OpenModelica Algorithmic Code Debugger for Modelica/MetaModelica

Adeel Asghar

Motivation

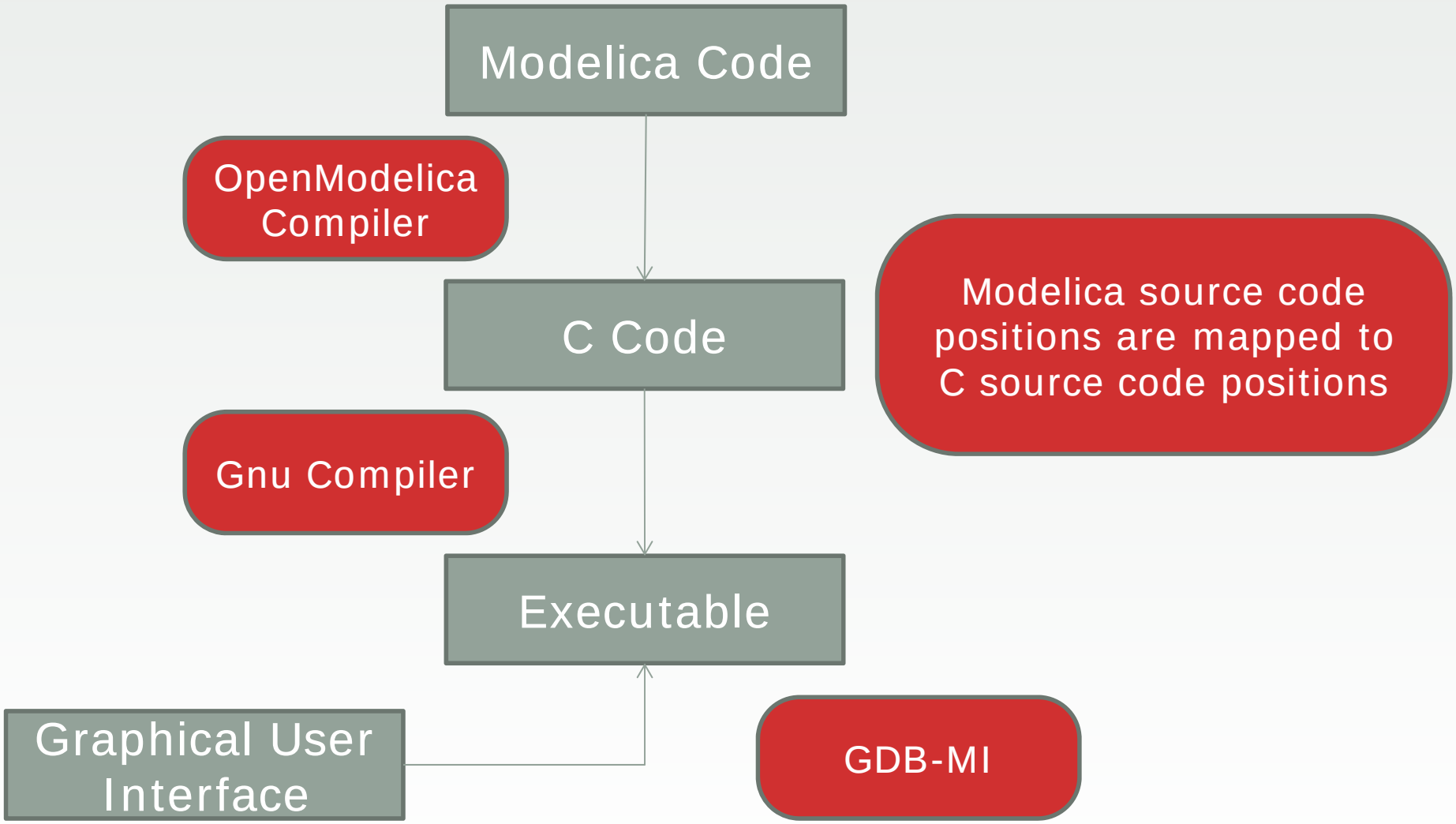
- Old Debugger
  - Slow Performance
  - High compilation time due to extra trace code
  - Slow execution due to tracing

Outline

- Section I - Architecture
  - How does it work?
  - Code Generation
- Section II – MDT-Debug & GDB-MI
  - Breakpoints Support
  - Program Execution
  - Events
- Section III – MDT-Debug Screens
- Section IV – Conclusion & Future Work

Section I
Architecture

How does it work?



