Extensible Debuggers for Extensible Languages

Domenik Pavletic[1], Markus Voelter[2], Syed Aoun Raza[3], Bernd Kolb[3], and Timo Kehrer[4]

[1] itemis, `pavletic@itemis.de`
[2] independent/itemis, `voelter@acm.org`
[3] itemis, `{raza,kolb}@itemis.de`.
[4] University of Siegen, Germany `kehrer@informatik.uni-siegen.de`

Abstract. Language extension enables integration of new language constructs without invasive changes to a base language (e. g., C). Such extensions help to build more reliable software by using proper domain-specific abstractions. Language workbenches significantly reduce the effort for building such extensible languages by synthesizing a fully-fledged IDE from language definitions. However, in contemporary tools, this synthesis does not include interactive debugging for programs written with the base language or its extensions. This paper describes a generic framework for extensible debuggers that enables debugging of the language extensions by definig mappings between the base language and the language extensions. The architecture is designed for extensibility, so debug support for future extensions can be contributed with little effort. We show an implementation of our approach for mbeddr, which is an extensible version of the C programming language. We also discuss the debugger implementation for non-trivial C extensions such as components. Finally, the paper discusses the extent to which the approach can be used with other base languages, debugger backends and language workbenches.

Keywords: Debugging, Domain-Specific Languages, Frameworks

1 Introduction

A program *source* is a description of the program *behavior*. Source and behavior are different: first, the source describes all dynamic execution behaviors of a program as a *static* structure. Second, the source considers all possible sets of input data. However, program execution always happens for a specific set of input values. Debugging helps programmers to *inspect* and *animate* the dynamic behavior of a program for a *specific set* of input values.

The way this is done depends on the language paradigm. Imperative languages, the focus of this work, use the step-through approach. Thus, users single-step through instructions and observe changes to the program state.

Programming languages such as C, Java or Python contain a *fixed* set of language construct and cannot easily be extended. The debuggers for such languages can be hand-crafted specifically for the constructs provided by the language. In contrast, modern language engineering allows the development of *extensible* languages [14, 13]. This allows users to add new constructs to a language in an incremental and modular way. These languages are used to describe solutions for technical or domain-specific problems on a higher level of abstraction. This approach supports domain-specific validation and verification of developed system at the same abstraction level as the problem. Thus it delivers improvements in reliability.

The concepts (structure definition) introduced by a language extension are translated to semantically equivalent base language code before compilation. For example, a `foreach` statement that supports iterating over a C array without manually managing the counter variable. This statement would be translated back to a regular C `for` statement. The transformation generates the code that manages the counter (see our example in Section 3).

To make debugging extensible languages useful to the language user, it is not enough to debug programs *after* extensions have been translated back to the base language (using an existing debugger for the base language). A debugger for an extensible language must be extensible as well, to support debugging of modular language extensions *at the extension-level*. Minimally, this means that users can step through the constructs provided by the extension and see watch expressions related to the extensions. In the `foreach` example, the user would see the `foreach` statement in the source code. Furthermore, the generated counter variable would not be shown in the watch window.

In this paper, we contribute a framework for building debuggers for extensible, imperative languages. With this framework, each language extension is debugged at its particular abstraction level. We illustrate the approach with an implementation based on mbeddr [13], an extensible C build with the JetBrains Meta Programming System (MPS) [8]. For a non-trivial C extension we show an example debugger extension. Further, we discuss whether and how the approach can be used with other extensible languages and language workbenches.

This paper is structured as follows: Section 2 provides an overview of the mbeddr technology stack which is the basis of our reference implementation. In Section 3, we introduce an example language extension for which we describe the debugger extension in Section 6. Section 4 lists the requirements for our extensible debugger and Section 5 describes the essential building blocks of the architecture. We validate our approach by discussing debuggers for non-trivial extensions of C in Section 7. In Section 8, we discuss the benefits, trade-offs and limitations of our approach. We then look at related work in Section 9 and conclude the paper with a summary and an outlook on future work in Section 10.

2 The mbeddr Technology Stack

mbeddr is an extensible version of C that can be extended with modular, domain-specific extensions. It is built on top of JetBrains MPS and ships with a set of language extensions dedicated to embedded software development. This section provides a brief overview of mbeddr; details can be found in [13].
Fig. 1 shows how mbeddr is organized into layers. The foundation is MPS, which supports language definition. Its distinctive feature is its projectional editor, which unlike textual editors, does not rely on a parser. Instead, the visual notation is a *projection* of the Abstract Syntax Tree (AST). This means, every change performed by the user is *directly* reflected in the AST.

