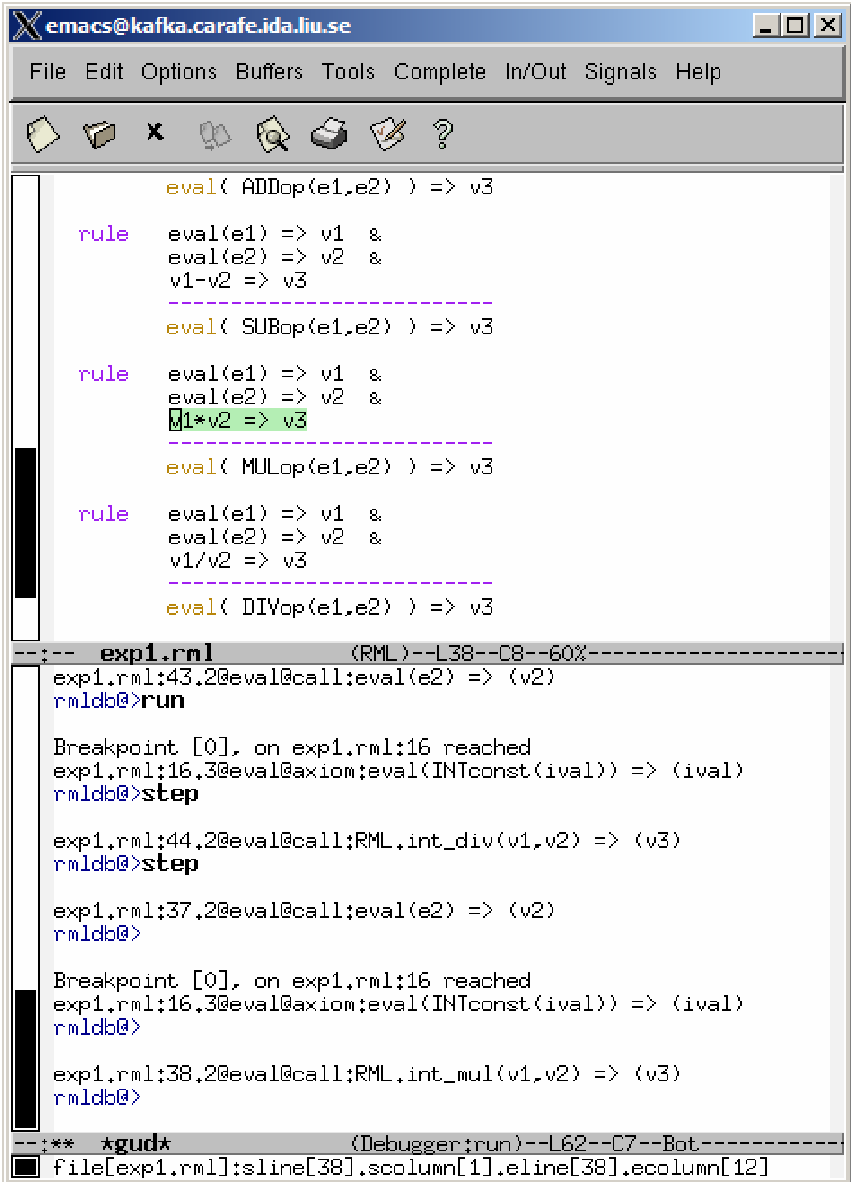
Demo

Thank you!
Questions?

