Interactive Debugging for Extensible Languages in Multi-Stage Transformation Environments

Domenik Pavletic
itemis AG
Stuttgart, Germany
pavletic@itemis.com

Kim Haßlbauer
Stuttgart, Germany
kim.hasslbauer@gmail.com

***Abstract*—Extensible languages have a base language that can be extended incrementally with new language extensions, forming a stack with high-level languages on top and lower level languages at the bottom. Programs written with these languages are usually a mixture of code using base language and several language extensions. These extensions come with generators that translate higher level language constructs to lower levels and ultimately to base language. Program bugs appearing at runtime can be introduced on the source level by language users or through faulty transformation rules by language engineers. The latter category of bugs are often analyzed with a base language debugger, because language constructs introducing the bug on intermediate levels usually have no representation on the source level. However, due to the semantic gap between generated code and the intermediate program where a bug is introduced, users have to map between abstraction levels manually, which is error prone and requires additional effort besides analyzing the bug.**

In this paper we present an approach to build multi-level debuggers for extensible languages that allow language users to debug their code on the source level and language engineers to debug on intermediate levels created during code generation. We illustrate this approach with an implementation for the MPS language workbench and mbeddr C, an extensible C language.

***Index Terms*—Formal languages, Software debugging.**

I. Introduction

Extensible languages are used to develop software systems on a higher level of abstraction and consist of a *base language*, usually a General Purpose Language (GPL), which can be extended with new, usually domain-specific, language extensions on the syntactical and semantic level. Hence, extensible language programs comprise different languages, residing on different abstraction levels. Each of these extensions comes with a generator that translates code to a more concrete language. This translation happens incrementally from higher levels to lower ones, until we get a pure base language program. Fig. 1 shows the process from transforming the *source-level* Abstract Syntax Graph (ASG) of an extensible language program to the *base level* via Model 2 Model (M2M) transformations and ultimately to text (*target level*) via Model 2 Text (M2T) transformations. Graphs inside the boxes represent ASGs of the respective abstraction level, colors indicate structural modifications.

The effort for building such extensible languages can be reduced by using a language workbench, an Integrated Development Environment (IDE) for language engineering. These tools provide facilities to implement language definitions and usually come with generator frameworks to build multi-stage transformations. JetBrains Meta Programming System (MPS) [1] is a workbench that supports the development of extensible languages, mbeddr [2] is one of these languages built with MPS. This language is based on C and comes with a set of language extensions for embedded software engineering.

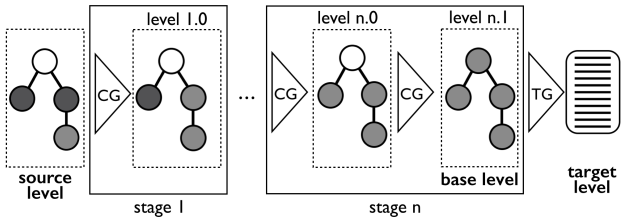

Fig. 1. The colors of ASG nodes indicate how M2M and M2T code generators transform extensible language programs incrementally across different stages from source to base level and ultimately to text (target level)

Runtime bugs that appear in extensible language programs during execution can either be introduced on the source-level by language users or on intermediate levels (e. g., *level 1.0* in the figure above) by language engineers through transformation rules. While language users require a source-level debugger to analyze bugs introduced by themselves, language engineers often investigate bugs introduced through faulty transformation rules by debugging the generated code or the transformation process. Analyzing the transformation process, e. g., by using an omniscient debugger [3], allows language engineers to identify the rule that introduced a faulty code segment and why this rule was executed, however, this approach does not support analyzing the runtime behavior of this generated code segment. Further, generators may have modified the program structure, e. g., by changing identifiers or the structure of `statements`. This abstraction mismatch makes it hard to debug the execution of the generated code.

We present in this paper an approach to build multi-level debuggers for extensible languages. These debuggers enable language engineers to debug programs on different levels, thus allowing them to analyze the runtime behavior of code generated from faulty transformation rules. Further, and essential, these debuggers imitate stepping behavior and show program state based on the languages used on the investigated abstraction level, which can be an intermediate level that has been generated by a faulty transformation. This approach also enables source-level debugging targeting language users.

II. Debugging Extensible Languages

Interactive debuggers allow users to inspect and animate the execution of a program. These debuggers usually operate on the source level and provide, depending on the actual tool, different functionalities: users can put breakpoints on source lines or memory addresses. Further, they can use stepping commands to animate the execution and inspect the program state. Some debuggers even allow users to manipulate values of watch variables or interpret arbitrary expressions.

