Modelling a Debugger for an Imperative Voice Control Language

Andreas Blunk, Joachim Fischer, and Daniel A. Sadilek

Humboldt-Universität zu Berlin
Unter den Linden 6
D-10099 Berlin, Germany
(blunk|fischer|sadilek)@informatik.hu-berlin.de

Abstract. Creating debuggers for languages has always been a hard task. The main reason is that languages differ a lot, especially in the way programs are executed on underlying platforms. The emergence of metamodel-based technologies for defining languages simplified the creation of various language tools, e.g., creating editors from notation descriptions became common practice. Another, relatively recent, example is the metamodel-based description of execution semantics from which an interpreter can be derived. Such a semantics allows one to apply a model-based approach also to debugger development. In this paper, we demonstrate how a debugger can be modelled for an imperative voice control language. We show models of the debugging context, breakpoints, and stepping of voice control programs. These models are processed by a generic debugger.

1 Introduction

Debuggers are critical tools in software development. They are used by programmers to determine the cause of a program malfunction or simply to understand program execution behaviour. Programmers can follow the flow of execution and, at any desired point, suspend further execution and inspect the program's state. Execution may either be suspended manually or by setting breakpoints at well-defined program locations. A debugger then visualises the program's state. It presents all relevant information in the current context of execution, such as visible variable values and the program location.

Debuggers are well-known for general-purpose languages (GPLs). But, they can also be useful for executable domain-specific languages (DSLs). These languages are tailored to specific application domains. They allow developers to use specific concepts and notations to create programs of a corresponding domain with more concise and readable expressions than in GPLs.

In traditional language engineering, tools including debuggers are usually implemented by hand. But this can be too expensive for DSLs if they are used in small projects. It can also be a problem for bigger languages, e.g. UML or SDL, which are first specified in a standardisation process before tools are implemented by hand. Such manual implementation not only causes a gap between specification and tools but also delays tool availability.

A new language description technique that makes both language and tool development less expensive is metamodelling. It allows to describe the different aspects of a language with models, from which tools can be derived [1]. This is, for example, done for editors and interpreters, but there is currently no such technique for modelling debuggers.

The reason for this is that debuggers heavily depend on how language instances are executed and how runtime states can be extracted [2]. Execution semantics can either be described by a transformation to another language or by interpretation. In transformational semantics, debuggers depend on operating system capabilities and also on compilers and linkers that generate the symbol table and align target code. Debuggers for interpreted languages depend on language-dependent interpreter interfaces. Such dependencies make it hard to develop a modelling technique for debuggers and to implement a generic debugger.

What can already be implemented in a generic fashion is the graphical user interface part of a debugger because it is similar in many debuggers. The Eclipse Debugging Framework (EDF) [3] is such an implementation. It defines a set of Java-Interfaces, which concrete debuggers must implement. The EDF provides generic functionality on the basis of these interfaces. They forward user interactions to concrete implementations and they query them for debugging information that is displayed in the user-interface.

In this paper, we advance this state-of-the-art with a technique for *modelling debuggers*. It requires a metamodel-based description of the abstract syntax of a language and an operational semantics. Such language descriptions allow to (1) access runtime states easily via model repositories and (2) control execution at the granularity of operational semantics steps. Our approach is based on an EDF-based implementation of a *generic* debugger and descriptions for *specific* DSL debuggers. In contrast to *generated* tools, the *generic* debugger processes DSL-specific debugging descriptions and allows for domain-specific debugging of DSL programs. With our approach, the debugging of a DSL is described on the basis of its metamodel. It consists of various descriptions of debugging concepts: context information, program locations, breakpoints and step-operations. We demonstrate a description of these concepts with the sample DSL Voice Control Language (VCL).

In the following section, we present the foundations of our approach in more detail and we explain the language description of VCL in special example sections. Section 3 explains our approach for modelling debuggers. We demonstrate the modelling of a debugger for VCL in Sects. 4 and 5. The paper ends with a short conclusion and future work in Sects. 7 and 8.

2 Foundations

We use the Eclipse Modeling Framework (EMF) [4] as a metamodelling framework for MOF-based metamodels [5] and the operational semantics framework EProvide [6] for executing DSL programs according to an operational semantics

description. Although we use EMF and EProvide, the approach is not limited to these tools. It may also be applied to other MOF-based metamodelling frameworks and other descriptions of operational semantics.