Generate C Code

- Convert Modelica code to C source code by adding Modelica line number references.

HelloWorld.mo

```
1  function HelloWorld
2    input Real x;
3    output Real y;
4  algorithm
5    y := sin(x);
6  end HelloWorld;
```

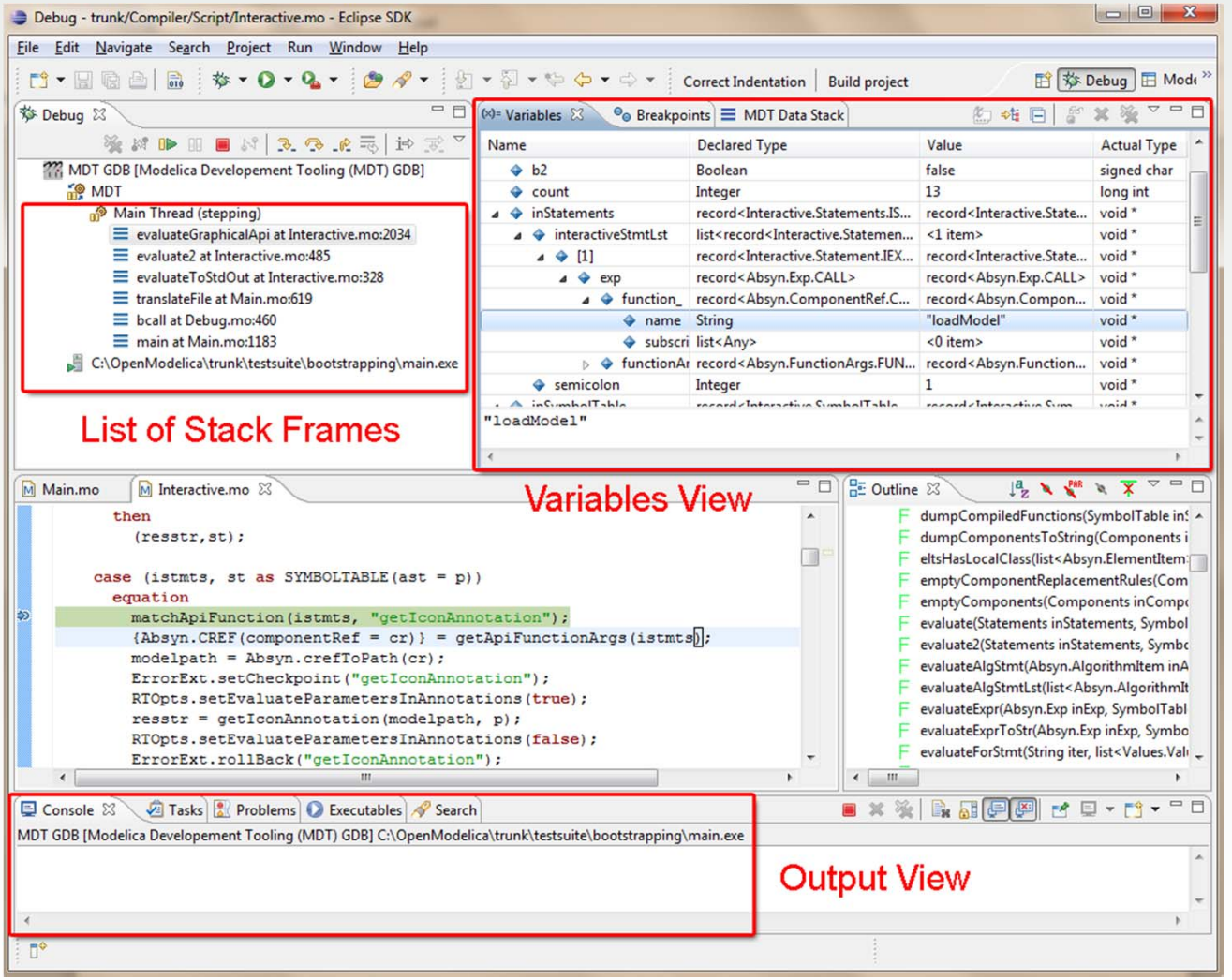
HelloWorld.conv.c

```
57 #line 29 "HelloWorld.c"
58   /* functionBodyRegularFunction: var inits */
59 #line 30 "HelloWorld.c"
60   /* functionBodyRegularFunction: body */
61 #line 5 "/c/workspace/HelloWorld/HelloWorld.mo"
62   tmp2 = sin(_x);
63 #line 5 "/c/workspace/HelloWorld/HelloWorld.mo"
64   _y = tmp2;
65 #line 35 "HelloWorld.c"
```