Programming languages usually support source-level debugging, as multi-level debugging makes little sense in this context: first, these languages are usually not extensible and get transformed to the target language, via a single intermediate language. Debugging intermediate levels is not useful to language users or compiler engineers, as this code is usually similar to the target level representation. Second, language users are often not familiar with the underlying target language (e. g., assembler). They rather consider the compiler as a black box that is configurable via parameters. With extensible languages, the situation is different: while extensions have a common base language, they can be stacked in an arbitrary hierarchical way. That is, new language extensions can be hooked into the generation process. This flexibility increases the complexity and the possibility for introducing bugs into the program through faulty transformations. As described at the end of the section above, bugs introduced by language users can be analyzed with a source-level debugger, while other bugs introduced by faulty transformations are harder to analyze. To support both categories of users we propose multi-level debugging that enables inspecting the program state and controlling execution on different abstraction levels.

To illustrate the usage of multi-level debuggers we consider an example language extension for mbeddr: we introduce an `loop` abstraction for C that allows users to specify iterations with lower and upper boundaries. To test the generator of this language abstraction, we start by writing a `testcase` (an mbeddr extension). Listing 1 below shows the test code: a *main* `function` invokes the `testcase` *sumTesting*, via a `test expression`. This `testcase` uses the `loop` to add up numbers from 0 to 10 in a `variable` *sum*. Finally, an `assert statement` verifies that the value of *sum* equals 55. If the assertion fails, the process returns a positive number, representing the number of failed assertions.

```
1  testcase sumTesting {
2     int32 sum = 0;
3     loop [0 to 10] {
4         sum += it;
5     }
6     assert sum == 55;
7  }
8  int32 main() {
9     return test[sumTesting];
10 }
```

Listing 1. Testing the `loop` generator

Looking at the code shown in Listing 1 and considering the semantics of our `loop`, the test should succeed. However, it fails with a return code 1 indicating a failed assertion. Suppose we have an interactive source-level debugger that allows us to debug Listing 1. By using this debugger we can see that the `loop` body is never reached. Instead, execution jumps directly to the `assert` from the `loop` header. Since we cannot detect the reason for this behavior on the source level, we are forced to use the base language debugger with the generated code shown in Listing 2. A starting point could be to locate the source lines representing our `loop`. Since our generated code only contains one `while`, this is trivial. In more complex scenarios, this would require additional effort.

```
1  int __testcases2323() {
2     int __failures = 0;
3     {    int sum = 0;
4        {  int __index = 10;
5           while(__index <= 0) {
6              sum += __index;
7              __index++;
8           }
9        }
10          if(!(sum == 55)) {
11             __failures++;
12          }
13       }
14       return __failures;
15 }
16 void main() {
17    return __testcases2323();
18 }
```

Listing 2. Generated code for testing the `loop statement`

As we can see on line number 4 and 5 in Listing 2, the initialization of the lower and upper bound was accidentally swapped by the code generator. Therefore the `while` condition never evaluates to `true` and the loop body is not entered. This is exactly the behavior we experienced when debugging on the source level. After identifying the bug, we can fix the problem in the generator first. Afterwards, knowing which program location caused the error, we can use a multi-level debugger to debug on the intermediate level where the `loop` is reduced, but all other abstractions are still present (see Listing 3). This way, we can verify the bug fix we have made in the generator and concentrate on the reduced `loop` while debugging, ignoring irrelevant, generated details.

```
1  testcase sumTesting {
2     int sum = 0;
3     {  int __index = 0;
4        while(__index <= 10) {
5           sum += __index;
6        }
7     }
8     assert sum == 55;
9  }
10 int32 main() {
11    return test[sumTesting];
12 }
```

Listing 3. Intermediate code used for debugging the `loop`

In the context of extensible languages we believe multi-level debuggers support language engineers in the language implementation and maintenance phase. Further, we believe source-level debuggers targeting users of these languages should be built in a way to support multi-level debugging as well. That is, program state and stepping behavior should be lifted incrementally from base to source level, considering all program modifications made by transformation rules in between. In contrast, debuggers for extensible languages operating directly between source and target level have limitations [4]. First, they usually cannot support multiple generators per language and multiple transformation rules per language construct. Second, since they depend on the structure of the generated code, modifying a code generator usually implies updating the debugger implementation.