2.1 Metamodelling

A metamodel defines the abstract syntax of a language. It describes the structure of a set of possible language instances with object-oriented description means, e.g. classes, attributes, and associations. Language instances are models that contain instances of metamodel elements, e.g. objects, attribute values, and links.

Metamodelling frameworks, e.g. EMF, allow to work with metamodels and models. They provide a model repository, editors and a programming environment, e.g. in Java, that can be used to write programs on the basis of a metamodel.

Example 1 *An example DSL is the Voice Control Language (VCL). It is an imperative language that can be used to write programs for controller modules connected to a telephone. Besides concepts of imperative languages, e.g. variables and control structures, VCL also contains domain-specific concepts like* say text, perform action *and* listen for key press*. These concepts and their relations are defined by the metamodel, depicted in Fig. 1.*

Basically, VCL programs consist of reusable Modules *that contain sequences of* Actions*. There are domain-specific* Actions *like* SayText *and* Listen *but also ones that remind us of GPLs like* EnterModule*,* Decision *and* Assignment*.* ExpressionActions *can access* Variables *that save program state. They have to be declared local to a specific module or global to all modules. For simplicity reasons* Variables *are always of type Integer.*

A sample instance of the metamodel is depicted in Fig. 2. It is a gambling game, called probe game. The game can be played by calling a telephone that is connected to a corresponding controller. It randomly chooses between 0 and 1 and it tells you to make a guess. You either win the game with three successful probes or you lose it after three tries.

At the program level the global variables score *and* tries *are declared. Execution begins in the main module* probe*. It first outputs some information to the caller and then assigns initial values to the global variables. Next is a* while *action, which is executed if there are tries left. Execution then proceeds in the sub module* doProbe*. Modules can be compared to functions in GPLs. The module* doProbe *declares the local variables* in *and* result*. The first action listens for one of two possible inputs. It then assigns* in *to either 0 or 1. After the listen action, a random number between 0 and 1 is computed via an external system call. If the result matches the value of* in*, the caller gets a point. In either case the caller's tries are decremented. The module* doProbe *is left after execution of the last action has been completed. Execution continues after the* EnterModule *action in the main module* probe*. The section* startConfig *defines a number of initial inputs. They let a language developer test the program.*

The textual representation of VCL instances is defined with TEF [7]. It allows to describe textual notations and derive textual editors automatically.

2.2 Operational Semantics

An operational semantics defines the execution of language instances as a step-wise transition of a runtime state [8]. In a metamodel-based operational semantics, possible runtime states are modelled as part of a DSL metamodel and transitions are defined as model-to-model transformations. Such transformations can be defined with EProvide in one of various languages, e.g. Java, QVT, ASMs, Prolog or Scheme. On the basis of such a definition, EProvide executes DSL instances step-wise. DSL developers can use the Eclipse launching dialog for specifying execution parameters, and they can control execution at the granularity of operational semantics steps. But up to now, EProvide does not support debugger features such as a variable view or more complex stepping.

Example 2 *To define the operational semantics of VCL programs, we extend the metamodel with a description of possible runtime states (emphasised elements in Fig. 1), e.g., the class* Environment*, which holds user inputs and program outputs, and the reference* Program.nextAction*, which plays the role of an instruction pointer.*

We define the transformation (step) and an initial state (reset) in Java (see Listing 1.1). It defines that in each step, one action is to be executed. Actions process inputs and outputs, which are contained in an Environment *object. Inputs are numeric keys pressed on a telephone key pad. They are consumed by* Listen *actions that define actions to be taken. Outputs are strings spoken to the caller. They are produced by* SayText *actions. All other actions define program state computations and conditional execution of actions.*

2.3 Model Transformations

In our approach, we also use model-to-model transformations in the transformation language QVT Relations [9]. A transformation is described on the basis of a source and a target metamodel. One specifies complex object patterns that, when found in a source model, result in matching another object pattern in a target model. If the target pattern does not exist, objects, links and values are created.

A transformation specification consists of a set of top-level relations that must hold in order for the transformation to be successful and ordinary relations that are executed conditionally in *when* and *where* clauses. A relation needs to hold only when the relations in its *when* clause hold. A relation called in a *where* clause needs to hold only when the calling relation holds.

3 An Approach for Modelling Debuggers

The DSLs that are used in our approach are special in two ways. First, their runtime state is completely contained in a model. This makes it possible to describe a debugging representation of a DSL instance on the basis of its metamodel.

