*15*

## Debugging DSLs

*Debugging is relevant in two ways in the context of DSLs and language workbenches. First, the DSL developer may want to debug the definition of a DSL, including constraints, scopes or transformations and interpreters. Second, programs written in the DSL may have to be debuggable by the end user. We address both aspects in this chapter.*

### 15.1 Debugging the DSL Definition

Debugging the definition of the DSL boils down to a language workbench providing a debugger for the languages used for language definition. In the section we look at understanding and debugging the structure and concrete syntax, the definition of scopes, constraints and type systems, as well as debugging interpreters and transformations.

#### 15.1.1 Understanding and Debugging the Language Structure

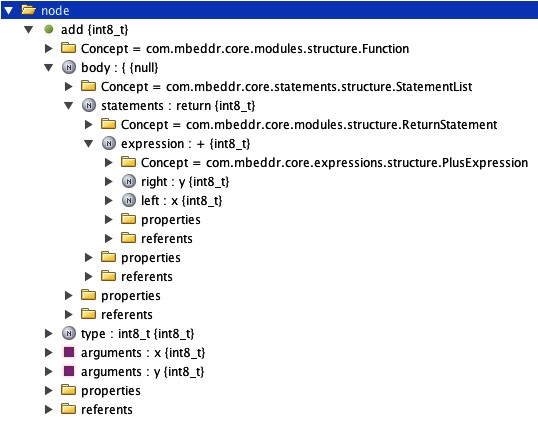
In parser-based systems, the transformation from text to the AST performed by the parser is itself a non-trivial process and has a potential for errors. Debugging the parsing process can be important.

|  |  |
| --- | --- |
| *Xtext* Xtext uses ANTLR under the hood. In other words, an ANTLR grammar is generated from the Xtext grammar which |  |
| performs the actual parsing2. So understanding and debug- |  |
| ging the Xtext parsing process means understanding and debugging the ANTLR parsing process. |  |

1

|  |  |  |  |
| --- | --- | --- | --- |
| There are two ways to do this. First, since ANTLR generates a Java-based parser, you can debug the execution of ANTLR | | |  |
| (as part of Xtext) itself3. Second, you can have Xtext generate a | | |  |
| debug grammar, which contains no action code (so it does not populate the AST). However, it can be used to debug the pars- | | |  |
| ing process with ANTLRWorks4. ANTLRWorks comes with an interactive debugger for ANTLR grammars.  *MPS* In MPS there is no transformation from text to the AST since it is a projectional editor. However, there are still means of helping to better understand the structure of an existing program. For example, any program element can be inspected in the *Explorer*. Fig. 15.1 shows the explorer contents for a trivial C function: | | |  |
| int8 add(int8 x, int8 y) {  **return** x + y;  } |

.



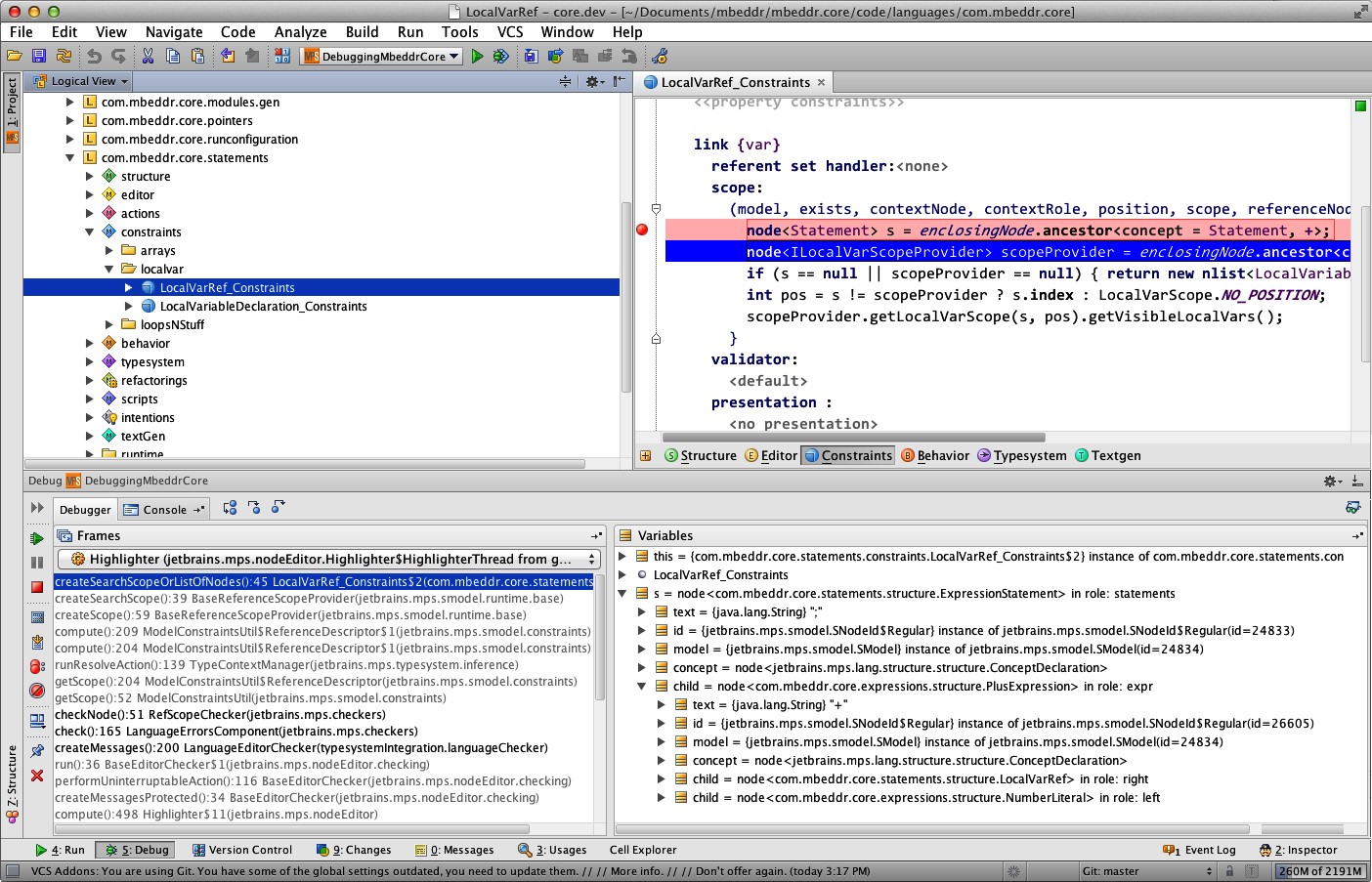
MPS provides similar support for understanding the projection rules. For any program node MPS can show the cell structure as a tree. The tree contains detailed information about the cell hierarchy, the program element associated with each cell, and the properties of the cell (height, width, etc.).