III. The MuLDer Framework

The Multi-Level Debugger (MuLDer) framework presented in this paper is based on an incremental approach lifting program state from the target level to the currently investigated abstraction level. To describe debugging semantics, debugger developers associate language constructs with debug semantics and specify rules in M2M and M2T transformations to lift the

program state from the generated level back to the original level. These rules annotate the generated code, but do not influence its semantics. They specify how the debugger should lift the lower level program state and get processed incrementally from target level to the currently investigated abstraction level. Further, to imitate stepping behavior, the framework provides two approaches: a control-flow based approach using target-level breakpoints and a single-stepping based approach using single-stepping functionality of an underlying GPL debugger.

A. Architecture

Fig. 2 illustrates with a Unified Modeling Language (UML) component diagram the software components that make up the framework architecture. Grey colored *software components* and *interfaces* describe functionality required from the language workbench (MPS in our reference implementation). They comprise the following parts: ASG access via *IASG*, possibility to retrieve control-flow information for a given program that we require for imitating stepping behavior via *IControlFlowProvider*, adding preference pages to configure debugging via *IPreferences* and contribution of User Interface (UI) components, e. g., a debugger view, via *IUIContribution*.

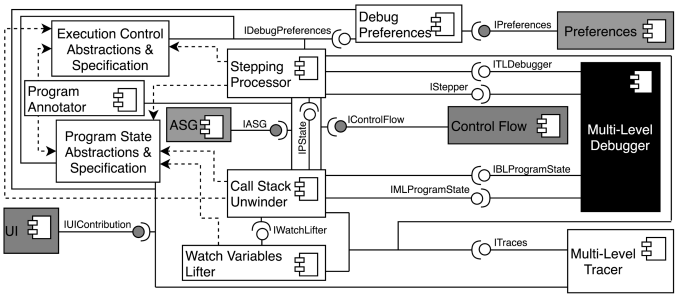

Fig. 2. UML component diagram showing the MuLDer software architecture

White colored boxes represent *abstractions*, *languages*, *components* and *interfaces* from MuLDer. Both *Program State Abstractions & Specification* and *Execution Control Abstractions & Specification* contain language abstractions and specification languages used to describe program state lifting, stepping behavior and translation of breakpoints. Following, *Program Annotator* operates on these language abstractions and accesses via *IASG* the ASG to automatically attach *lifting rules* to ASG nodes (discussed later). Next, *Debug Preferences* contributes via *IPreferences* and *IUIContribution* preference pages to the language workbench. It also provides a UI to manage these preferences, comprising options for defining the currently selected stepping algorithm and configuring visible debug information, e. g., lifting rules, in ASGs. Further, *Stepping Processor* operates on the *Execution Control Abstractions* and provides via *IStepper* an interface to imitate *stepping commands* by using the currently selected stepping algorithm. This component requires the interface *ITLDebugger* to invoke *target-level* stepping commands, used by the single-stepping algorithm, and to manage breakpoints, required by our control-flow based stepping algorithms. *Stepping Processor* requires for some of its algorithms the *program state* being accessed from *Call Stack Unwinder* via *IPState*, access to the ASG via *IASG* and control-flow information being retrieved from *IControlFlowProvider*. However, the latter is only required by the control-flow related stepping algorithms. Next, the call stack is lifted by *Call Stack Unwinder*, which operates on the *Program State Abstractions* and requires ASG access via *IASG*, the incrementally lifted *base-level program state* obtained via *IBLProgramState*, *tracing information* being accessed via *ITraces* and the *lifted watch variables* provided by *Watch Variables Lifter* via *IWatchLifter*. The previously described *Multi-Level Tracer* maintains tracing information describing how ASG nodes are translated across all abstraction levels and provides access to this information via *ITraces*. Finally, *Watch Variables Lifter* lifts lower level watch variables and their associated values based on the *program state abstractions*.

B. Language Abstractions

MuLDer provides a set of abstractions represented by interfaces to specify debugging semantics of language constructs. Debugger developers implement these interfaces by creating a set of queries using a language extension for *Base Language* (an extensible Java coming with MPS). We do not describe these queries in detail, instead, we discuss those being required by our case study in Section IV. The following list contains all abstractions coming with MuLDer: *Steppable* lives inside a *Steppable Composite* (e. g., `statement list`) and is a language construct comparable to `statement` on which we can invoke a stepping command. Next, *Callable* represents a reusable code fragment similar to `function` and can be invoked from other program locations via a *Callable Call* (e. g., `function call`). *Control Flow Provider* is a *Callable* and provides control-flow information for the contained code. *Watch Provider*s contribute watch variables to the debugger view and are usually represented by `variable`s. They are contained in a nestable *Scope Provider* (e. g., `statement list`) and resolve their value by a *Value Provider*, e. g., the `type` of the `variable`.