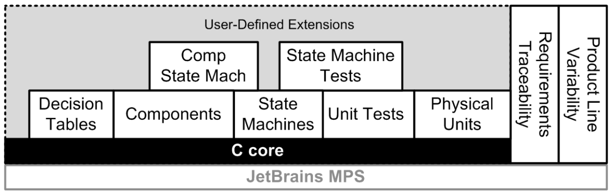

Fig. 1. An overview of the mbeddr stack.

The second layer (C Core) provides an implementation of C99[7]. The next layers contain the default language extensions, generally useful for embedded software development. These include languages for specifying test cases, state machines, interfaces/components and decision tables.

3 Language Extension Examples

This section illustrates how to extend mbeddr with a `foreach` statement that allows programmers to iterate over arrays. The code below shows an example:

```
int8[] numbers = {1, 2, 3};
int32 sum = 0;
foreach (numbers sized 3) { sum += it; }
```

Defining a language or extension in MPS comprises the following steps: structure (syntax or meta model), the concrete syntax (editor definition in MPS), the type system (for defining and checking typing rules), a generator (for mapping extensions to the base language), various IDE features such as quick fixes or refactorings, and of course the debugger. This section only discusses aspects important for debugger definition. For a more detailed discussion of language extension we refer to [13]. The debugger extension itself is shown in Section 6.

A particularly important feature of MPS as a language workbench is its support for *modular* language extension, where an extension lives in its own language module. While such a module may depend on (and use concepts from) a base language, it cannot invasively change this base language.

In terms of structure, our `foreach` language consists of a `ForeachStatement` and an `ItExpression` (dark grey boxes in Fig. 2). `ForeachStatement` extends `Statement` to make it usable wherever C allows `Statements`. It consists of three children: an `Expression` for the array, an `Expression` for the array length, and a `StatementList` for the body. `Expression`, `Statement` and `StatementList` are defined in C and reused by `ForeachStatement`. `ItExpression` represents the current element and extends C's `Expression` to make it usable where expressions are expected.

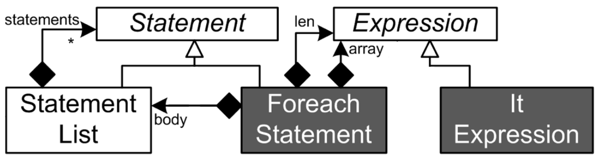

Fig. 2. UML class diagram showing the structure of the language extensions. Concepts from C are colored in white, others from `foreach` are dark grey.

The details on the transformation to C are described in [13]. For implementing the debugger extension in Section 6, it is important to understand the structure of the generated code. The code snippet below shows an example usage of `foreach` (left column) and the corresponding generated C code (right column).

```
int8 sum = 0;
int8[] sortedNumbers = {1, 2, 3};
foreach (sortedNumbers sized 3) {

  sum += it;
}
```

```
int8_t sum = 0;
int8_t[] sortedNumbers = {1, 2, 3};
for (int __c = 0; __c < 3; __c++) {
  int8_t __it = sortedNumbers[__c];
  sum += __it;
}
```

4 Requirements on the Debugger

Debuggers for extensible languages should provide the same functionality as the corresponding base language debugger. This includes debug commands (stepping and breakpoints) and inspection of the program state (watches and call stack).

In general, the execution of a program is debugged by a base language debugger (e. g., `gdb` in case of C). To enable debugging on the abstraction level of extensions, a mapping must be implemented between the base language debugger and the program as represented on the extension-level. Fig. 3 illustrates the relationship and information flow between the extension and base-level debugging mechanism: stepping must be mapped from the extension-level to the base-level and the program state must be represented in terms of the extension-level. This methodology is also applicable to hierarchical language extensions.

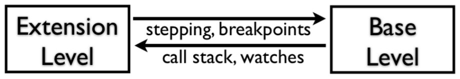

Fig. 3. Debugging interactions between extension- and base-level.

MPS allows the composition of multiple independently developed language extensions in a single program [12]. mbeddr capitalizes on this feature. Hence, programs are typically a mix of code written in C and in several language extensions (the code snippet above uses concepts from C and the `ForeachLanguage`). A debugger for such a language workbench should provide the capability to debug mixed-language programs. Considering this, a debugger for extensible languages should support the following general requirements GR1 through GR3:

GR1 Modularity: Language extensions are modular, so debugger extensions must be modular as well. This means, to enable debugging for programs developed with a particular language extension, no changes to the base language must be necessary. Also, the debugger definitions for independently developed language extensions must be usable together in a single program.