#### 15.1.2 Debugging Scopes, Constraints and Type Systems

Depending on the level of sophistication of a particular language, a lot of non-trivial behavior can be contained in the code that determines scopes, checks constraints or computes

types. In fact, in many languages, these are the most sophisticated aspects of language definition. Consequently, there is a need for debugging those.

|  |  |
| --- | --- |
|  |  |
| tem rules and all IDE aspects. Consequently, all these aspects |  |
| can be debugged by using a Java debugger. To do this, you can simply launch the Eclipse Application that contains the language and editor in debug mode and set breakpoints at the relevant locations6. |  |

 *Xtext* In Xtext all aspects of a language except the grammar and the abstract syntax are defined via Java5 programs using Xtext APIs. This includes scopes, constraints, type sys-

*MPS* MPS comes with a similar facility, in the sense that a second instance of MPS can be run "inside" the current one. This inner instance can be debugged from the outer one. This approach can be used for all those aspects of MPS-defined languages that are defined in terms of the BaseLanguage, MPS’ version of Java. For example, scopes can be debugged this way: in Fig. 15.2 we debug the scope for a **LocalVariableRef**.

|  |
| --- |
| A related feature of MPS is the ability to analyze exception stack traces. To implement a language, MPS generates Java code from language definitions and then executes this Java code. If an exception occurs in language implementation code it produces a Java stack trace. This stack trace can be pasted into a dialog in MPS. MPS then produces a version of the stack trace in which the code locations in the stack trace (which are relative to the generated Java) have been translated to locations in the DSL definition (expressed in Base Language). The locations can be clicked directly, opening the MPS editor at the respective location. |

Relative to the type system, MPS comes with two dedicated debug facilities (beyond debugging a new instance of MPS inside MPS mentioned above). First, pressing **Ctrl-Shift-T** on any program element will open a dialog that shows the type of the element. If the element has a type system error, the dialog also lets the user navigate to the rule that reported the error. The second facility is much more sophisticated. For any program node, MPS can show the *type system trace* (Fig. 15.3 shows a simple example). Remember how the MPS type system relies on a solver to solve the type system equations associated with program elements (specified by the language developer for the respective concepts). This means that each program has an associated set of type system equations, which contain explicitly

specified types as well as type variables. The solver tries to find type values for these variables such that all type system equations become true. The type system trace essentially visualizes the state of the solver, including the values it assigns to type variables, as well as which type system rules are applied to which program element[[1]](#footnote-1).

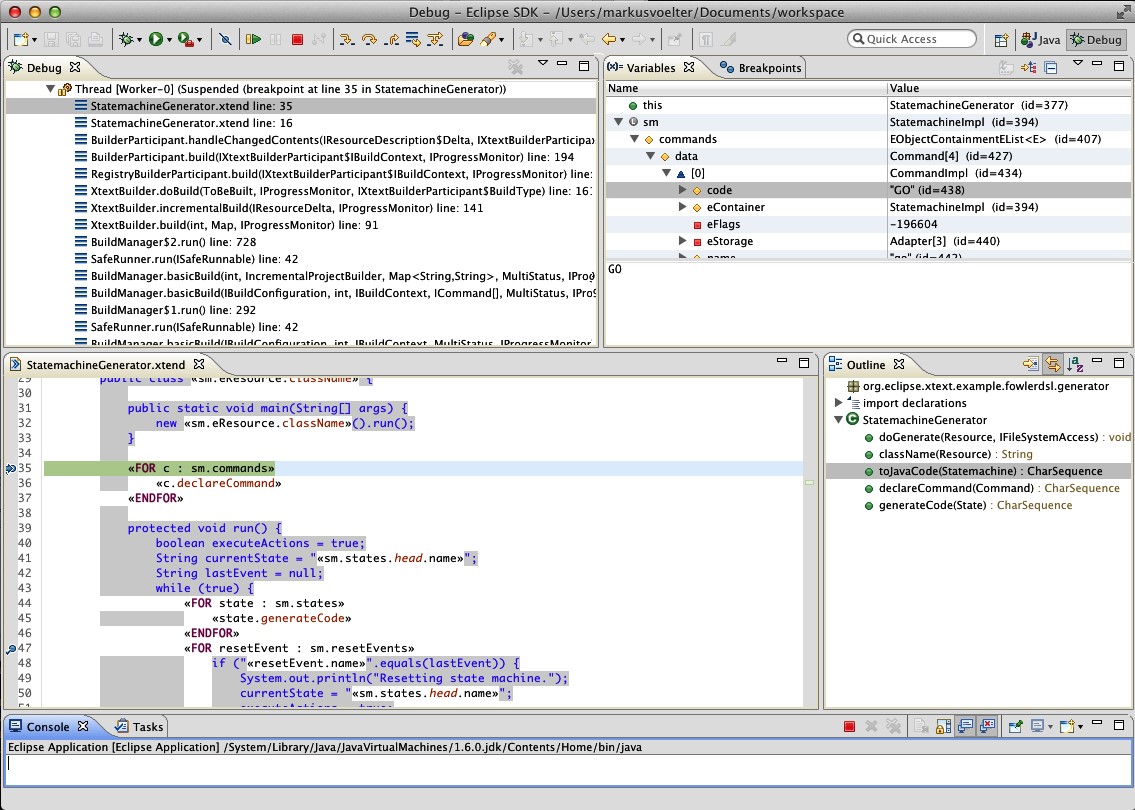
#### 15.1.3 Debugging Interpreters and Transformations

Debugging an interpreter is simple: since an interpreter is just a program written in some programming language that processes and acts on the DSL program, debugging the interpreter simply uses the debugger for the language in which the interpreter is written (assuming there is one)8.

|  |  |
| --- | --- |
| Debugging transformations and generators is typically not quite as trivial, for two reasons. First, transformations and generators are typically written in DSLs optimized for this task. So a specialized debugger is required. Second, if multi-step transformations are used, the intermediate models may have to be accessible, and it should be possible to trace a particular element through the multi-step transformation.  *Xtext* Xtext can be used together with any EMF-compatible code generator or transformation engine. However, since Xtext ships with Xtend, we look at debugging Xtend transformations. Model-to-model transformations and code generators in Xtend look very similar: both use Xtend to navigate over and query the model, based on the AST. The difference is that, as a side effect, model-to-model transformations create new model elements and code generators create strings, typically using rich strings (aka template expressions).  Any Xtend program can be debugged using Eclipse "out of the box". In fact, you can debug an Xtend program either on the |  |
| Xtend level or on the level of the generated Java source9. Since |  |
| interpreters and generators are just regular Xtend programs, they can be debugged in this way as well. Fig. 15.4 shows an example of debugging a template expressions.  Xtend is a fundamentally an object-oriented language, so the step-through metaphor for debuggers works. If Xtend is used for code generation or transformation, debugging boils down |  |
| to stepping through the code that builds the target model10. |  |

*MPS* In MPS, working with several chained transformations is normal, so MPS provides support for debugging the transformation process. This support includes two ingredients. The first one is showing the mapping partitioning. For any given model, MPS automatically computes the order in which transformations are executed, based on the relative priorities specified for the generators involved. The mapping partitioning reports the overall transformation schedule to the user.