C. Value Contracts

Value Contracts define *default value lifting rules* and the *structure* of watch variable values that *Value Provider*s contribute. While the *structure* is used for writing formal *Value Transformation*s (discussed later), we use the *rule* to lift low-level watch variable values for which the generated (level `n`) and origin *Value Provider* (level `n-1`) are the same. The *Program Annotator* (see Section III-A) is responsible for attaching these *rules* to intermediate ASGs after code generation. Consider we use a base language type (a *Value Provider*) in our program that is simply translated to text and not modified by any of the code generators. This is one example where the *Program Annotator* would attach the *default value lifting rule* of this type to instances of each level, where the input node of a type is the same type.

We consider in Fig. 3 the *Value Contract* for the mbeddr *PointerType*, consisting of a `complex-value` with *reference semantics* that embodies one `watch` holding an `absent-value`, the pointer target that is known at compile

time (e. g., an `int` type). The code snippet in the box below shows the implementation of the *default value lifting rule* for `complex-value`, other parts, e. g., for `absent-value`, are not shown. In this rule, we use a Domain-Specific Language (DSL) to return the textual `presentation` of the watch variable value. However, as seen in the code completion menu, we can also access other value properties, e. g., `subvalue(s)`, because we have defined the value to be a `complex-value` and `isNull` as the value has *reference semantics* (`*->`).

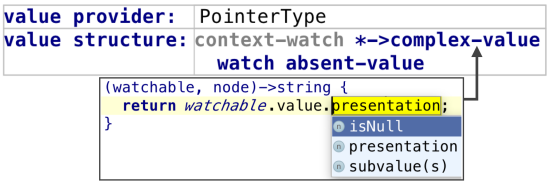

Fig. 3. *Value Contract* for the mbeddr *PointerType*

D. Value Transformations

*Value Transformation*s operate on *Value Contract*s and describe transformations of watch variable values being associated with different *Value Provider*s. To implement such *transformations*, developers specify the structure of a *source* and *target watch* and annotate the latter with *lifting rules* used to construct the *lifted value*.

To illustrate *Value Transformation*s, we consider an example from mbeddr, translating a value of pointer on `char` type to a *StringType* value. For this purpose, we create the *Value Transformation* partially shown in Fig. 4. First, we create value structures for the *source* and *target watch variable* on top, describing the former as *PointerType* with a child value that is associated with a *CharType*. To specify `source-watches`, we refer to information from *Value Contract*s of the referenced *Value Provider*s. After selecting a *Value Provider*, the editor projects the *value structure* of the specified *Value Contract* with the possibility to concretize `absent-values`. In our example, *PointerType* embodies a `watch` of value `absent-value` that we concretize with *CharType*, causing the editor to project the `primitive-value`, coming from the *Value Contract* for *CharType*. Next, we describe the `target-watch` by referencing *StringType*, which projects the `primitive-value` from the *Value Contract*. After describing the structures, we continue with the *default value lifting rule* for `primitive-value` that is shown in the figure below. For this purpose, we use a regular expression that extracts a string enclosed in two quotation marks. Similarly to *Value Contract*s, the properties that we can access on `watchable.value` are based on a value structure, `source-watch` in this context.

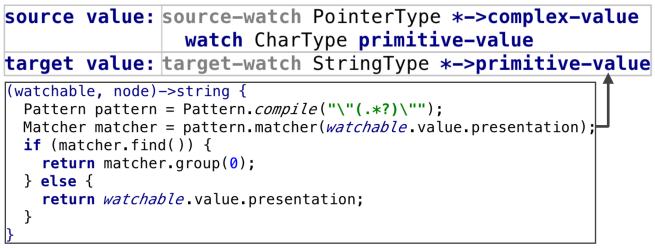

Fig. 4. *Value Transformation* for constructing a string value

E. Target to Base-Level Lifting

Lifting the program state from target to base level is driven by using *identifiers* whereas we perform lifting between other levels by using references between ASG nodes. This section describes the *lifting rules* and *specification languages* we provide to lift program state from target to base level.

Because program code is on base and target level similarly structured, we usually have a one-to-one mapping between base-level ASG nodes and target-level text lines. However, we must track identifiers that we use to establish a mapping between watch variables and stack frames from target and base level. For this purpose we provide a set of Text 2 Model (T2M) *lifting rules* in form of annotations attached to base-level ASG nodes and used to lift program state from *target level* (text) to the *base level* (ASG). As we will later show in Section III-F, the base level is also annotated with M2M *lifting rules*, incrementally lifting program state from base level to the level on which the user debugs his code. Hence, the base level contains annotations to lift program state from target to base level and from base level to the last intermediate level.