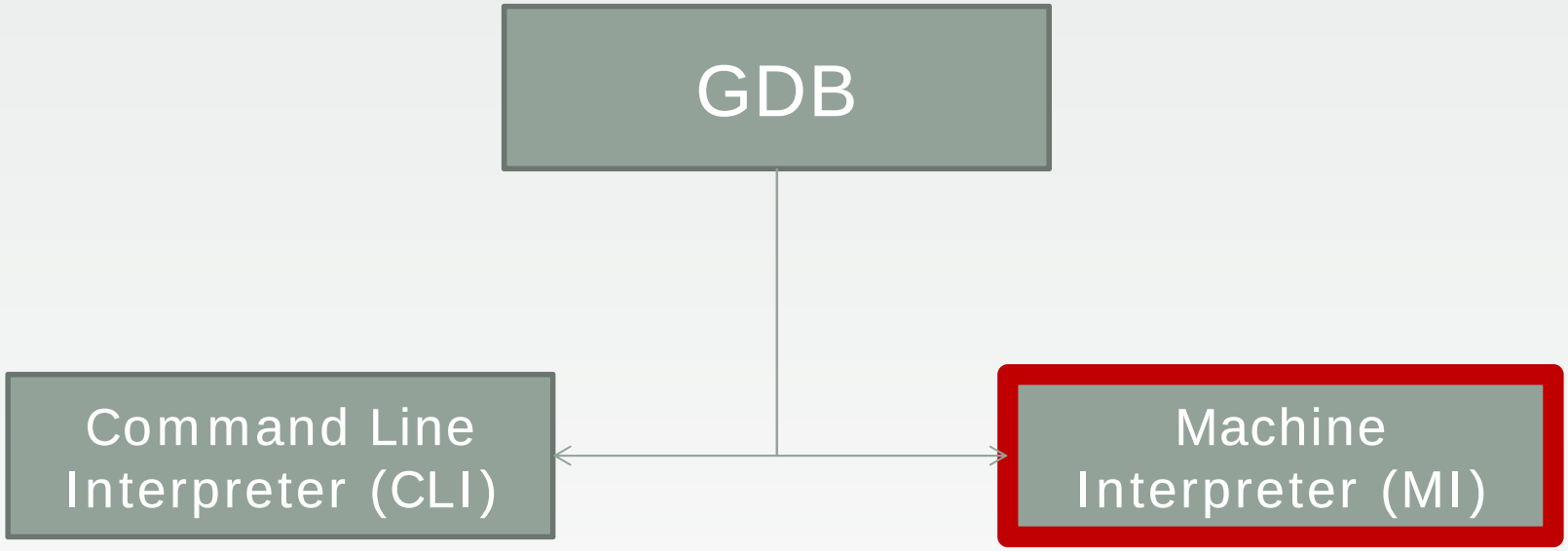
Section II
MDT-Debug & GDB-MI

MDT-Debug & GDB-MI

- MDT (Modelica Development Tooling) is the integrated development environment.
- The new debugger is implemented as a debug plugin within MDT.



MDT-Debug & GDB-MI (contd.)