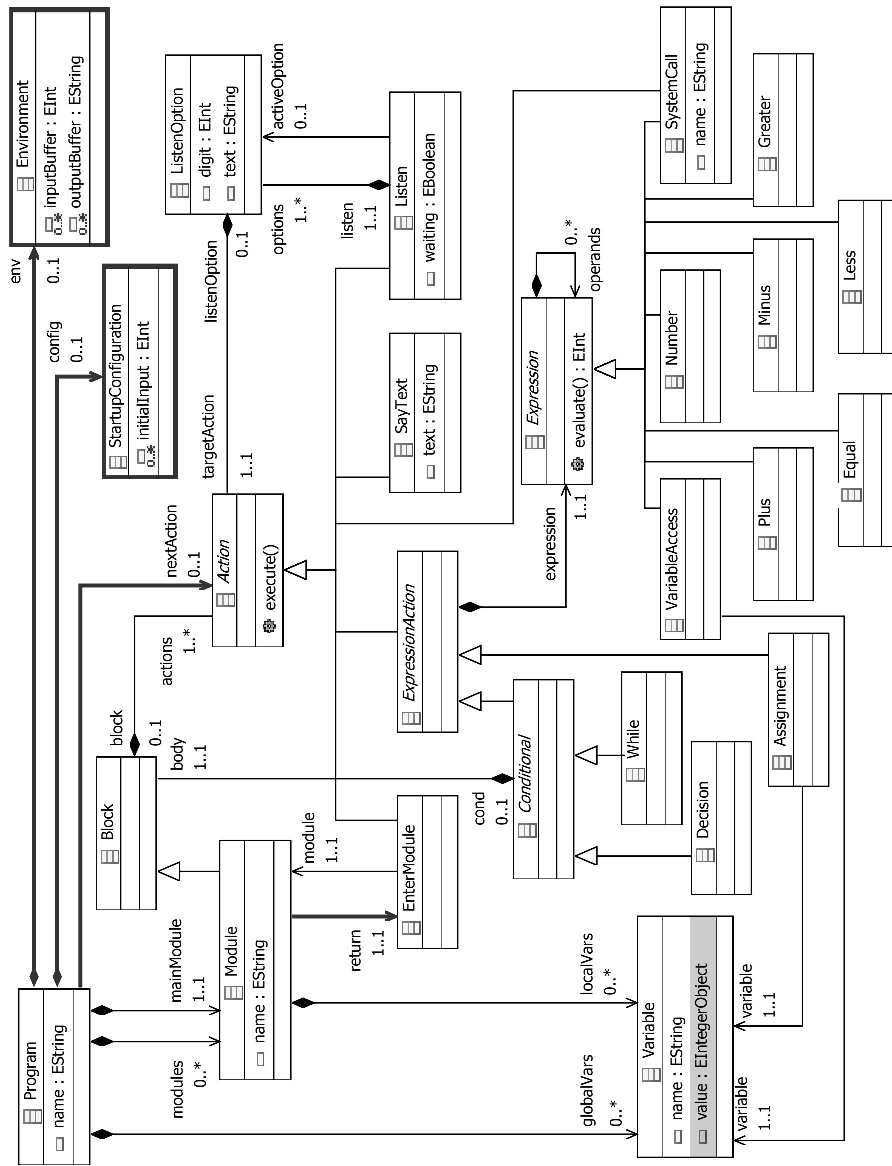

Fig. 1. VCL metamodel.

```
program probeGame {
    decl score, tries;
    mainModule probe {
        say "probe on 0 or 1";
        score = 0;
        tries = 3;
        while tries > 0 {
            enter doProbe;
            if score = 3 {
                tries = 0;
                say "you win";
            }
        }
        if score < 3 {
            say "you lose";
        }
    }
```

```
    module doProbe {
        decl in, result;
        listen {
            0 : "probe 0" : in = 0;
            1 : "probe 1" : in = 1;
        }
        result = sys "random" (0, 1);
        if result = in {
            score = score + 1;
        }
        tries = tries - 1;
    }
    startConfig {
        input = [0, 1, 2];
    }
}
```

Fig. 2. Probe game written in VCL.

```
public class JavaSemanticsProvider implements ISemanticsProvider {

  public void step(Resource model) {
    Program program = getProgram(model);
     ...
    Action action = program.getNextAction();
     if (action == null) return;
     if (action instanceof SayText) {
       program.getEnv().getOutputBuffer().add(((SayText) action ).getText ());
    }
    else if (action instanceof Listen ) { ... }
    program.setNextAction (...);
  }

  public void reset (Resource model) {
    Program program = getProgram(model);
    Action firstAction = program.getActions().get (0);
    program.setNextAction( firstAction );
    program.getEnv().getInputBuffer (). clear ();
    program.getEnv().getOutputBuffer(). clear ();
  }
}
```

Listing 1.1. Operational semantics description for VCL programs.