GR2 Framework Genericity: In addition, implementing debug support for new language extensions must not require changes to the debugger framework or its interfaces (not just to other languages, as described in GR1).

GR3 Ease of Extensibility: Language workbenches make language development relatively simple. So, to make an extensible debuggers fit in, the development of debugger extensions must be comparatively simple as developing the language itself.

Depending upon the nature of the base language and its extensions, there can be additional requirements for a debugger. mbeddr addresses embedded software development, which leads to the specific requirements shown below. In other domains, these concerns may be useful as well, whereas for embedded software they are essential.

ER1 Limited Overhead: In embedded software, runtime overhead is always a concern. So the framework should limit the amount of additional debugger-specific code generated into the executable. Additional code increases the size of the binary, potentially preventing debugging on target devices with small amount of memory.

ER2 Debugger Backend Independence: Depending upon the target device vendor, embedded software projects use different C debuggers. Re-implemention of the debugger logic is cumbersome, hence debugger backend independence is crucial.

5 General Debugger Framework Architecture

The framework architecture can be separated into the specification aspect (Section 5.1) and the execution aspect (Section 5.2). The *specification* aspect declaratively describes the debug behavior of language concepts and enables modular extension of the debugger (GR1). The *execution* aspect implements the extensible debugger framework in a generic and reusable manner (GR2).

5.1 Specification Aspect

The debugger specification is based on a set of abstractions that can be split into four main areas: breakpoints, stepping, watches and stack frames (see Fig. 4). Debuggers are defined by mapping concepts from the base language and language extensions to these abstractions. To map a language concept to any of these abstractions, it implements one or more of these interfaces (other language workbenches may use other approaches [9]). To specify the concept-specific implementation behavior of such an interface, mbeddr provides a DSL (Domain-Specific Language) for debugger specification (GR3; the DSL is shown in Section 6). This way the approach facilitates modularity (GR1) by associating the debug specification directly with the respective language concept.

Breakpoints Breakpoints can be set on `Breakable`s. In imperative languages, breakpoints can be set on statements so they will be mapped to `Breakable`.

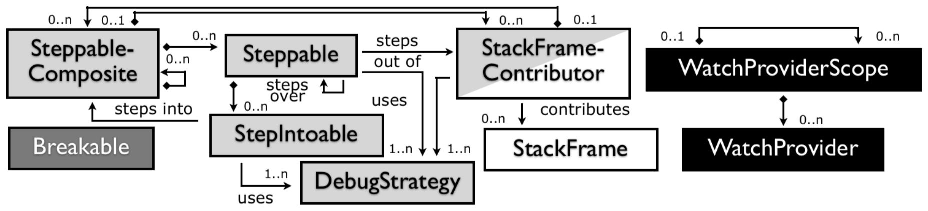

Fig. 4. Abstractions for the debugger specification

Stack Frames `StackFrameContributor`s are concepts that are translated to base-level callables (functions or procedures) or represent callables on the extension-level. They contribute `StackFrame`s, each is linked to a base-level stack frame and states whether it is visible in the extension-level call stack or not.

Stepping `Steppable` concepts support stepping, which is separated into *step over*, *step into* and *step out*. For *step over*, a `Steppable` defines where program execution must suspend next, after the debugger steps over an instance of `Steppable`. Again, statements are the typical examples of `Steppables` in an imperative language. If a `Steppable` contains a `StepIntoable` under it, then the `Steppable` also supports *step into*. `StepIntoable`s are concepts that branch execution into a `SteppableComposite`, typically requiring its own stack frame via `StackFrameContributor`. The canonical example in programming languages are function calls (`StepIntoable`) and the corresponding functions (`SteppableComposite`). Once we are in a `SteppableComposite` we may want to *step out*; hence a `StackFrameContributor` contributes *step out* behavior.

All *stepping* is implemented by setting breakpoints and then resuming execution until one of these breakpoints is hit (this approach is adopted from [16]). The actual stepping functionality of the underlying debugger is not used. `Steppable`s use `DebugStrategies` that determine where to set the base-level breakpoints to implement a particular stepping behavior for an extension concept. Language extensions can implement their own strategies (GR2) or use predefined ones (GR3). For instance, to prepare for *step over* on the extension-level, the `SingleBreakpoint` strategy retrieves for a node the first line of generated base-level code and sets a breakpoint on it. This way no further dependencies to other extensions are required, and extensions remain modular (GR1).