This is useful in understanding which transformations are executed in which order, and in particular, to debug transformation priorities. Let us investigate a simple example C program that contains a message definition and a **report** statement. The



**report** statement is transformed to **printf** statements:

|  |
| --- |
| **module** Simple **imports** nothing {  **message list** messages {  INFO aMessage() **active**: something happened  }  **exported** int32 main(int32 argc, int8\*[] argv) { **report** (0) messages.aMessage(); **return** 0;  }  } |

Below is the mapping configuration for this program:

[ 1 ] com.mbeddr.core.modules.gen.generator.template.main.removeCommentedCode [ 2 ]

com.mbeddr.core.util.generator.template.main.reportingPrintf [ 3 ] com.mbeddr.core.buildconfig.generator.template.main.desktop com.mbeddr.core.modules.gen.generator.template.main.main

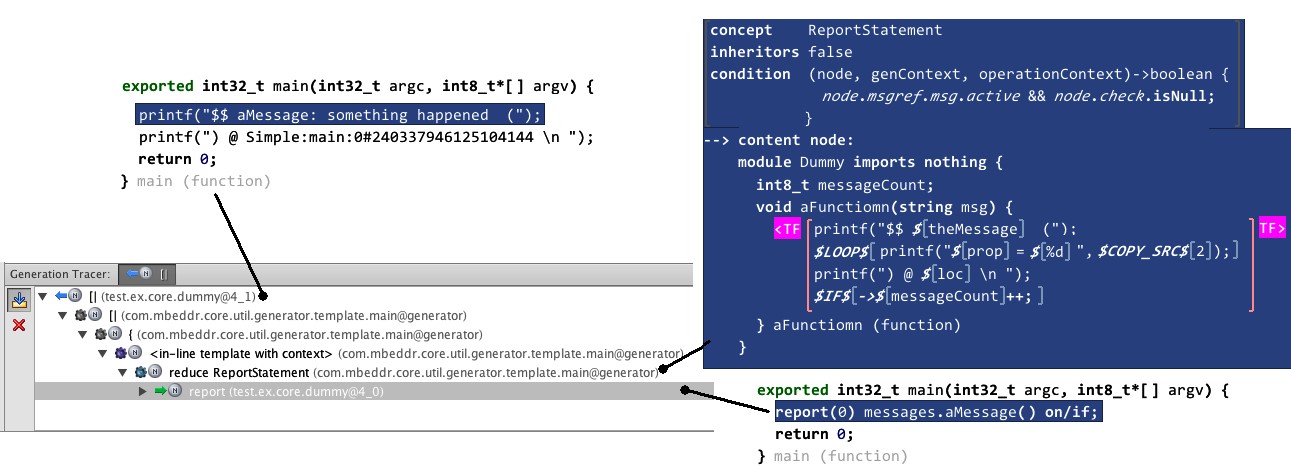
This particular model is generated in three phases. The first one removes commented code to make sure it does not show up in the resulting C text file. The second phase runs the gen-

11 genererator that transforms **report** statements into **printf**s. Finally, the **desktop** generator generates a **make** file from the build con- figuration, and the last step generates the C text from the C tree11.

By default, MPS runs all generators until everything is either discarded or transformed into text. While intermediate models exist, they are not shown to the user. For debugging purposes though, these intermediate, transient models can be retained for inspection. Each of the phases is represented by one or more transient models. As an example, here is the program after the **report** statement has been transformed:

|  |
| --- |
| **module** Simple **imports** nothing {  **exported** int32 main(int32 argc, int8\*[] argv) { printf("$$ aMessage: something happened "); printf("@ Simple:main:0#240337946125104144 \n "); **return** 0;  }  } |

MPS also supports tracing an element through the intermediate models. Fig. 15.5 shows an example. Users can select a program element in the source, target or an intermediate model and trace it to the respective other ends of the transformation..



Note how this approach to debugging transformations is very 12

different from the Xtend example above: instead of stepping MPS functionality, MPS transformations through the transformation code12, MPS provides a *static* rep-

resentation of the transformation in terms of the intermediate models and the element traces through them.

### 15.2 Debugging DSL Programs

|  |  |
| --- | --- |
| or because the *LD*−1 is so low-level and complex that is bears no obvious resemblance to the DSL program.  The way to build debuggers for DSLs of course depends on the DSL itself. For example, for DSLs that only describe structures, debugging does not make much sense in the first place. For DSLs that describe behavior, the debugging approach depends on the behavioral paradigm used in the DSL. We have discussed this in Section 5. In this section we focus mostly on |  |
| the imperative paradigm14. |  |
| Building a debugger poses two challenges. The first one is the debugger UI: creating all the buttons and views for controlling the debugger and for showing variables and treads. The second challenge concerns the control of and data exchange with the program to be debugged. The first challenge is relatively simple to solve, since many IDE frameworks (including Eclipse and MPS) already come with debugger frameworks.  The second challenge can be a bit more tricky. If the DSL is executed by an interpreter, the situation is simple: the interpreter can be run and controlled directly from the debugger. It is easy to implement single-stepping and variable watches, for example, since the interpreter can directly provide the re- |  |
| spective interfaces15. On the other hand, if the DSL program |  |

To find errors in DSL programs, we can either debug them on the level of the DSL program or in its *LD*−1 representation (i.e. in the generated code or the interpreter). Debugging on *LD*−1 is useful if you want to find problems in the execution engine, or, to some extent, if the language users are programmers and they have an intimate understanding of the *LD*−1 representation of the program. However, for many DSLs it is necessary to debug on the level of the DSL program, either because the users are not familiar with the *LD*−1 representation13,

is transformed into code that is executed in some other environment outside of our control, it may even be impossible to build a debugger, because there is no way to influence and inspect the running program. Alternatively, it may be necessary to build a variant of the code generator which generates a *debug version* of the program that contains specialized code to interact with the debugger. For example, values of variables may be stored in a special data structure inspectable by the debugger, and at each program location where the program may have to stop (in single-step mode or as a consequence of a breakpoint)

code is inserted that explicitly suspends the execution of the program, for example by **sleep**ing the current thread. However, such an approach is often limited and ugly – in the end, an execution infrastructure must provide debug support to enable robust debugging.

#### 15.2.1 Print Statements – a Poor Man’s Debugger

As the above discussion suggests, building full-blown debuggers may be a lot of work. It is worth exploring whether a simpler approach is good enough. The simplest such approach is to extend the DSL with language concepts that simply print interesting aspects of the executing program to the console or a log file. For example, the values of variables may be output this way.