Such a representation is defined by a model-to-model transformation of DSL instances to instances of a debugging metamodel, e.g. in QVT Relations. The debugging metamodel describes concepts for visualising threads, stack frames, variables and values; its instances are mapped to objects in EDF, which are displayed in the user-interface. Besides such state information, program locations are another part of a program's runtime state. Model objects that represent program locations are extracted by model queries and then highlighted in a concrete syntax.

The second characteristic is the step-wise execution of DSL instances. This makes the implementation of a generic debugger possible, which checks a program for active breakpoints to suspend further execution. Breakpoints are based on possible program locations. They can be installed for model objects that may represent program locations. Execution automatically suspends if an object that was marked as a breakpoint is included in a query for program locations. Step-Operations, e.g., step-into, step-over, and step-return, are described similarly by model queries that extract model objects for target program locations. For such target locations, temporary breakpoints are installed and execution automatically suspends when one of those breakpoints is reached.

4 Debugging Context

The presentation of context information is one of the major tasks of a debugger. Context information exists when the execution of a program is suspended. It tells a user where execution currently resides, how execution reached this point and what the values of visible variables are. Context information is derived from the runtime state of a program and displayed in different views in the user-interface of a debugger. The location where execution currently resides is referred to as the *current program location*. It is usually highlighted in an editor for a language. Information about how execution reached a program location includes (1) concurrency information, e.g. information about threads, and (2) information about program parts that were activated during execution. An example for activations are stack-frames that are created for function invocations. We refer to runtime elements that contain such activations as *activation frames*. Each concurrency context contains a sequence of activation frames that reflect the point in time where parts of a program were activated. By selecting an activation frame, information about variable structures is displayed as variables and values.

Example 3 *Figure 3 displays the context information of a VCL program. We can see that the current program location is a* While *action (editor in the middle), that the module probe has been entered to reach this location (left side) and that the variable score is allocated to the value 0 (right side).*

The context information that is displayed depends on the current selection of elements in a debugger's user-interface. An example is the selection of activation frames, which determines the presentation of visible variables and values.

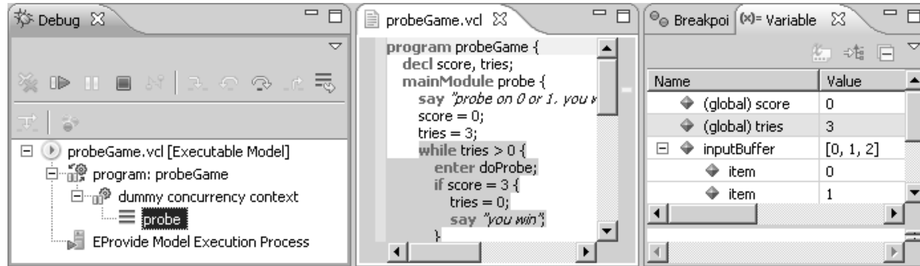

Fig. 3. VCL debugger.

We define a *debugging context* to include runtime information of a suspended program that is relevant to debugging, depending on the current selection of debugger elements. A debugging context includes the following types of information: (1) a set of concurrent execution contexts, (2) a sequence of currently activated program parts as activation frames, (3) a set of visible variables and their values, and (4) a program location. Information types 1-3 are referred to as *debugging state information* because they are described in a different way than *program locations*. There can be many debugging contexts and many debugging states in a suspended program.

4.1 Debugging State

Debugging state information is represented as structured data in two different views of a debugger. Such data can be described by a metamodel as depicted in Fig. 4. The metamodel defines concepts for representing all possible debugging states and their relationships. The presentation of one of these debugging states depends on the current selection of an activation frame. The metamodel is thus referred to as *debugging states metamodel*.