Watches `WatchProvider`s can contribute entries to the debugger's watch window. For example a local variable declaration whose value can be inspected in the watch window. A `WatchProviderScope` is a nestable context in which `WatchProvider`s are declared and are valid e. g., two functions declaring local variables. When suspending within any of them, the corresponding `WatchProviderScope` causes the debugger to only show its local variables.

5.2 Execution Aspect

The framework relies on traces and the AST to provide the debug support. This means no debugger-specific code should be generated into the executable (ER1). Fig. 5 shows the components of the execution aspect and their interfaces to implement this AST/trace-based approach: `ProgramStructure` provides access to the AST via `IASTAccess`. `Trace Data` provides `ITraceAccess` that is used to locate the AST node (base and extension-level) that corresponds to a segment or line in the generated base-language code, and vice versa. The `Low-Level Debugger` represents the native debugger for the base language. It provides the `ILLDebuggerAPI` interface, which is used to interact with the `Low-Level Debugger` (ER2). These interactions include setting breakpoints, finding out about the program location of the `Low-Level Debugger` when it is suspended at a breakpoint and access watches. Languages provide `Debugger Extensions` (GR3), based on the abstractions discussed in Section 5.1. The `IDebugControl` interface is used by the `Debugger UI` to control the `Mapper` which integrates the other components. For example, `IDebugControl` provides a `resume` operation, `IBreakpoints` allows the UI to set breakpoints on program nodes and `IWatches` lets the UI retrieve the data items for the watch window. All these interfaces are used by the `Mapper`, which controls the `Low-Level Debugger` and is invoked by the user through the `Debugger UI`. The `Mapper` uses the `Program Structure`, the `Trace Data` and the debugging implementation from the `Debugger Extensions`.

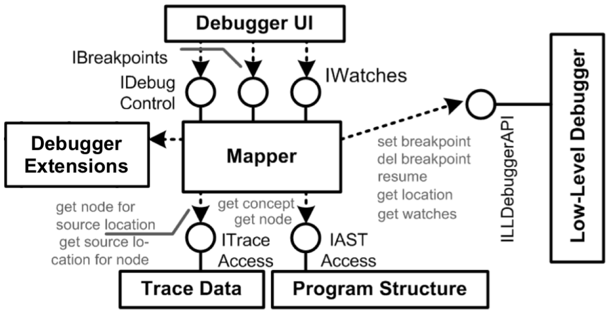

Fig. 5. The components of the execution aspect and their interfaces

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

– Ask the current `Steppable` for its *step over* strategies; these define all possible locations where the debugger may have to break after the *step over*.
– Query `TraceData` for corresponding lines in the generated C code for those program locations.
– Set breakpoints via `ILLDebuggerAPI` on those lines in the low-level code.
– Use `ILLDebuggerAPI` to resume program execution. It will stop at any of the just created breakpoints.
– Use `ILLDebuggerAPI` to get the low-level call stack.
– Query `TraceData` to find out for each C stack frame the corresponding nodes in the extension-level program.
– Collect all relevant `StackFrameContributor`s (see next section). The `Mapper` uses them to construct the extension-level call stack.
– Get the currently visible symbols and their values via `ILLDebuggerAPI`.
– Query the nodes for `WatchProvider`s and use them to create watchables.

At this point, execution returns to the `Debugger UI`, which then gets the current location and watchables from the `Mapper`. With this information, it highlights the `Steppable` on which execution is suspended and populates the watch window.

6 Example Debugger Extension

After discussing the architecture, this section now shows the implementation of a debugger extension for the language described in Section 3. The specification of the extension resides in the respective language module (GR1). As part of the mbeddr implementation, we have developed a DSL for debugger specification (GR3) that integrates directly with MPS' language definition language. We use this DSL for implementing the debugger extension in this section.

Breakpoints We want to be able to set breakpoints on a `foreach` statement. Since both concepts are derived from `Statement`, we can already set breakpoints on them; no further work is necessary.

Stack Frames The `foreach` statement does not have callable semantics and is not generated to C functions (see Section 5.1). Hence, suspending the debugger in a `foreach` statement affects not the number of stack frames in the call stack.