The following list describes T2M annotations we provide for lifting program state from target to base level. *T2MFrame2Frame* annotates a base-level *Callable* and holds its generated target-level *identifier*. We use this *identifier* to associate the annotated *Callable* with target-level stack frames. Next, *T2MWatch2Watch* annotates a *Watch Provider* and holds its generated target-level *identifier*. Additionally, this annotation refers to a *Value Provider* lifting the *value*. Following, *T2MValueLifter* annotates a *Value Provider* and refers either to a *Value Transformation* or to a different *Value Provider* delegating the program state *lifting* to it. Finally, *T2MConstant* tracks generated *identifiers*, e. g., `enum` literals.

MPS' M2T transformation language is extensible and translates to *Base Language*. We have exploited this fact by developing a declarative language extension to describe T2M annotations for a given M2T transformation. For this language extension we generate code that attaches the respective annotation to the transformed base-level ASG node at generation time. Fig. 5 below shows parts of the annotated M2T transformation for *Argument*, a *Watch Provider* from mbeddr. We have annotated this transformation with an *M2TWatchProvider* annotation (`@WatchProvider` on top), attached a *M2TIdentifier* to `node.name` (`@IdentifierProvider`) and a *M2TValue* to `node.type` (`@ValueProvider`). During transformation execution an *T2MWatch2Watch* annotation gets attached to the transformed base-level *Argument* comprising information about the generated *identifier* and the *Value Provider*.

F. Incremental Lifting

In contrast to the language extension for MPS' M2T transformation language, we did not extend MPS' generator language. Instead, we provide a set of *rules* to be used in transformations for annotating the generated code.

To unwind the call stack we provide three different annotations: *M2MInlineFrame* annotates a *Callable* for which we *inline* its associated stack frames on the higher level,

```
@WatchProvider
  kind: argument
  category: argument
text gen component for concept Argument {
  (context, buffer, node)->void {
    if (node.storeInRegister) {
      append {register };
    }
    if (node.type.requiresSpecialTextGenLogic()) {
                              @IdentifierProvider
      string variableName = node.name          ;
                           @ValueProvider
      node<Type> type = node.type     ;
```

Fig. 5. Lifting target-level watch variables for mbeddr *Arguments*

M2MFrame2Frame annotates a *Callable* as well, but lifts its stack frames to a *Callable* from the next higher level. Finally, *M2MOutlineFrame* annotates a generated ASG node originating from a *Callable* for which we outline a stack frame.

To lift watch variables, we provide two annotations, both annotate a *Watch Provider*: *M2MWatch2Watch* and *M2MChildWatches2Watches*. The former lifts watch variables contributed by the annotated *Watch Provider* to another *Watch Provider* from the next higher level. In contrast, the latter lifts child values (also contributed by *Watch Provider*s) as top-level watch variables to the next higher level.

To lift watch variable values originating from *Value Provider*s, we provide three different annotations: first, *M2MLiftValue* refers to a *default value lifting rule* and is automatically attached by the *Program Annotator*, second, *M2MGeneratedDelegateToValueProvider* is created by the debugger developer and refers to another *Value Provider* delegating *value lifting* to it. Finally, *M2MGeneratedValueLifter* is also manually created and refers to a *Value Transformation* being used to lift the value representation of a generated *Value Provider*. We have demonstrated in Section III-D a *Value Transformation* for unveiling a string literal from a pointer on `char`. Fig. 6 below shows the transformation rule for *StringType* with a *M2MGeneratedValueLifter* being attached to the generated type and referring to our previously created *Value Transformation* (`liftCharPointer2StringType`).

```
@GeneratedValueLifter: liftCharPointer2StringType
char*
```

Fig. 6. Annotating a generator template with a M2M lifting rule

IV. Case Study

mbeddr comes with an extension to declare and instantiate `components` and `mock components`, both illustrated by the example in Listing 4. We declare in this listing two `interfaces`, *ILogger* representing a logging service and *IAdder* for adding up two numbers. Further, we implement a `mock` *Logger* that `provides` the `interface` *ILogger* and contains a `sequence` modeling with `sequence steps` the order in which operation calls are expected. Next, we declare a `component` *Adder* that `requires` *ILogger* to log added values and `provides` an implementation of *IAdder* to add up two numbers. This `component` also contains `runnables`, which have `arguments`, a *return type*, a *body* (`statement list`) containing the implementation, and a `trigger`. While *setup* initializes the logger and acts as a constructor (`OnInit`), the other `runnable` is bound to the `provided port` and contains the C implementation to add up both `arguments`. To instantiate both `components`, we create an `instance configuration` that `connects` both instances based on their `provided` and `required ports`. Finally, we create a *main* `function` that invokes a `testcase` testing the *Adder* `component`. In this test, we initialize both components (`initInstances`) and invoke the *add* `operation` on the *adder* instance. Afterwards, we validate the result using `assertEquals` and verify the call sequence on *Logger* using `assertMock`.