The mbeddr **report** statement is an example of this approach. A **report** statement takes a message text plus a set of variables. It then outputs the message and the values of these variables. The target of the report statement can be changed. By default, it reports to the console. However, since certain target devices may not have any console16, alternative transforma-

|  |  |
| --- | --- |
| tions can be defined for **report** statements, that, for example, could output the data to an error memory or a serial line. A particularly interesting feature of **report** statements is that the transformation that handles them knows where in the program the **report** statement is located and can add this information |  |
| to the output17. |  |
| An approach based on print *statements* is sometimes clumsy, because it requires factoring out the expression to be printed[[2]](#footnote-2), |  |

and it only works for an imperative language in the first place. For languages that make use of sophisticated expressions, a different approach is recommended. Consider the following example:

Collection[Type] argTypes = aClass.operations.arguments.type;

If you wanted to print the list of operations and arguments, you would have to change the program to something like this:

print("operations: " + aClass.operations); print("arguments: " + aClass.operations.arguments); Collection[Type] argTypes = aClass.operations.arguments.type;

A much simpler alternative uses *inlined* reporting expressions:

|  |
| --- |
| Collection[Type] argTypes = aClass.operations.print("operations:")  .arguments.print("arguments:").type; |

To make this convenient to use, the **print** function has to return the object it is called on (the one before the dot), and it must be typed accordingly if a language with static type checking is used19.

|  |  |  |
| --- | --- | --- |
| *15.2.2 Automatic Program Tracing*  As languages and programs become more complex, an automated tracing of program execution may be useful. In this approach, all execution steps in a program are automatically traced and logged into a tree-like data structure. The refrigerator cooling language uses this approach. Here is an example program:   |  | | --- | | **cooling program** HelloWorld { **var** temp: **int start**:  **entry** { **state** s1 }  **state** s1: **check** temp < 10 { **state** s2 }  **state** s2: } |   Upon startup, it enters the **start** state and immediately transitions to state **s1**. It remains in **s1** until the variable **temp** becomes less than 10. It then transitions to **s2**. Below is a test for this program that verifies this behavior:   |  | | --- | | **test** HelloWorldTest **for** HelloWorld { **prolog** { **set** temp = 30  } **step**  **assert**-**currentstate**-**is** s1 **step**  **mock**: **set** temp = 5  **step assert**-**currentstate**-**is** s2 } |   Fig. 15.6 shows the execution trace. It shows the execution of each statement and the evaluation of each expression. The log viewer is a tree table, so the various execution steps can be selectively expanded and collapsed. Users can double-click on an entry to select the respective program element in the source node. By adding special comments to the source, the log can be structured further[[3]](#footnote-3).  The execution engine for the programs is an interpreter, which makes it particularly simple to collect the trace data[[4]](#footnote-4). |

19 The original openArchitectureWare Xtend did it this way.

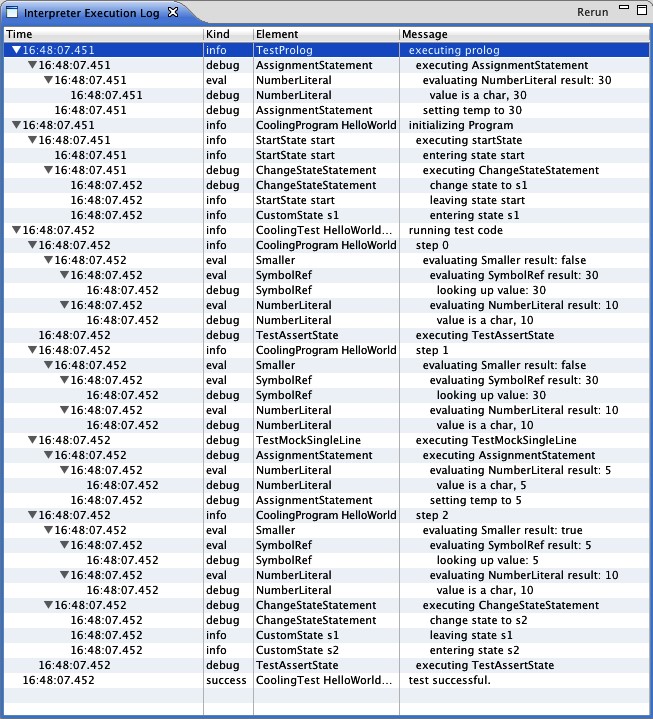
All interpreter methods that execute statements or evaluate expressions take a **LogEntry** object as an additional argument. The methods then add children to the current **LogEntry** that describe whatever the method did, and then pass the child to

any other interpreter methods it calls. As an example, here is the implementation of the **AssignmentStatement**:

|  |
| --- |
| **protected void** executeAssignmentStatement(AssignmentStatement s,  LogEntry log) {  LogEntry c = log.child(Kind.info, context,  "executing AssignmentStatement" );  Object l = s.getLeft();  Object r = eval(s.getRight(), c); eec().environment.put(symbol, r); c.child(Kind.debug, context,  "setting " + symbol.getName() + " to " + r);  } |

#### 15.2.3 Simulation as an Approximation for Debugging

The interpreter for the cooling programs mentioned above is of course not the final execution engine – C code is generated that is executed on the actual target refrigerator hardware. However, as we discussed in Section 4.3.7, we can make sure the generated code and the interpreter are semantically identical by running a sufficient (large) number of tests. If we do this, we can use the interpreter to test the programs for logical errors.



|  |  |
| --- | --- |
| If you already have the interpreter24, expanding it into a simulator/debugger is relatively simple. Essentially only three things have to be done:   * First, the execution of the program must be controllable from the outside. This involves setting breakpoints, singlestepping through the program and stopping execution if a breakpoint is hit. In our example case, we do not single-step through statements, but only through steps25. Breakpoints are essentially Boolean flags associated with program elements: if the execution processes a statement that has the **breakpoint** flat set to **true**, execution stops. * Second, we have implemented an Observer infrastructure for all parts of the program state that should be represented in the simulator UI. Whenever one of them changes (as a side effect of executing a statement in the program), an event is fired. The UI registers as an observer and updates the UI in accordance with the event. * Third, values from the program state must be changeable from the outside. As a value in the UI (such as the temperature of a cooling compartment) is changed by the user, the value is updated in the state of the interpreter as well.   *15.2.4 Automatic Debugging for Xbase-based DSLs* |  |
| DSLs that use Xbase, Xtext’s reusable expression language, get debugging mostly26 for free. This is because of the tight integration of Xbase with the JVM. We describe this integration in |  |

The interpreter can also be used interactively, in which case it acts as a simulator for the executing program. It shows all variables in the program, the events in the queue, the running tasks, as well as the values of properties of hardware elements and the current state. It also provides a button to single-step the program, to run it continuously, or to run it until it hits a breakpoint. In other words, although the simulator does not use the familiar22 UI of a debugger, it actually is a debugger23!

|  |
| --- |
| more detail in Section 16.3; here is the essence.  A DSL that uses Xbase typically defines its own structural and high-level behavioral aspects, but uses Xbase for the finegrained, expression-level and statement-level[[5]](#footnote-5) behavior. For example, in a state machine DSL, states, events and transitions would be concepts defined by the DSL, but the guard conditions and the action code would reuse Xbase expressions. |

|  |  |
| --- | --- |
| When mapping this DSL to Java28, the following approach |  |
| is used: the structural and high-level behavioral aspects are mapped to Java, but *not* by generating Java text, but by mapping the DSL AST to a Java AST29. For the reused Xbase as- |  |
| pects (the finer-grained behavior) a Java generator (called the Xbase compiler) already exists, which we simply call from our generator.  Essentially, we do not create a code generator, but a modelto-model transformation from the DSL AST to the Java AST. As part of this transformation (performed by the *JVMModel inferrer*), trace links between the DSL code and the Java code are established. In other words, the relationship between the Java code and the DSL code is well known. This relationship is exploited in the debugging process. Xbase-based DSLs use the Java debugger for debugging. In addition to showing the generated Java code, the debugger can also show the DSL code, based on the trace information collected by the JVMModel inferrer. In the same way, if a user sets a breakpoint in the DSL code, the trace information is used to determine where to set the breakpoint in the generated Java code. |  |
| *15.2.5 Debuggers for an Extensible Language*  This section describes in some detail the architecture of an extensible debugger for an extensible language30. We illustrate |  |

the approach with an implementation based on mbeddr, an extensible version of C implemented with the MPS. We also show the debuggers for non-trivial extensions of C.