All concepts have a textual representation that is defined by the attribute *labelText* in class *LabeledElement*. The root element of a debugging state is an *MProgramContext*. It contains information about concurrently executing program parts as *MConcurrencyContext*s. Each such context holds a sequence of activated program parts as *MActivationFrame*s. Activation frames contain visible variables and their values as *MVariable*s and *MValue*s. A variable may also contain other variables.

The generic debugger processes debugging state models and presents them in the user-interface. What has to be supplied is a mapping of possible runtime states of DSL programs to instances of the debugging states metamodel. Such a mapping can be defined as a model-to-model transformation in QVT Relations. On the basis of a mapping the generic debugger computes a debugging state model and display its content.

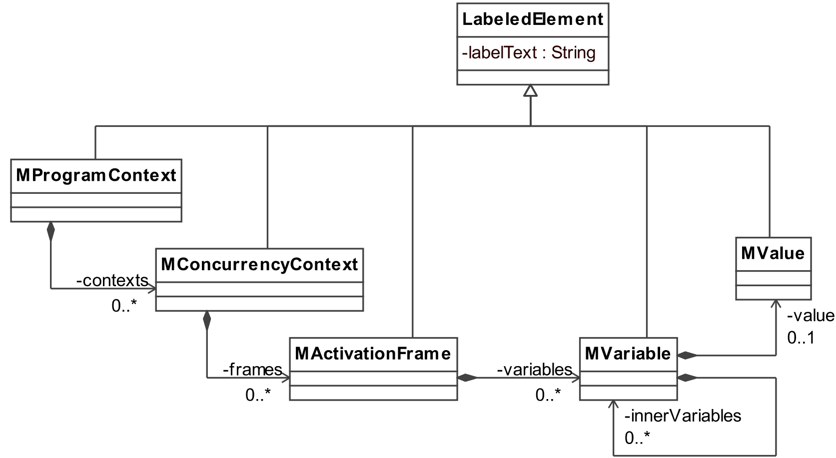

Fig. 4. Debugging states metamodel.

Example 4 *For VCL, the mapping is defined in 115*[a] *lines of QVT Relations statements. Table 4.1 summarises the mapping. The table shows how VCL instances and their attributes are mapped to debugging states instances. We use the colon notation* o:Class *to indicate the presence or the creation of an object* o *for class* Class *and we use the punctuation notation* o.assocEnd *for referring to attributes or association ends.*

The first row indicates that each instance of Program *is mapped to an instance of* MProgram *plus an instance of* MConcurrencyContext*. VCL does not define concurrent execution of program parts. Therefore a mapping to a dummy* MConcurrencyContext *is necessary in order to map activation frames. The program's name* p.name *maps to the label of* MProgramContext mp.labelText*. The module that the next action is contained in* p.nextAction.module *maps to an activation frame in* cc.frames*. If the module was entered from another action and thus* p.nextAction.module.return *is not null, an activation frame is created in* cc.frames*. The mapping continues recursively for the entering action. After these explanations, the rest of the table should be comprehensible to the reader.*

An excerpt of the QVT Relations transformation is displayed in Listing 1.2. The top-level relation localVar *corresponds to the emphasised row in Table 4.1. It maps local VCL variables to debugging state variables. The relation needs to hold only if the relation* frame(m,af) *holds, i.e. there must be an activation frame* af *for the module* m*. The source object pattern checks for local variables and enforces corresponding* MVariable*s to exist. The transformation then continues in the where clause, which maps the variables values.*

With such a transformation, a debugging states model can be created and displayed by the generic debugger, e.g. the left and right side of the VCL debugger in Fig. 3.

[a] Line measurements in this paper do not include comments, empty lines and lines with ending braces.

Table 1. Mapping VCL instances to debugging states instances.