With MuLDer, debugging support is always built per language construct. Hence, a debugger built with this framework consists of debugging implementations for different language constructs. In this case study we build debugging support for the `mock component` language. Because this language extends the mbeddr base language (C) and gets reduced to mbeddr's `components` language, we expect to have functioning debugging support for language constructs from these languages. Debugging support for all languages that are used on the source level and intermediate levels is a prerequisite of our approach. First, we define debugging semantics for `mock component` and `sequence step`. For other language constructs from the `components` language debugging semantics are already defined. `mock component` extends `component`, which already implements *Value Provider* and *Scope Provider*, hence, we do not require any additional interface implementations. Because `sequence steps` can be invoked, we implement *Callable* in the language construct returning the step index as *name* for contributed stack frames. Further, because `sequence steps` can contain an optional *body* in which stepping functionality can be used, we implement *Steppable Composite* and return the contained *body* in the required query. `statement list` comes from mbeddr C which already specifies the required interfaces.

Next, in Fig. 7 below we annotate the transformation rule for `mocks`, describing the program state lifting. First, we create `fields` (used for storing state) that track the number of *failed expectations* and overall *call counts*. Second, we

```
1  mock Logger {
2    provides ILogger logger
3    sequence {
4      0:logger.init
5      1:logger.log
6    }
7  }
8  instance configuration cfg {
9    Adder adder
10   LoggerMock logger
11   connect logger to adder
12 }
13 component Adder {
14   requires ILogger logger
15   provides IAdder adder
16   void setup() trigger OnInit {
17     logger.init();
18   }
19   int add(int a, int b) trigger adder.add {
20     logger.log("adding:", a, b);
21     return a + b;
22   }
23 }
24
25 int main() { return test[testAdd]; }
26 interface ILogger {
27   void log(string msg, int a, int b)
28 }
29 interface IAdder {
30   int add(int a, int b)
31 }
32 testcase testAdd {
33   initInstances cfg;
34   assertEquals cfg.adder.add(2,2) == 4;
35   assertMock cfg.logger
36 }
```

Listing 4. Illustrating the usage of `components` and `mocks` in a unit test

copy *content* (`COPY_SRCL`) from our `mock` to the `component`. Third, we generate a `runnable` being used by the generator for `assertMock` to request the number of *failed expectations*. Fourth, we generate at the bottom of the `component` for each `operation` of our `provided ports` a `runnable` that inherits the signature and has a `trigger` being bound to the `operation` and the associated `provided port`. Stack frames for these generated `runnables` are not lifted, instead, we generate for each `sequence step` a `statement list` being annotated with an *M2MOutlineFrame* annotation, outlining a stack frame for the `sequence step`. The specification shown at the bottom of Fig. 7 configures these stack frames: program counters for *outer stack frames* are redefined with the current *node* and we *associate* unwound stack frames with the higher level `sequence step`. The `statement list` we generate from `sequence steps` increments the *call count* and verifies that current and expected *call count* are equal, if not, we increment *failed expectations*. Further, we annotate the generated `component` with an *M2MGeneratedDelegateToValueProvider* annotation, referring to the *Value Transformation* shown in the middle. This *transformation* constructs a `complex-value` with the `mock` *name* as top-level value, whereas child values, *content* of kind *Watch Provider*, are lifted from `subvalues` of the current watch variable value.