*Requirements for the Debugger* Debuggers for imperative languages support at least the following features: *breakpoints* suspend execution on arbitrary statements; *single-step execution* steps over statements, and into and out of functions or other callables; and *watches* show values of variables, arguments or other aspects of the program state. *Stack frames* visualize the call hierarchy of functions or other callables.

When debugging a program that contains extensions, breakpoints, stepping, watches and call stacks, these elements *at the extension-level* differ from their counterparts *at the base-level*. The debugger has to perform the mapping from the base-level to the extension-level (Fig. 15.7). We distinguish between the *tree* representation of a program in MPS and the generated *text* that is used by the C compiler and the debugger backend. A program in the tree representation can be separated into parts

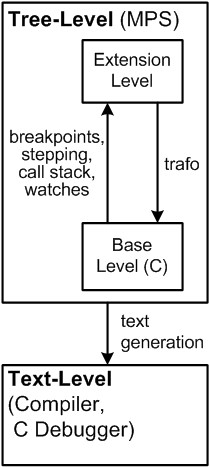


Figure 15.7: An extension-aware debugger maps the debug behavior from the base-level to the extension-level (an extension may also be mapped onto other extensions; we ignore this aspect in this section).

|  |
| --- |
| expressed in the base language (C in this case) and parts expressed using extensions. We refer to the latter the *extension*level or *DSL*-level (see Fig. 15.7). An extensible tree-level debugger for mbeddr that supports debugging on the base-level and extension-level, addresses the following requirements:  *Modularity* Language extensions in mbeddr are modular, so debugger extensions must be modular as well. No changes to the base language must be necessary to enable debugging for a language extension.  *Framework Genericity* In addition, new language extensions must not require changes *to the core debugger infrastructure* (not just the base language).  *Simple Debugger Definition* Creating language extensions is an integral part of using mbeddr. Hence, the development of a debugger for an extension should be simple and not require too much knowledge about the inner workings of the framework, or even the C debugger backend.  *Limited Overhead* As a consequence of embedded software development, we have to limit the additional, debugger-specific code generated into the binary. This would increase the size of the binary, potentially making debugging on a small target device infeasible.  *Debugger Backend Independence* Embedded software projects use different C debuggers, depending on the target device. This prevents modifying the C debugger itself: changes would have to be re-implemented for every C debugger used.  *An Example Extension* We start out by developing a simple extension to the mbeddr C language[[6]](#footnote-6). The **foreach** statement |

can be used to conveniently iterate over C arrays. Users have to specify the array as well as its size. Inside the **foreach** body, **it** acts as a reference to the current iteration’s array element[[7]](#footnote-7).

int8 s = 0; int8[] a = {1, 2, 3}; **foreach** (a **sized** 3) { s += it;

}

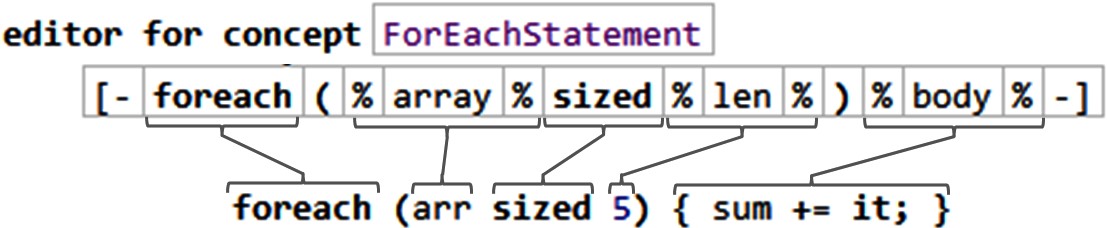
The code generated from this piece of extended C looks as follows. The **foreach** statement is expanded into a regular **for** statement and an additional variable **\_\_it**:

|  |
| --- |
| int8 s = 0; int8[] a = {1, 2, 3};  **for** (**int** \_\_c = 0; \_\_c < 3; \_\_c++) { int8 \_\_it = a[\_\_c]; s += \_\_it;  }  To make the **foreach** extension modular, it lives in a separate language module named **ForeachLanguage**. The new language extends C, since we will refer to concepts defined in C (see  Fig. 15.8).  *Developing the Language Extension* In the new language, |

we define the **ForeachStatement**. To make it usable wherever C expects **Statement**s (i.e. in functions), it extends C’s

**Statement**. As Fig. 15.8 shows, **ForeachStatement**s have three children: an **Expression** that represents the array, an **Expression** for the array length, and a **StatementList** for the body. **Expression** and **StatementList** are both defined in C.

The editor is shown in Fig. 15.9. It consists of a horizontal list of cells: the **foreach** keyword, the opening parenthesis, the embedded editor of the **array** child, the **sized** keyword, the embedded editor of the **len** expression, the closing parenthesis and the editor of the **body**.



As shown in the code snippet below, the **array** must be of type **ArrayType**, and the type of **len** must be **int64** or any of its shorter subtypes.

|  |
| --- |
| **rule** typeof\_ForeachStatement **for** ForeachStatement **as** fes **do** { **typeof**( fes.len ) :<=: <int64>;  **if** (!(fes.array.type.isInstanceOf(ArrayType))) { **error** "array required" -> fes.array; }  } |

As shown above, the generator translates a **ForeachStatement** to a regular **for** statement that iterates over the elements with a counter variable **\_\_c** (Fig. 15.10). Inside the **for** body, we create a variable **\_\_it** that refers to the array element at position **\_\_c**. We then copy in the other statements from the body of the **foreach**.