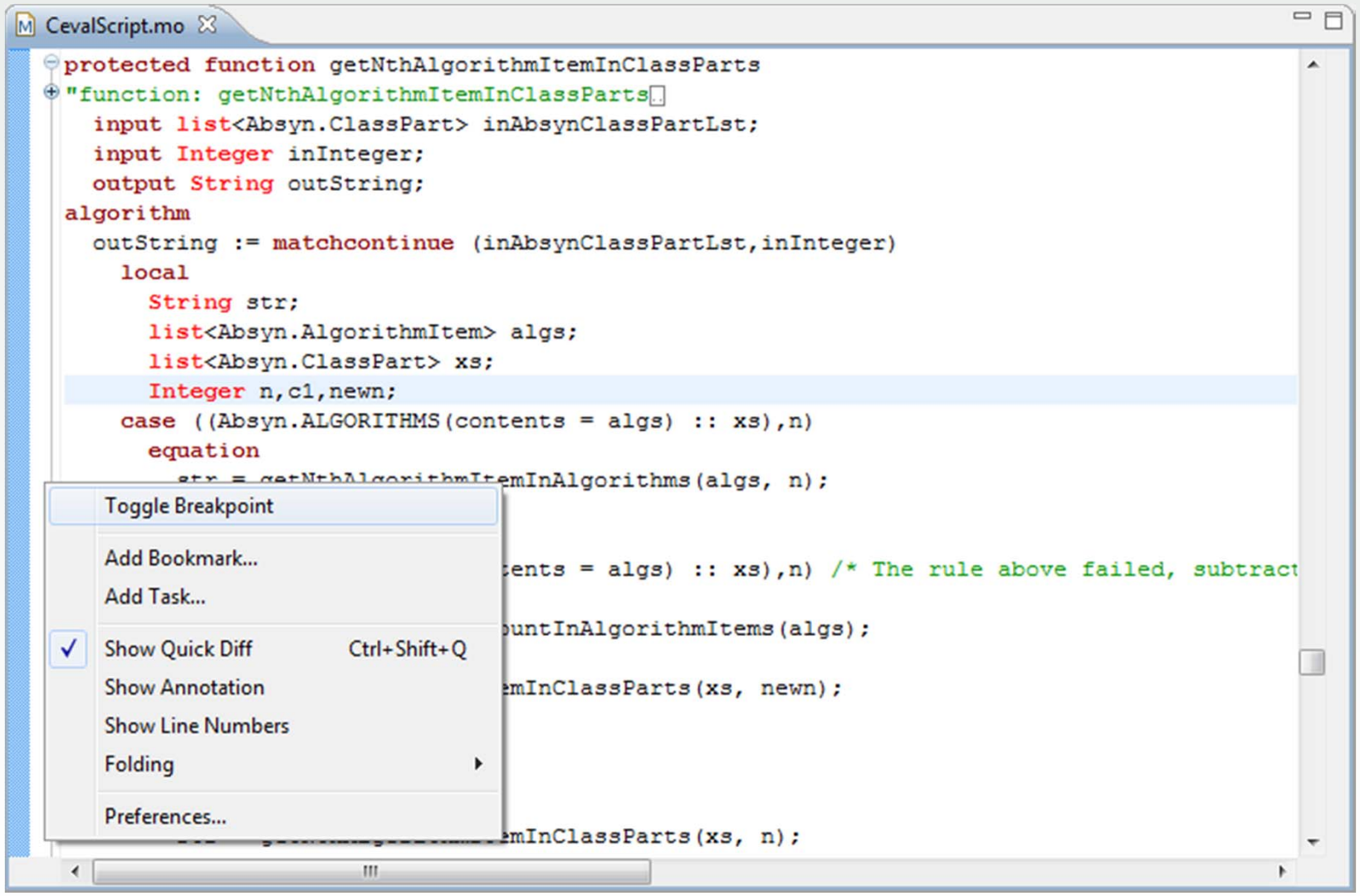


- CLI interpreter is more user friendly.
- MI interpreter is more convenient for detailed control of the execution at the low level.