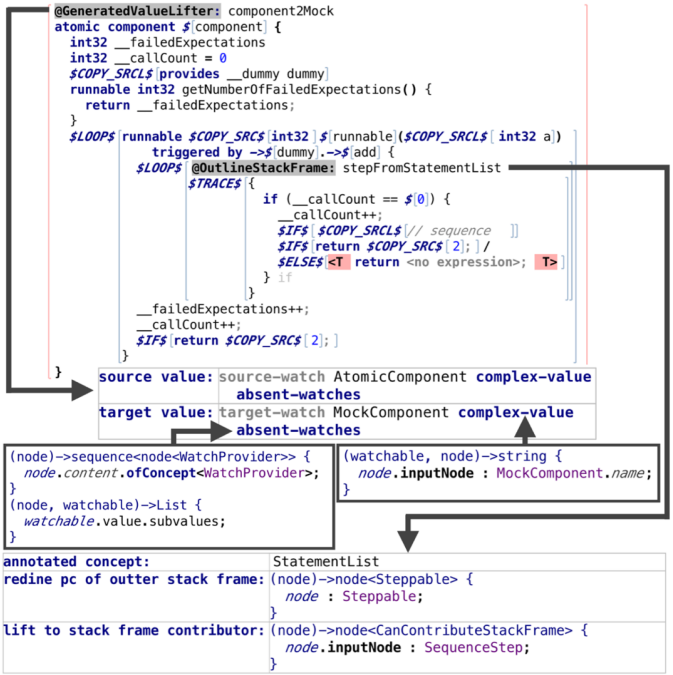

Fig. 7. Transformation rule for `mocks` and associated *Value Transformation*

V. Discussion

MuLDer is a language-oriented and incremental framework enabling multi-level debugging for extensible languages. By using the underlying approach, debugger developers specify debugging behavior in two steps. First, they describe *debugging semantics* of language constructs, e. g., for `callables` or `variables`, second, they annotate transformations with *rules* that appear hereby on intermediate level ASGs. Debuggers built with our approach use these rules at debug time to lift program state from a lower level back to the origin level, incrementally across intermediate levels. Because we describe these lifting rules inside transformations, we support multiple generators per language and multiple transformation rules per language construct. Consider the language extensions for `mock components` from Section IV, where we have described program state lifting from `components` back to `mock components`, ignoring how `components` are further translated towards the target level. Because we specify debugging behavior between generated and origin level, debuggers built with our approach are not affected by changes in lower level generators, an important requirement in the extensible language context. While MuLDer can be used to build debugging support for many extensible languages, the underlying approach has some limitations that we discuss next.

A. Statically Typed Languages

To describe the lifting of watch variable values, our approach requires `variables` to be associated with a type (*Value Provider*). This is no limitation for mbeddr, because the language is statically typed. However, due to this limitation we cannot support dynamically typed languages, e. g., JavaScript.

B. Stage-wise ASG Node Modifications

Output nodes of one transformation cannot be transformed by the same generator again, because we would lose information about code modification this way. Instead, when using our approach, these output nodes can only be modified by another code generator being executed afterwards.

C. Performance Overhead

The runtime behavior of lifting program state is driven by the number of instantiated *abstractions* (a), e. g., *Callables*, and the complexity of their associated *specifications* (s). Further, because we lift program state incrementally, the number of intermediate *levels* (l) is another factor that influences runtime behavior. Because we build interactive debuggers, runtime performance is a critical aspect, as users expect fast feedback from the tool. However, the debugging performance can dramatically decrease by increasing the program size over time and using more high-level languages.

VI. Related Work

A. Debuggers for Abstraction-Raising Languages

Renggli et al. [5] describe a source-level debugger for Helvetia, a tool that allows users to embed DSLs into Smalltalk host programs, extending Smalltalk with new syntax and semantics. While Helvetia translates DSL code directly to Smalltalk, our approach targets multi-stage transformations, reducing code incrementally to a base language. Further, we show program state based on the currently investigated abstraction level, while the Helvetia debugger shows this information in terms of the generated Smalltalk code.

MPS comes with an extensible Java and a debugger for this language. While debuggers built with our approach show the

program state and imitate stepping commands based on the currently investigated abstraction level, MPS' Java debugger shows the target-level call stack and directs each stepping command to the target level, not imitating the expected behavior.

We have built a source-level debugger framework for mbeddr that maps debug information directly between target and source level [6]. While MuLDer enables *encapsulated* debugger modules that are not affected by changes in lower level generators, debuggers built with the mbeddr framework depend on the structure of the generated target-level code. Hence, modifying lower level generators usually implies updating debuggers that have been built with this framework.

Mierlo [7] describes a debugging approach for modeling languages. With this approach, the modal behavior of a simulator for the language is described as a state chart, which is extended with debugging information. While this approach targets modeling languages with a fixed set of language constructs, our approach targets extensible languages. Further, the approach presented by Mierlo is used for debugging models on the source-level, while our approach allows multi-level debugging. Finally, Mierlo requires language engineers to describe the executable semantics by using state charts, while with our approach debugging semantics of language constructs are described and transformation rules are annotated.