The **ItExpression** extends C’s **Expression** to make it usable where expressions are expected. The editor consists of a single cell with the keyword **it**. A constraint enforces the **ItExpression** to be used only inside the body of a **foreach**:

|  |
| --- |
| **concept constraints** ItExpression {  **can be child**  (context, **scope**, parentNode, **link**, childConcept)->**boolean** { parentNode.ancestor<ForeachStatement, +>.isNotNull && parentNode.ancestor<StatementList, +>.isNotNull;  }  } |

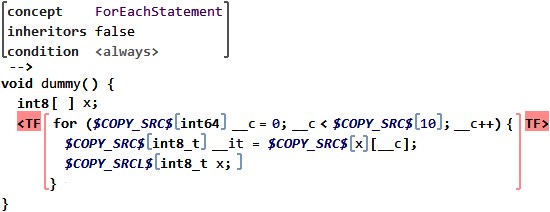
The type of **it** must be the base type of the **array** (e.g. **int** in the case of **int[]**), as shown in the code below:

node<Type> basetype = **typeof**(it.ancestor<ForeachStatement>.array)

:ArrayType.baseType;

**typeof**(it) :==: basetype.copy;

The **foreach** generator already generated a local variable **\_\_it** into the body of the **for** loop. We can thus translate an **ItExpression** into a **LocalVariableReference** that refers to **\_\_it**.



|  |  |
| --- | --- |
| *Developing the Debug Behavior* The specification of the debugger extension for **foreach** resides completely in the **ForeachLanguage**; this keeps the debugger definition for the extension local to the extension language.  To set a breakpoint on a concept, it must implement the  **IBreakpointSupport** marker interface. **Statement** already implements this interface, so **ForEachStatement** implicitly implements this interface as well.  Stepping behavior is implemented via **ISteppable**. The **ForeachStatement** implements this interface indirectly via **Statement**, but we have to overwrite the methods that define the step over and step into behavior. Assume the debugger is suspended on a **foreach** and the user invokes *step over*. If the array is empty or we have finished iterating over it, a step over |  |
| ends up on the statement that follows *after the whole* **foreach** statement. Otherwise we end up on the first line of the **foreach** body (**sum += it;**)33. |  |

The debugger cannot guess which alternative will occur, since it would need to know the state of the program and to evaluate the expressions in the (generated) **for**. Instead we set breakpoints *on each of the possible next statements* and then resume execution until we hit one of them. The implementations of the **ISteppable** methods specify strategies for setting breakpoints on these possible next statements. The **contributeStepOverStrategies** method collects strategies for the *step over* case:

|  |
| --- |
| **void** contributeStepOverStrategies(**list**<IDebugStrategy> res) { **ancestor**  **statement list**: **this**.body } |

The method is implemented using a domain-specific language for debugger specification, which is part of the mbeddr debugger framework34. It is an extension of MPS’ BaseLanguage, a

Java-based language used for expressing behavior in MPS. The **ancestor** statement delegates to the **foreach**’s ancestor; this will lead to a breakpoint on the subsequent statement. The second line leads to a breakpoint on the first statement of the **body** statement list.

Since the **array** and **len** expressions can be arbitrarily complex and may contain invocations of callables (such as function calls), we have to specify the *step into* behavior as well. This requires the debugger to inspect the expression trees in **array** and **len** and find any expression that can be stepped into. Such expressions implement **IStepIntoable**. If so, the debugger has to step into each of those, in turn. Otherwise the debugger falls back to *step over*. An additional method configures the expression trees which the debugger must inspect:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **void** contributeStepIntoStrategies(list<IDebugStrategy> res) {  **subtree**: **this**.array **subtree**: **this**.len  }   |  |  | | --- | --- | | By default, the Watch window contains all C symbols (global and local variables, arguments) as supplied by the native C |  | | debugger35. To customize watches, a concept has to implement |  | |

**IWatchProvider**. Here is the code for **foreach**, also expressed in the debugger definition DSL:

|  |
| --- |
| **void** contributeWatchables(list<UnmappedVariable> unmapped, list<IWatchable> mapped) {  **hide** "\_\_c" **map** "\_\_it" **to** "it" **type**: **this**.array.**type** : ArrayType.baseType **category**: WatchableCategories.LOCAL\_VARIABLES context: **this**  } |

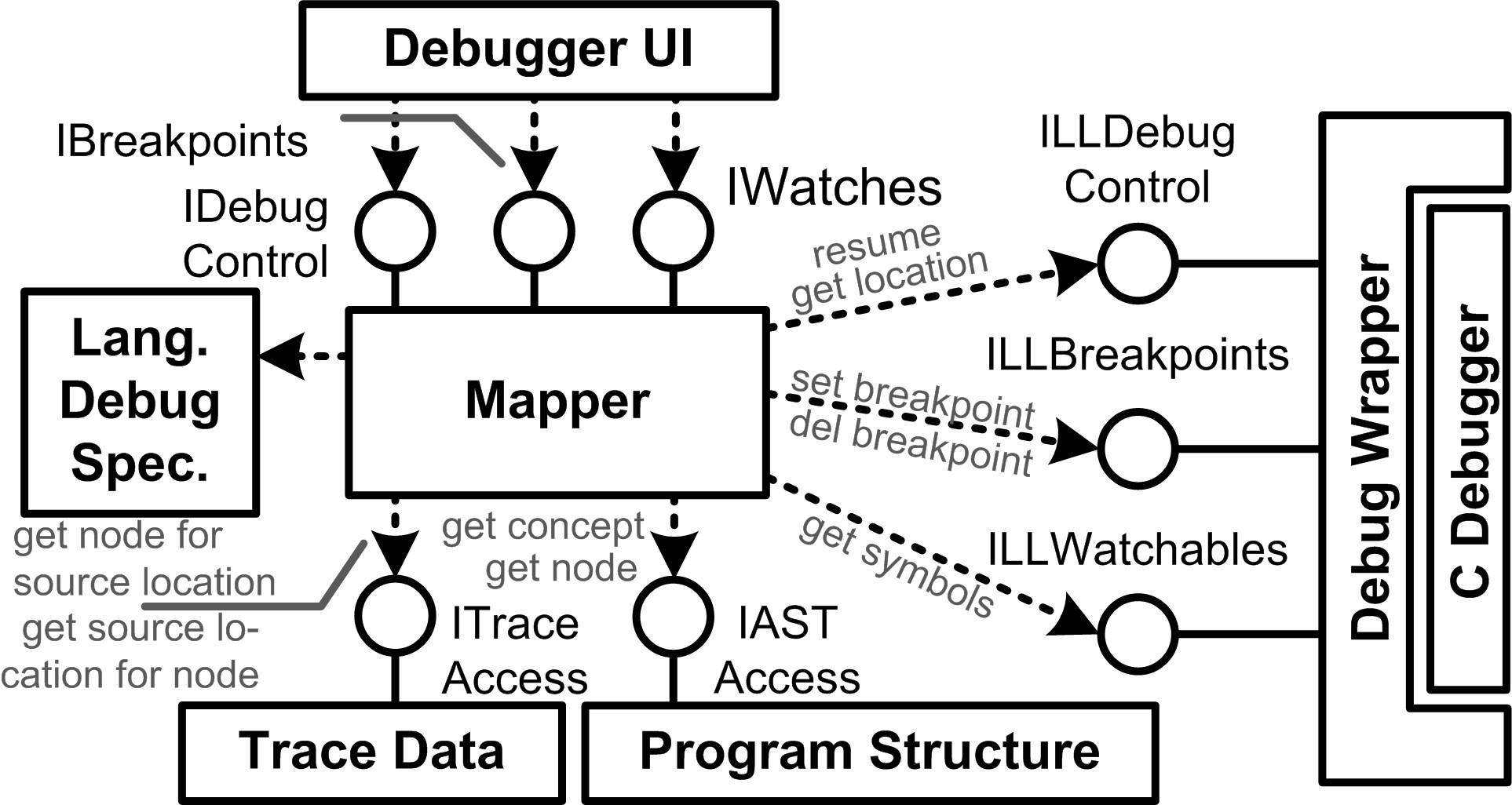
The first line hides **\_\_c**. The rest **map**s a base-level C variable to a watchable. It finds a C variable named **\_\_it** (inserted by the **foreach** generator) and creates a watch variable named **it**. At the same time, it hides the base-level variable **\_\_it**. The type of **it** is the base type of the array over which we iterate. We assign the **it** watchable to the local variables section and associate the **foreach** node with it (double-clicking on the **it** in the Watch window will highlight the **foreach** in the code).