VCL instance	**debugging states instance**
p:Program	mp:MProgram, cc:MConcurrencyContext
p.name	mp.labelText
	cc.labelText = 'dummy concurrency context'
p.nextAction.module	cc.frames
p.nextAction.module.return ...	cc.frames
p.env	cc.frames.variables
p.globalVars	cc.frames.variables
m:Module	af:MActivationFrame
m.name	af.labelText
m.localVars	*af.variables*
v:Variable	mv:MVariable
v.name	mv.labelText
v.value	val:MValue, val.labelText = v.value, mv.value = val
e:Environment	env:MVariable
e.inputBuffer	in:MVariable, env.innerVariables = in
e.inputBuffer.EInt	iv:MValue, iv.labelText = EInt, in.value = iv
e.outputBuffer	out:MVariable, env.innerVariables = out
e.outputBuffer.EString	ov:MValue, ov.labelText = EString, out.value = ov

4.2 Program Location

A program location is a highlighting of an element in the notation of a program that is somehow related to the current point of execution. There are different kinds of program locations. *Current program locations* exist for every concurrent execution context. They highlight a part of a program that will be executed when execution continues. Besides current program locations, debuggers usually also display *context-dependent program locations* for selected activation frames. These program locations highlight a part of a program that is currently being executed.

In GPLs, program locations are often displayed as highlighted statements in a textual concrete syntax. These statements are derived from some kind of instruction pointer. But metamodel-based DSLs are not necessarily textual and they do not need to define explicit instruction pointers. Generally, a program location of such DSLs results from arbitrarily connected objects and their attribute values. Possible program locations are described by formulating an OCL [10] query that extracts model objects from the runtime state of a program. These model objects represent program locations and are highlighted in a notation of the program.

An example are Petri nets. The current program locations of a Petri net are determined by active transitions that are the result of a set of place objects and their markings. Program locations cannot be described by identifying static structures in a Petri net metamodel because there does not have to be an explicit

```
transformation RuntimeStateToDebuggingState(vclModel:vcl, dsModel:debuggingstate) {
  top relation localVar {
    n: String;
    checkonly domain vclModel m : vcl::Module {
      localVars = lv : vcl :: Variable {
        name = n
      }
    };
    enforce domain dsModel af : debuggingstate:: MActivationFrame {
      variables = var : debuggingstate:: MVariable {
        labelText = n
      }
    };
    when { frame(m,af); }
    where { lv.value.oclIsUndefined() or value(lv,var); }
  }
  ...
```

Listing 1.2. QVT transformation for mapping VCL programs to debugging state models.

reference to active transitions. Instead, a model query is necessary that extracts them.

Model queries for program locations are naturally described by OCL queries. But practical realisation requires a connection to an OCL editor, which we did not implement in our approach. Instead, program locations have to be described by implementing a Java interface that is defined by the generic debugger. In order to describe program locations in OCL, some additional code is necessary that evaluates OCL queries via MDT OCL[1] [11]. The generic debugger processes such descriptions when execution has suspended and informs appropriate editors that queried objects need to be highlighted.

The problem when highlighting current program locations is that there can be many such locations if there are multiple concurrent execution contexts. Context information is used to restrict these locations to one context-dependent program location. The interface that has to be implemented defines the operations *getCurrentLocations* and *getLocationInActivationFrame* (see Listing 1.3). The first operation is purely used for breakpoint checking, which is explained in section 5.1, and only the second operation is actually used for highlighting program locations on the basis of a selected activation frame.

[1] MDT OCL is an Eclipse-based OCL implementation for EMF.

```
public class VclSyntaxLocationProvider implements ISyntaxLocationProvider {

  public Collection<EObject> getCurrentLocations(Resource model) {
    // query model objects by using the Java API of MDT OCL
    // or by accessing the model with Java directly.
  }

  public EObject getLocationInActivationFrame(EObject dslFrame) {
    if (dslFrame instanceof Module) {
      Module module = (Module) dslFrame;
      Action nextAction = ((Program) module.eContainer()).getNextAction();
      // determines the currently executing action in the given module recursively.
      return getActionForModule(nextAction, module);
    }
    return null;
  }
  ...
```

Listing 1.3. Description of current program locations for VCL.

Example 5 *In VCL, there is only one current program location, which is determined by the next action to be executed. This action is defined by the current allocation of* Program.nextAction*. Thus, the program location can be described by the following OCL query:* `Program.allInstances()->collect(p : Program | p.nextAction)->asSet()`*. It extracts all instances of* Program *and for the one program that exists, it retrieves the current allocation of* nextAction*. The complete Java implementation consists of around 30 lines of code.*

5 Execution Control

Basic execution control is already part of the generic interpreter EProvide. It allows to start, suspend, and terminate execution and to step forward and backward at the granularity of operational semantics steps. For a full-featured debugger, breakpoints and additional step-operations are necessary.