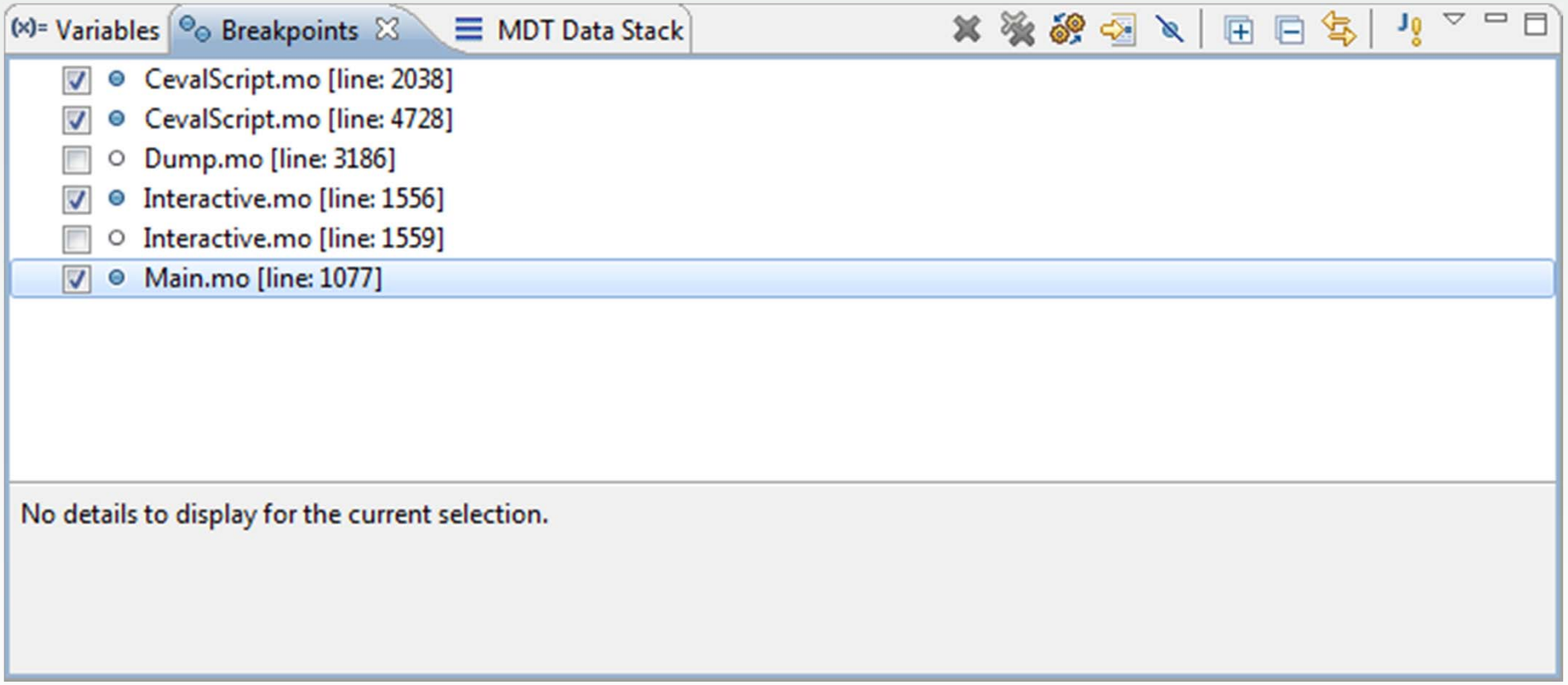
Breakpoints Support

- -break-insert –f <*filename:linenumber*>
  - -f create a pending breakpoint
- -break-enable <*breakpoint-number*>
- -break-disable <*breakpoint-number*>
- -break-delete <*breakpoint-number*>
- Other alternatives
  - -break-insert –f *function* (not supported)
  - -break-insert –f *filename:function* (not supported)

Breakpoints Support (contd.)



Breakpoints Support (contd.)



Program Execution

- GDB command types
  - Synchronous commands
  - Asynchronous commands
- The program execution commands are asynchronous
  - -exec-run runs the program
  - -exec-continue continue the program
  - -exec-next performs the step over
  - -exec-step performs the step into
  - -exec-finish executes the function and return to the function call