Stepping into the **foreach** body does not affect the call stack, since the concept represents no callable (for details, see the next paragraph). So we do not have to implement any stack frame related functionality.

*Debugger Framework Architecture* The central idea of the debugger architecture is this: from the C code in MPS and its extensions (tree level) we generate C text (text level). This text is the basis for the debugging process by a native C debugger. We then use trace data to find out how the generated text maps back to the tree level in MPS.

At the core of the execution architecture is the **Mapper**. It is driven by the **Debugger UI** (and through it, the user) and controls the C debugger via the **Debug Wrapper**. It uses the

**Program Structure** and the **Trace Data**. The **Mapper** also uses a language’s debug specification, discuss in the next subsection. Fig. 20.6 shows the components and their interfaces.

.

The **IDebugControl** interface is used by the **Debugger UI** to control the **Mapper**. For example, it provides a **resume** operation. **IBreakpoints** allows the UI to set breakpoints on program nodes. **IWatches** lets the UI retrieve the data items for the Watch window. The **Debug Wrapper** essentially provides the same interfaces, but on the level of C (prefixed with **LL**, for

"low level"). In addition, **ILLDebugControl** lets the **Mapper** find out about the program location of the **C Debugger** when it is suspended at a breakpoint. **IASTAccess** lets the **Mapper** access program nodes. Finally, **ITraceAccess** lets the **Mapper** find out the program node (tree level) that corresponds to a specific line in the generated C source text (text level), and vice versa.

To illustrate the interactions of these components, we describe a *step over*. After the request has been handed over from the UI to the **Mapper** via **IDebugControl**, the **Mapper** performs the following steps:

* Asks the current node’s concept for its *step over* strategies; these define all possible locations where the debugger could end up after the *step over*.
* Queries **TraceData** for the corresponding lines in the generated C text for those program locations.
* Uses the debugger’s **ILLBreakpoints** to set breakpoints on those lines in the C text.
* Uses **ILLDebugControl** to resume program execution. It will stop at any of the breakpoints just created.
* Uses **ILLDebugControl** to get the C call stack.
* Queries **TraceData** to find out, for each C stack frame, the corresponding nodes in the tree-level program.
* Collects all relevant **IStackFrameContributor**s (see the next section). The **Mapper** uses these to construct the tree-level call stack.
* Gets the currently visible symbols and their values via **ILLWatchables**.
* Queries the nodes for all **WatchableProvider**s and use them to create a set of watchables.

At this point, execution returns to the **Debugger UI**, which then gets the current location and watchables from the **Mapper** to highlight the statement on which the debugger is suspended and populate the Watch window.

In our implementation, the **Debugger UI**, **Program Repository** and **Trace Data** are provided by MPS. In particular, MPS builds a trace from the program nodes (tree level) in MPS to the generated text-level source. The **Debug Wrapper** is part of

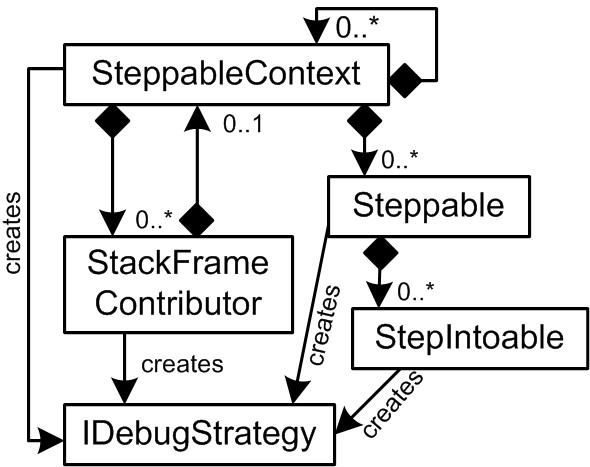
### 36

|  |  |
| --- | --- |
| mbeddr and relies on the Eclipse CDT Debug Bridge , which |  |
| provides a Java API to **gdb**37 and other C debuggers.  *Debugger Specification* The debugger specification resides in the respective language module. As we have seen in the **foreach** example, the specification relies on a set of interfaces and a number of predefined strategies, as well as the debugger specification DSL.  The interface **IBreakpointSupport** is used to mark language concepts on which breakpoints can be set. C’s **Statement** implements this interface. Since all statements inherit from **Statement** we can set breakpoints on all statements by default.  When the user sets a breakpoint on a program node, the mapper uses **ITraceAccess** to find the corresponding line in the generated C text. A statement defined by an extension may be expanded to several base-level statements, so **ITraceAccess** actually returns a range of lines, the breakpoint is set on the first one.  Stack frames represent the nesting of invoked callables at |  |
| runtime38. We create stack frames for a language concept if |  |
| it has callable semantics. The only callables in C are functions, but in mbeddr, test cases, state machine transitions and |  |

component methods are callables as well. Callable semantics on extension level do not necessarily imply a function call on the base level. There are cases in which an extension-level callable is *not* mapped to a function and where a non-callable *is* mapped to a function. Consequently, the C call stack may differ from the extension call stack shown to the user. Concepts with callable semantics on the extension level or base level implement **IStackFrameContributor**. The interface provides operations that determine whether a stack frame has to be created in the debugger UI and what the name of the stack frame should be.

Stepping behavior is configured via the **IStackFrameContributor**, **ISteppable**, **ISteppableContext**, **IStepIntoable** and **IDebugStrategy** interfaces. Fig. 15.12 shows an overview.

The methods defined by these interfaces return *strategies* that determine where the debugger may have to stop next if the user selects a stepping operation (remember that the debugger framework sets breakpoints to implement stepping). New strategies can be added without changing the generic execution aspect of the framework.



Stepping relies on **ISteppable** contributing *step over* and *step into* strategies. Many **ISteppable**s are embedded in an

**ISteppableContext** (e.g., **Statement**s in **StatementList**s). Strategies may delegate to the containing **ISteppableContext** to determine where to stop next (the **ancestor** strategy in the **foreach** example).

For *step into* behavior, an **ISteppable** specifies those subtrees in which instances of **IStepIntoable** may be located (the **array** and **len** expressions in the **foreach** case). The debugger searches these subtrees at debug-time and collects all instances of **IStepIntoable**. An **IStepIntoable** represents a callable invocation (e.g., a **FunctionCall**), and the returned strategies suspend the debugger within the callable.