5.1 Breakpoints

Breakpoints are markings of possible program locations where execution should automatically be suspended. The generic debugger inspects the current program locations for breakpoint markings after each operational semantics step and suspends or continues further execution. This way, DSL developers do not need to describe the reaching of breakpoints explicitly. What has to be supplied is a description of a marking function that checks whether a breakpoint marking for certain model objects should be allowed or not. The function has to be defined

```
public class VclBreakpointDescription implements IBreakpointDescription {
  public boolean isBreakpointable(EObject object) {
    return object instanceof Action;
  }
}
```

Listing 1.4. Description of breakpoints for VCL.

by implementing the Java interface *IBreakpointDescription*. It is used by the generic debugger when a user selects model objects in Eclipse (see Fig. 5). If the marking function evaluates to true, a special breakpoint activation entry is added to the context menu. Such a breakpoint description makes it possible to implement the breakpoint parts of the generic debugger in a completely generic way.

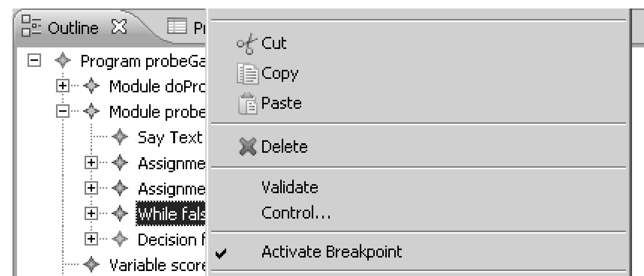

Fig. 5. Context menu for activating breakpoints.

Example 6 *For VCL, breakpoints can be added only to* Actions*. The code in Listing 1.4 shows an implementation in Java.*

5.2 Step-Operations

Step-Operations allow to continue execution to a certain point. In an operational semantics step-operations result in the execution of several transformations until a certain state is reached. The state is determined by extracting target program locations from a program. At these locations, temporary breakpoints are installed and execution suspends again when one of these breakpoints is reached.

There are different kinds of step-operations. A step-over executes invocations of functions to completion. It depends on a program location, which is determined by a currently selected activation frame. On the basis of such a location, target program locations are extracted from the program. Generally, there are

many target locations because execution may proceed at one of several program locations. Such a situation arises if continuation causes the execution of a conditional expression, for example an if-expression.

A step-return executes a currently entered function until completion, i.e. execution continues at the caller of the function. The target of a step-return depends on a selected activation frame. On the basis of such a frame, target program locations are extracted from the program.

The step-operation step-into remains to be the default operational semantics step. Like program locations, step-operations are also described by implementing the Java interface *ISyntaxLocationProvider*.

Example 7 *For VCL, step-operations are described in Java as displayed in Listing 1.5. A step-over can be performed for the actions* EnterModule*,* Listen *and* Conditional*. Execution continues at the action that is located right after the current action at the same branching level. For example, in a* Decision *action it is the action that follows the* Decision *action.*

The target of a step-return is extracted from the selected activation frame, which is in the case of VCL a module. The generic debugger keeps track of source objects that activation frames are created from. Such a source object is provided as parameter dslFrame*. The target is the action that follows the* EnterModule *action, that caused the entering of the current module.*

6 Related Work

An approach for generating debuggers for grammar-based DSLs is the DSL Debugging Framework (DDF) [12]. The abstract syntax of a DSL has to be defined by a grammar and its execution semantics by a transformation to a general-purpose language (GPL) like Java. DSL debuggers are based on a mapping between DSL and GPL code and a GPL debugger. While a mapping describes the debugging concepts of a DSL, the actual debugging process works on the GPL level by using the GPL debugger. Mapping information is used to map DSL debugging commands to GPL debugger commands and GPL debugger results back to DSL results. The approach is limited to textual languages and it needs a GPL debugger for the target language.