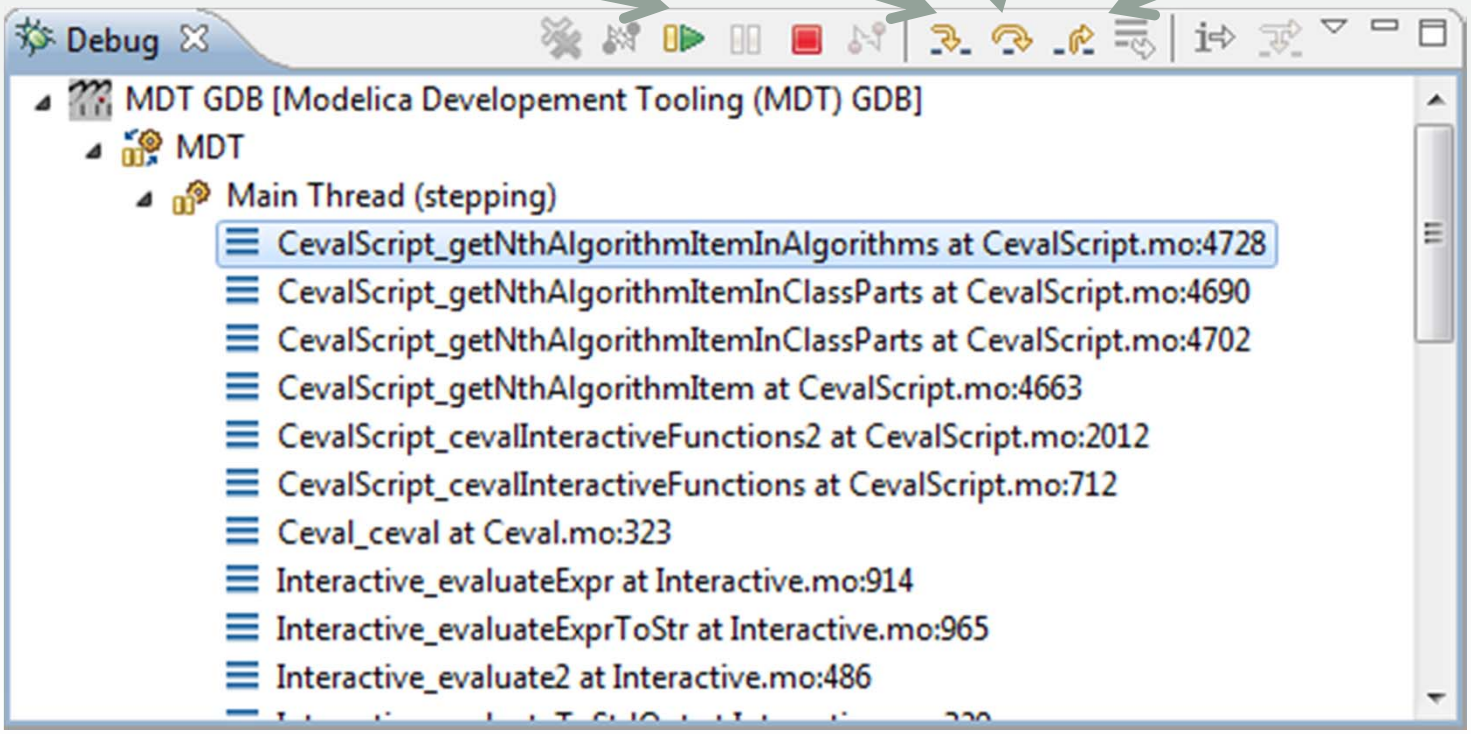
Program Execution (contd.)