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

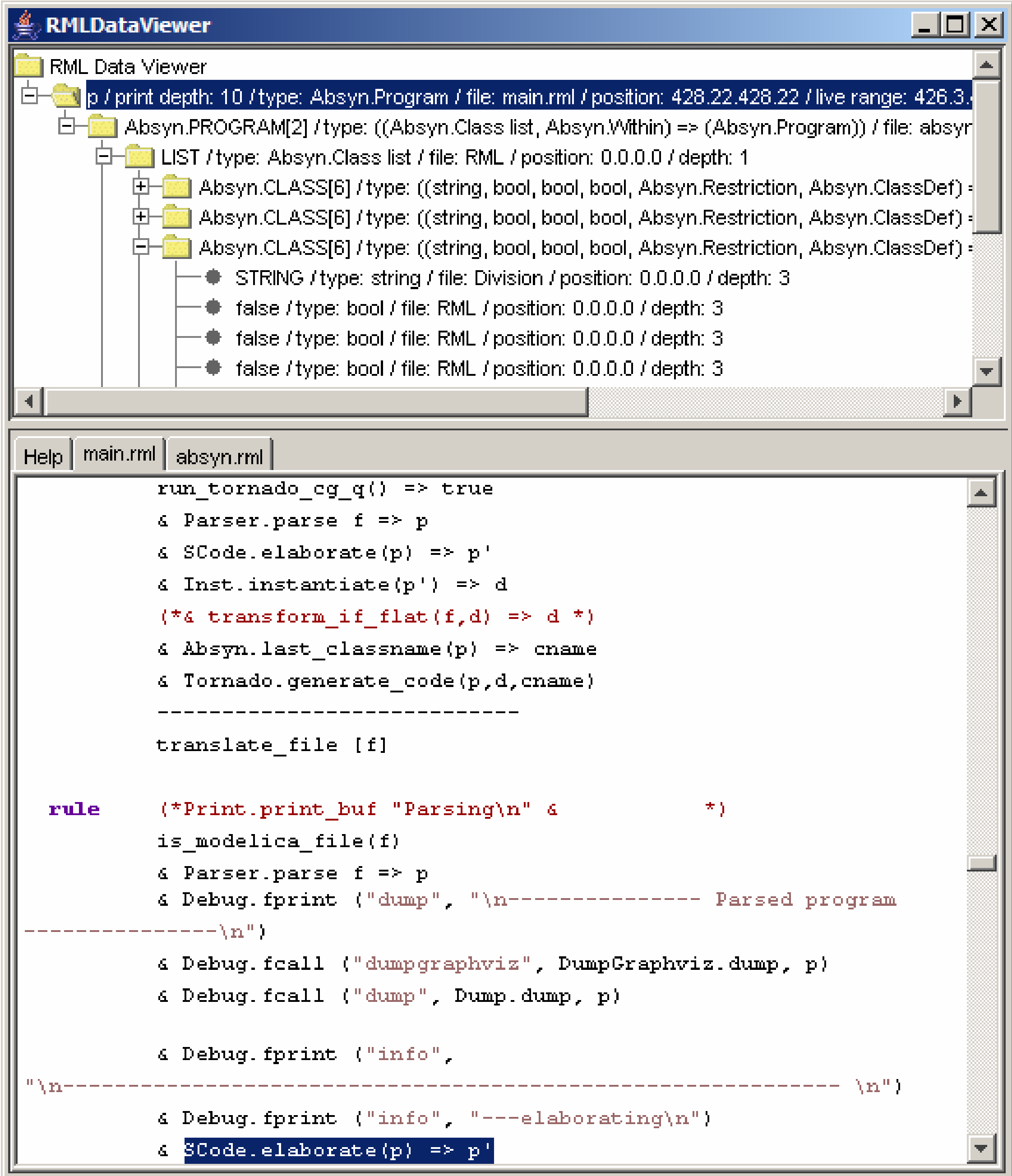
Debugger Emacs integration

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



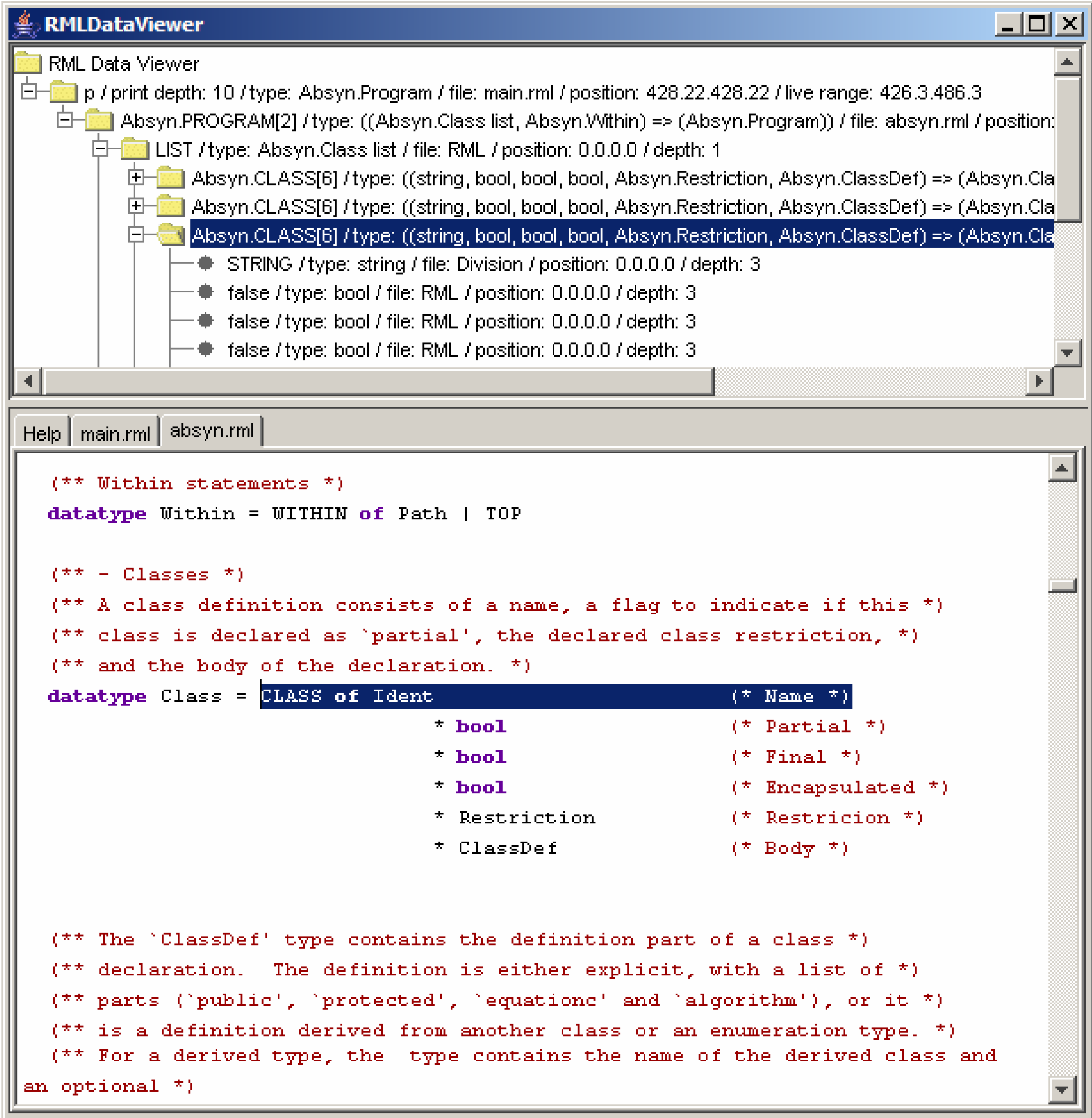
Data Value Browser

- 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`