Other approaches like ldb [13] and cdb [14] concentrate on generic debuggers for the programming language C. These debuggers can be re-used for varying target architectures, i.e., varying operating systems, compilers and machine architectures. They define a machine independent interface for common debugger actions, e.g. setting breakpoints or reading variable values. The interface has to be implemented for each target architecture. It encapsulates the technical and machine-dependent parts of a C debugger. The debugger itself is implemented on the basis of the machine-independent interface. This approach is also limited to grammar-based languages. Furthermore, execution semantics need to be defined by a special compiler (lcc) that automatically generates information for the debugger.

```
public class VclSyntaxLocationProvider implements ISyntaxLocationProvider {
  ...
  public Collection<EObject> getStepOverLocations(EObject curLocation) {
    Collection<EObject> locations = new HashSet<EObject>();
      if (curLocation instanceof EnterModule || curLocation instanceof Listen
           || curLocation instanceof Conditional) {
        Action action = (Action) curLocation;
        locations.add(action.getAfterAction());
        return locations;
      }
    return null;
  }
  public EObject getStepReturnLocation(EObject dslFrame) {
    if (dslFrame instanceof Module) {
      Module module = (Module) dslFrame;
      if (module.getReturn() != null) {
        module.getReturn().getAfterAction();
      }
    }
    return null;
  }
}
```

Listing 1.5. Description of step-operations for VCL.

7 Conclusion

We presented a novel approach for modelling debuggers of metamodel-based DSLs that have an operational semantics definition[2]. The complete debugging description for VCL programs consists of around 160 lines of different descriptions in OCL, Java, and QVT. We are confident that such a description is a lot smaller and less expensive than a manually implemented debugger, although a direct comparison has not yet been conducted.

8 Future Work

We believe that our approach can also be applied to other metamodel-based languages, e.g. UML activities and the Object Constraint Language OCL. Future work could deal with the description of debuggers for such languages. Our experience with the sample language VCL shows that the availability of descriptions of runtime states and operational semantics is the main obstacle. But if there are such descriptions, debugging can be described with little effort.

[2] The presented approach for describing DSL debuggers and the implementation of the generic debugger, which is called MODEF, are part of a diploma thesis [15]. The thesis includes more detailed information and more complete examples, but is unfortunately only available in German.

Another area of interest is how the mapping for debugging states is described. Our experience shows that the structure of debugging states is, without any or with little structural changes, already part of a DSL metamodel. Consequently, another way to define the mapping could be to add inheritance relations between the classes of a DSL metamodel and the classes of the debugging states metamodel. Debugging state classes could declare associations as derived and DSL classes could specify how these associations are derived. This way, every DSL instance would also be a debugging state instance and could instantly be processed by the generic debugger. We believe that such a mapping would be easier to define and would execute faster than a transformation in QVT Relations.

References

1. Scheidgen, M.: Adopting Meta-modelling for ITU-T Languages: Language Tool Prototypes as a by-Product of Language Specications. SDL Forum Society Event (2008)
2. Rosenberg, J.: How Debuggers Work - Algorithms, Data Structures, and Architecture. John Wiley & Sons, Inc. (1996)
3. Eclipse Foundation: Eclipse Debugging Framework. http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/debug.htm
4. Eclipse Foundation: Eclipse Modeling Framework (EMF). http://www.eclipse.org/modeling/emf
5. OMG: Meta Object Facility (MOF) Core Specification Version 2.0. http://www.omg.org/spec/MOF/2.0/
6. Sadilek, D., Wachsmuth, G.: EProvide: Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages. Fourth European Conference on Model Driven Architecture Foundations and Applications (2008)
7. Humboldt-Universität zu Berlin: Textual Editing Framework (TEF). http://www2.informatik.hu-berlin.de/sam/meta-tools/tef
8. Plotkin, G.: A Structural Approach to Operational Semantics. Technical Report (DAIMI FN-19), University of Aarhus (1981)
9. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. http://www.omg.org/spec/QVT/1.0/
10. OMG: Object Constraint Language (OCL) Version 2.0. http://www.omg.org/spec/OCL/2.0/
11. Christian W. Damus: Implementing Model Integrity in EMF with MDT OCL. http://www.eclipse.org/articles/article.php?file=Article-EMF-Codegen-with-OCL/index.html
12. Wu, H., Gray, J., Mernik, M.: Grammar-Driven Generation of Domain-Specific Language Debuggers. Software: Practice and Experience (Summer 2007)
13. Ramsey, N.: A Retargetable Debugger. Ph.D. Thesis, Princeton University, Princeton, NJ, (1993)
14. Hanson, D., Raghavachari, M.: A Machine-Independent Debugger. Software: Practice and Experience (November 1996)
15. Blunk, A.: MODEF – Ein generisches Debugging-Framework für domänenspezifische Sprachen mit metamodellbasierter Sprachdefinition auf der Basis von Eclipse, EMF und EProvide. Diploma Thesis, Humboldt-Universität zu Berlin (2009)