*Step out* behavior is provided by implementors of **IStackFrameContributor** (mentioned earlier). Since a callable can be called from many program locations, the call site for a particular invocation cannot be determined by inspecting the program structure; a call stack is needed. We use the ordered list of

**IStackFrameContributor**s, from which the tree-level call stack is derived, to realize the *step out* behavior. By "going back" (or "out") in the stack, the call site for the current invocation is determined. For *step out*, the debugger locates the enclosing

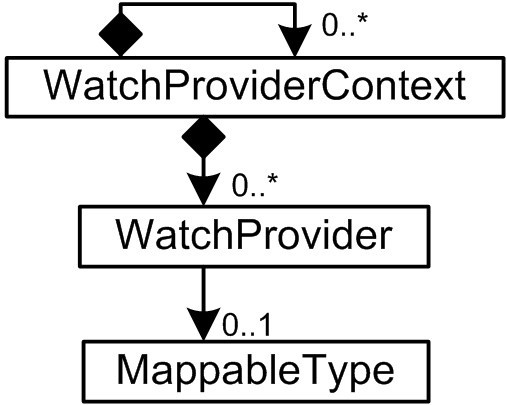
**IStackFrameContributor** and asks it for its *step out* strategies.

Strategies implement **IDebugStrategy** and are responsible for setting breakpoints to implement a particular stepping behavior. Language extensions can either implement their own strategies or use predefined ones. These include setting a breakpoint on a particular node, searching for **IStepIntoables** in expression subtrees (step into), or delegating to the outer stack frame (step out).

To support *watches*, language concepts implement **IWatchProvider** if they directly contribute one or more items into the Watch window. An **IWatchProviderContext** contains zero or more watch providers. Typically these are concepts that own statement lists, such as **Function**s or **IfStatement**s. If the debugger is suspended on any particular statement, we can find all visible watches by iterating through all ancestor

**IWatchProviderContext**s and asking them for their **IWatchProvider**s. Fig. 15.13 shows the typical structure of the concepts.

An **IWatchProvider** implements the **contributeWatchables** operation. It has access to the C variables available in the native



|  |
| --- |
| C debugger. Based on those, it creates a set of watchables. The method may hide a base-level C variable (because it is irrelevant to the extension-level), promote C variable to a watchable or create additional Watchables based on the values of C variables. The representation of a watchable often depends on the variable’s type *as expressed in the extension program*. This type may be different from the one in the C program. For example, we represent values of type **Boolean** with *true* and *false*, even though they are represented as **int**s in C. As the watchable is created, we specify the type that should be used. Types that should be used in this way must implement **IMappableType**. Its method **mapVariable** is responsible for computing a typeappropriate representation of a value.  *More Examples* To illustrate mbeddr’s approach to extensible debuggers further, we have implemented the debugging behavior for mbeddr C and all default extensions. We discuss some interesting cases in this section.  We encounter many cases where we cannot know statically which piece of code will be executed when *stepping into* a callable.  Consider polymorphic calls on interfaces.  The mbeddr components extension provides interfaces with operations, as well as components that **provide** and **use** these interfaces. The component methods that implement interface operations are generated to base-level C functions. The same interface can be implemented by *different* components, each implementation ending up in a *different* C function. A client component only specifies the *interface* it uses, not the component. Hence we cannot know statically which C function will be called if an operation is invoked on the interface. However, we do know statically all components that implement the interface, so we know *all possible C functions* that may be invoked. A strategy implemented specifically for this case sets breakpoints on the first line *of each of these functions* to make sure we stop in the first line of any of them if the user *steps into* a method invocation[[8]](#footnote-8). |

In many cases a single statement on the extension level is mapped to several statements or whole blocks on the base level. *Stepping over* the single extension-level statement must step over the whole block or list of statements in terms of C. An example is the **assert** statement used in test cases. It is mapped to an **if** statement. The debugger has to step over

the complete **if** statement, independent of whether the condition in the **if** evaluates to **true** or **false**. Note that we get this behavior free40: the **assert** statement sets a breakpoint on

|  |  |
| --- | --- |
| the base-level counterpart of the *next tree-level statement*. It is irrelevant how many lines of C text further down this is.  Extensions may provide custom data types that are mapped to one or more data types or structures in the generated C. The debugger has to reconstruct the representation in terms of the extension from the base level data. For example, the state of a component is represented by a **struct** that has a member for each of the component fields. Component operations are mapped to C functions. In addition to the formal arguments declared for the respective operation, the generated C function also takes this **struct** as an argument. However, to support the polymorphic invocations discussed earlier, the type of this argument is **void\***. Inside the operation, the **void\*** is cast down to allow access to the component-specific members. The debugger performs the same downcast to be able to show watchables for all component fields.  *Discussion* To evaluate the suitability of our solution for our purposes, we revisit the requirements described earlier.  *Modularity* Our solution requires no changes to the base language or its debugger implementation to specify the debugger for an extension. Also, independently developed extensions retain their independence if they contain debugger |  |
| specifications41. |  |
| *Framework Genericity* The extension-dependent aspects of the debugger behavior are extensible. In particular, stepping behavior is factored into strategies, and new strategies can be implemented by a language extension. Also, the representation of watch values can be customized by making the respective type implement **IMappableType** in a suitable way. |  |

*Simple Debugger Definition* This challenge is solved by the debugger definition DSL. It supports the definition of stepping behavior and watches in a declarative way, without concerning the user with implementation details of the framework or the debugger backend.

*Limited Overhead* Our solution generates no debugger specific code at all (except the debug symbols added by compiling the C code with debug options). Instead we rely on trace

data to map the extension level to base level and ultimately to text. This is a trade-off: first, the language workbench must be able to provide trace information. Second, the generated C text cannot be modified by a text processor before it is compiled and debugged, since this would invalidate the trace data42. Our approach has another advantage: we

do not have to change existing transformations to generate debugger-specific code. This keeps the transformations independent of the debugger.

*Debugger Backend Independence* We use the Eclipse CDT Debug Bridge to wrap the particular C debugger, so we can use any compatible debugger without changing our infrastructure. Our approach requires no changes to the native C debugger itself, but since we use breakpoints for stepping, the debugger must be able to handle a reasonable number of breakpoints43. The debugger also has to provide an API for

setting and deleting breakpoints, for querying the currently visible symbols and their values, as well as for querying the code location where the debugger suspended.

*15.2.6 What’s Missing?*

The support from language workbenches for building debuggers for the DSLs defined with the language workbench is not where it should be. In the face of extensible languages or language composition especially, the construction of debuggers is still a lot of work. The example discussed in Section 15.2.5 above is *not* part of MPS in general, but instead a framework that has been built specifically for mbeddr – although we believe that the architecture can be reused more generally.

Also, ideally, programs should be debuggable at any abstraction level: if a multi-step transformation is used, then users should be able to debug the program at any intermediate step. Debug support for for Xbase-based DSLs is a good example of this, but it is only one translation step, and it is a solution that is specifically constructed for Xbase and Java, and not a generic framework.

So there is a lot of room for innovation in this space.

1. . [↑](#footnote-ref-1)
2. . [↑](#footnote-ref-2)
3. . [↑](#footnote-ref-3)
4. d [↑](#footnote-ref-4)
5. . [↑](#footnote-ref-5)
6. s [↑](#footnote-ref-6)
7. , [↑](#footnote-ref-7)
8. e [↑](#footnote-ref-8)