-exec-continue

-exec-step

-exec-next

-exec-finish

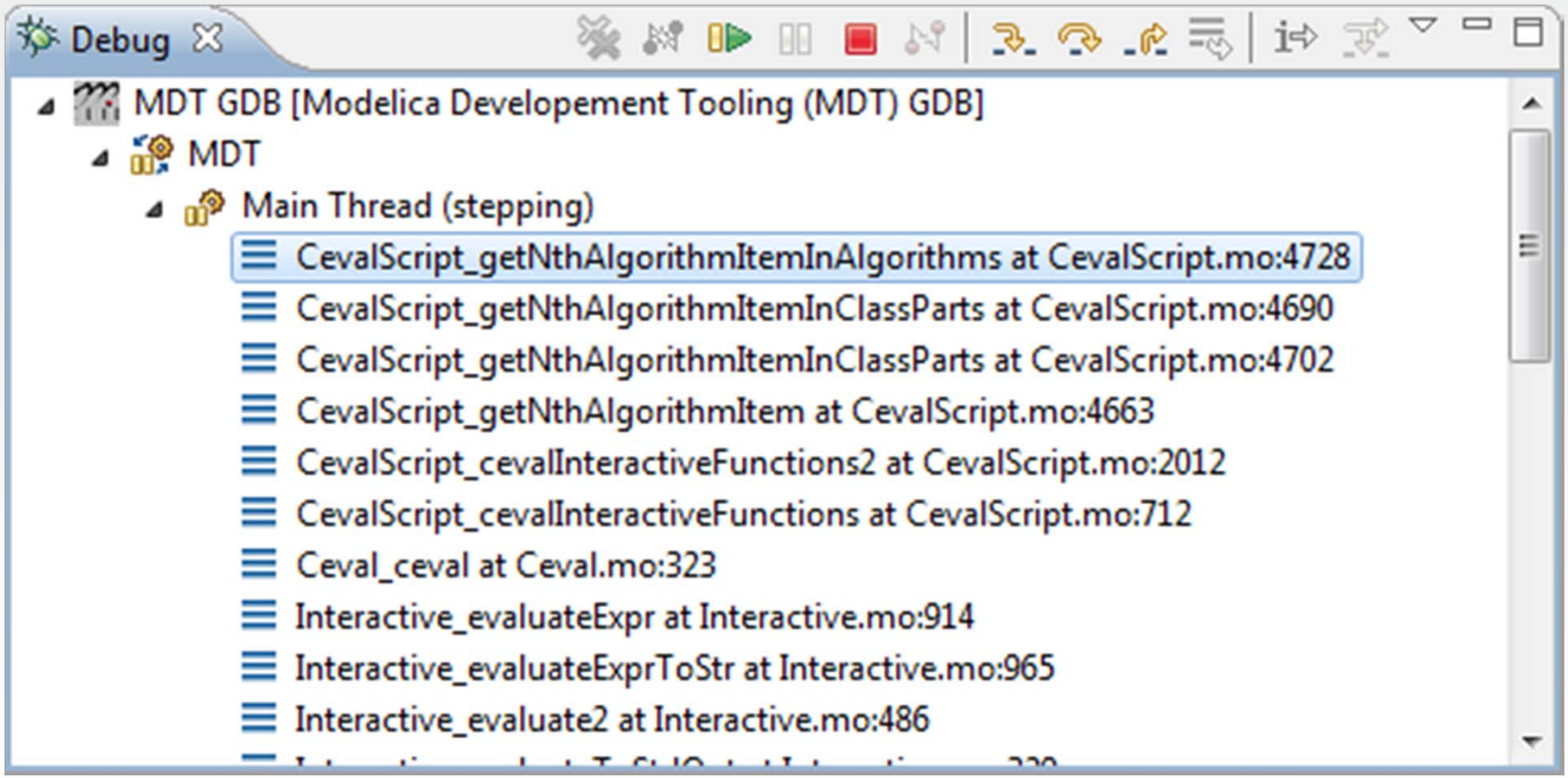


Events

- GDB raise events
  - For asynchronous commands
  - For notifying program state
- For example
  - breakpoint-hit when a breakpoint is hit
  - end-stepping-range when a step into or step over operations are finished
  - function-finished when a step return operation is finished
  - signal-received e.g SIGSEGV

Section III
MDT-Debug Screens

Stack Frames



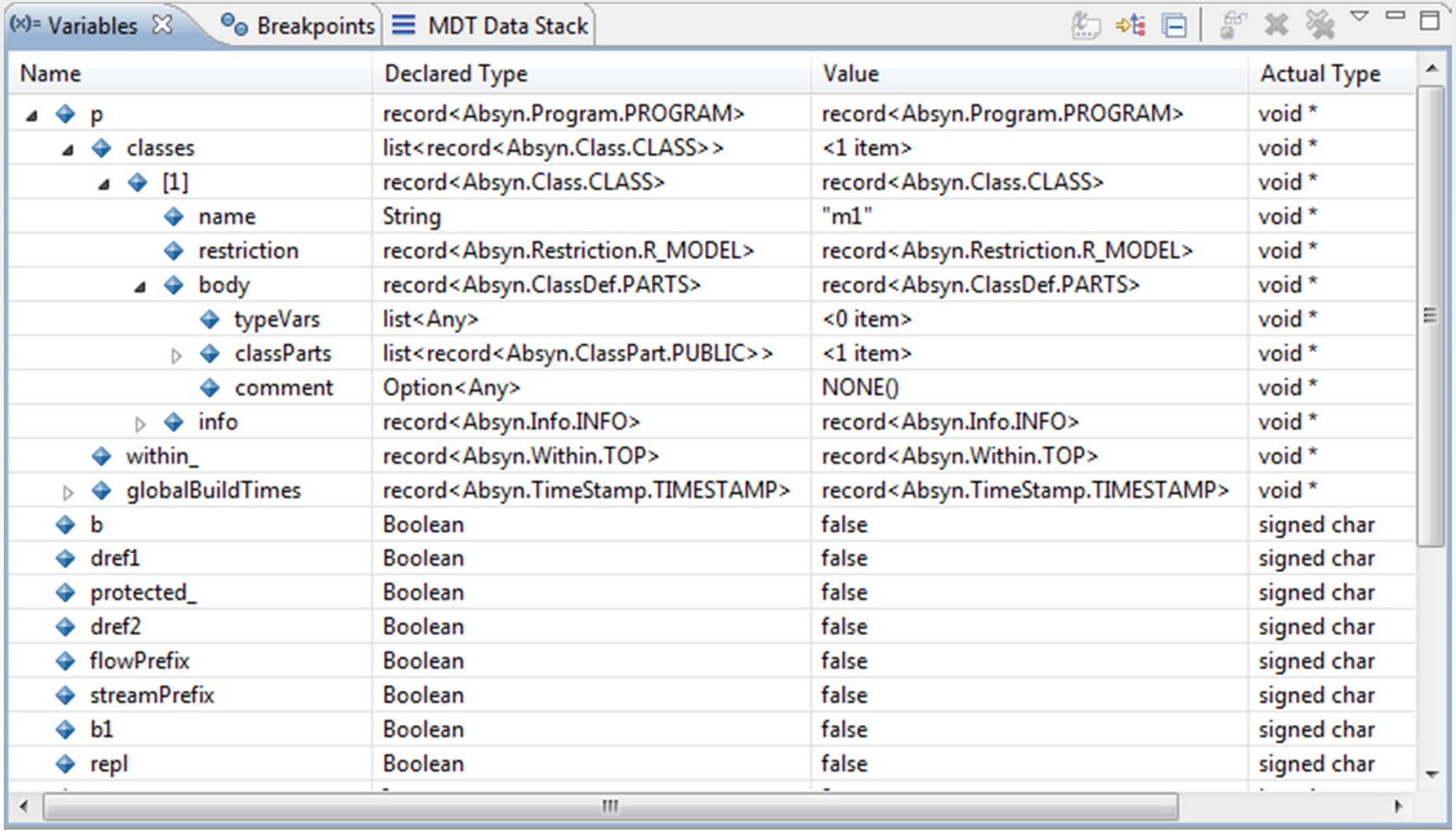
Stack Frames (contd.)