B. Multi-Level Debuggers

Florisson [8] describes a multi-level debugger for Cython, a language allowing users to mix C and Python code in the same program. Such programs are compiled to C and integrated with a Python Application Programming Interface (API) for C, thus being accessible from Python. The resulting C code is further translated to a CPython extension module, which can afterwards be called from regular Python code. While code written with Cython interacts with Python code, users cannot debug it with a Python debugger. For this purpose, Florisson proposes a multi-level debugger that allows users to debug Python, Cython and C code simultaneously. While debugging calls from Python to Cython, the debugger follows control flow skipping the C abstraction level, such as calls to the Python interpreter (Python API). Thus, during stepping, the user will see Python code calling Cython code calling C code and can investigate the program state in terms of the respective language. Our approach also covers debugging mixed language programs, however, we support multiple abstraction levels and do not switch between them while performing a stepping command. Instead, our users switch manually between the various abstraction levels. Additionally, we target extensible languages, while Cython, Python and C have a fixed syntax.

Xia et al. [9] present in their work *multi-level debugging*, an approach for automatically detecting bugs in transformation rules. This approach is based on a set of error checking algorithms that analyze sequential and parallel aspects of programs created during transformation. While their approach automatically detects bugs, we allow users to debug their program execution on the source level and intermediate levels created by transformation rules.

Mannadiar and Vangheluwe [10] describe a debugger for a language that is used to model mobile applications. Because their modeling tool enables tracing across intermediate levels created during code generation, they can debug their model by inspecting on each intermediate level the currently involved model elements. While they provide tracing for intermediate representations, we allow interactive debugging on each level.

VII. Summary and Future Work

In this paper we have presented an incremental approach to build debuggers for extensible languages. While this approach enables source-level debugging for language users, it additionally allows language engineers to debug programs on intermediate levels to analyze bugs introduced by faulty transformation rules. Further, we have illustrated the MuLDer framework, an MPS-based implementation of this approach. We have used this framework to build a multi-level debugger for mbeddr and demonstrated in this paper how debugging behavior for `mock components` is implemented.

In the future, we plan to use MuLDer to build debuggers for other extensible languages and in other workbenches to evaluate its genericity. Additionally, we plan to extend the specification languages coming with MuLDer for specifying not only debuggers, but also interpreters to allow multi-level interpretation of extensible language programs.

References

[1] JetBrains, "Meta Programming System," http://jetbrains.com/mps, 2015.

[2] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb, "Mbeddr: An Extensible C-based Programming Language and IDE for Embedded Systems," in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH '12. New York, NY, USA: ACM, 2012, pp. 121–140.

[3] E. Bousse, J. Corley, B. Combemale, J. G. Gray, and B. Baudry, "Supporting efficient and advanced omniscient debugging for xdsmls," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*, R. F. Paige, D. D. Ruscio, and M. Völter, Eds. ACM, 2015, pp. 137–148.

[4] D. Pavletic and S. A. Raza, "Multi-Level Debugging for Extensible Languages," *Softwaretechnik-Trends*, vol. 35, no. 1, 2015.

[5] L. Renggli, T. Gîrba, and O. Nierstrasz, "Embedding Languages without Breaking Tools," in *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia*, ser. Lecture Notes in Computer Science, vol. 6183. Springer, June 2010, pp. 380–404.

[6] D. Pavletic, M. Voelter, S. A. Raza, B. Kolb, and T. Kehrer, "Extensible Debugger Framework for Extensible Languages," in *20th Ada-Europe International Conference on Reliable Software Technologies, Madrid Spain, June 22-26, 2015, Proceedings*, ser. Lecture Notes in Computer Science, vol. 9111. Springer, 2015, pp. 33–49.

[7] S. V. Mierlo, "Explicit modelling of model debugging and experimentation," in *Proceedings of Doctoral Symposium co-located with 17th International Conference on Model Driven Engineering Languages and Systems (2014), Valencia, Spain, September 30, 2014.*, ser. CEUR Workshop Proceedings, B. Baudry, Ed., vol. 1321, 2014.

[8] M. Florisson, "Multi-Level Debugging for Cython," *14th Twente Student Conference on IT*, vol. 14, no. 1, Jan. 2011.

[9] R. Xia, T. Elmas, S. A. Kamil, A. Fox, and K. Sen, "Multi-level Debugging for Multi-stage, Parallelizing Compilers," EECS Department, University of California, Berkeley, Tech. Rep., Dec 2012.

[10] R. Mannadiar and H. Vangheluwe, "Debugging in Domain-Specific Modelling," in *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 6563. Berlin, Heidelberg: Springer, 2010, pp. 276–285.