- Whenever a program execution is stopped e.g because of any event.
- A stack of frames is created.
- -stack-list-frames (returns a list of frames).
- Filter C files.

HelloWorld.conv.c

```
57 #line 29 "HelloWorld.c"
58   /* functionBodyRegularFunction: var inits */
59 #line 30 "HelloWorld.c"
60   /* functionBodyRegularFunction: body */
61 #line 5 "/c/workspace/HelloWorld/HelloWorld.mo"
62   tmp2 = sin(_x);
63 #line 5 "/c/workspace/HelloWorld/HelloWorld.mo"
64   _y = tmp2;
65 #line 35 "HelloWorld.c"
```

Variables View



Name	Declared Type	Value	Actual Type
p	record<Absyn.Program.PROGRAM>	record<Absyn.Program.PROGRAM>	void *
classes	list<record<Absyn.Class.CLASS>>	<1 item>	void *
[1]	record<Absyn.Class.CLASS>	record<Absyn.Class.CLASS>	void *
name	String	"m1"	void *
restriction	record<Absyn.Restriction.R_MODEL>	record<Absyn.Restriction.R_MODEL>	void *
body	record<Absyn.ClassDef.PARTS>	record<Absyn.ClassDef.PARTS>	void *
typeVars	list<Any>	<0 item>	void *
classParts	list<record<Absyn.ClassPart.PUBLIC>>	<1 item>	void *
comment	Option<Any>	NONE()	void *
info	record<Absyn.Info.INFO>	record<Absyn.Info.INFO>	void *
within_	record<Absyn.Within.TOP>	record<Absyn.Within.TOP>	void *
globalBuildTimes	record<Absyn.TimeStamp.TIMESTAMP>	record<Absyn.TimeStamp.TIMESTAMP>	void *
b	Boolean	false	signed char
dref1	Boolean	false	signed char
protected_	Boolean	false	signed char
dref2	Boolean	false	signed char
flowPrefix	Boolean	false	signed char
streamPrefix	Boolean	false	signed char
b1	Boolean	false	signed char
repl	Boolean	false	signed char

Section IV
Conclusion & Future Work

Conclusion and Future Work

- The debugger supports extended Modelica (MetaModelica) algorithmic code.
- Operates efficiently on large algorithmic code. (Tested on OpenModelica compiler with more than 100 000 lines of code).
- Only MetaModelica datatypes including primitive Modelica types Integer, Real, Boolean and String are supported.
- The debugger needs to be extended to support all Modelica datatypes (arrays, enumerations and records).


Thank You
Mahalo
Kiitos
Tack
Toda
Thanks
Grazie
Obrigado
Takk
Merci
Gracias