Stepping `ForeachStatement` is derived from `Statement`. Its implementation for *step over* suspends the debugger on the `Statement` following the current one. `Statement` realizes this by delegating the *step over* request to its surrounding `SteppableComposite` (`ancestor` in the AST). This `SteppableComposite` simply sets a breakpoint on the next `Steppable`, which is again a `Statement`:

```
void contributeStepOverStrategies() { ancestor; }
```

We must overwrite this default behavior for the `ForeachStatement`, since its stepping behavior differs. Consider we suspend our debugger on the `foreach` that is shown in last snippet of Section 3 and perform a *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;`). This is the first line of the mbeddr program, *not* the first line of the generated base program (which would be `int8_t __it = sortedNumbers[__c];`). The debugger cannot guess which alternative will occur since it would require knowing the state of the program and evaluating the expressions in the (generated) `for`. Instead, we set breakpoints *on all possible next statements* and then resume execution until we hit one of them — the created breakpoints are then removed again. The code below shows the implementation: we delegate the request to the surrounding `SteppableComposite`, but we also set a breakpoint on the first statement in the `body` (if not empty).

```
void contributeStepOverStrategies() {
  delegate to ancestor;
  break at this.body.first;  }
```

Let us look at *step into*. Since the `array` and `len` expressions of a `foreach` 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`, `len` and find any expression that can be stepped into. Such expressions implement `StepIntoable`. The following code shows the *step into* implementation:

```
void contributeStepIntoStrategies() { inspect this.array, this.len for StepIntoables; }
```

Watches By default, the watch window contains all C symbols (global and local variables, arguments) as supplied by the C debugger. In case of `foreach`, this means the `it` expression is not available, but the two generated variables `__it` and `__c` are. This is exactly the wrong way: the watch window should show `__it` as `it` and hide `__c`. The code below shows the implementation in `foreach`:

```
1 void contributeWatchables() {
2   hide local variable with identifier "__c"
3   map by name "__it" to "it"
4     type mapper: this.array.type
5 }
```

Line 2 `hide`s the C variable `__c`. The rest `map`s a base-level C variable to a watchable. It finds the C variable named `__it` inserted by the `foreach` generator, hides it and creates a watch variable named `it`. The type of `it` is the base type of the array over which we iterate. This type is responsible for mapping the value (`type mapper`), in our example above the `int32` type simply returns the C representation of an `int32_t` value.

To complete the implementation, we must implement `WatchProviderScope` in `foreach`. This requires an implementation of `collectWatchProviders` to collect instances of `WatchProviders` in a scope. `foreach` owns a `StatementList` which implements this interface already and collects `WatchProviders` in the top-level scope of the body. Hence, a `foreach` simply contributes itself (for hiding `__c` and mapping `__it`), which is expressed with the following implementation:

```
void collectWatchProviders() { collect watch providers: this; }
```

7 Validation

To validate our approach, we have implemented the debugging behavior for mbeddr C and most of its default extensions (components, state machines and unit testing). This section discusses some interesting cases we have experienced during implementation.

7.1 Polymorphic Calls

There are situations when static determination of a *step into* target is not possible e. g., polymorphic calls on interfaces. Our components extension provides interfaces with operations, as well as `component`s that `provide` and `require` these interfaces. The `component` methods that implement interface operations are generated to base-level C functions. The same interface can be implemented by *different* `component`s, each implementation ending up in a *different* C function. A client `component` only specifies the *interface* it calls, not the `component`. So, we cannot know statically which C function will be called if an operation is invoked on the interface. However, statically, we can find all `component`s that implement the interface (in a given executable), so we know *all possible C functions* that may be invoked. A strategy specific for this case (GR2) sets breakpoints on the first line *of each of these functions*. Consequently, we stop in any of them if the user *steps into* an operation invocation.

7.2 Mapping to Multiple Statements

In many cases a single statement on the extension-level is mapped to several statements or blocks on the base-level. So *stepping over* the extension-level statement must step over the block or list of statements in terms of C. An example is the `assert` statement (used in tests) which is mapped to an `if`. The debugger has to step over the complete `if`, independent of whether the condition in the `if` evaluates to `true` or `false`. Note that we get this behavior for free: we never step actually over statements. In contrast, we set breakpoints at all possible code locations where the debugger may have to stop next. The `assert` statement sets a breakpoint on the base-level counterpart of the *next extension-level statement*.

7.3 Datatype Mapping

Language extensions may provide new data types in addition to the existing base language data types. During code generation, these additional data types are translated to the base language data types. In mbeddr, a trivial example for this is the `boolean` type, which is translated to C's `int` type. When inspecting the value of a watchable that is of type `boolean` we expect the debugger to render the `int` value either as `true` or `false`.

For mbeddr's `components` a more complex mapping is needed. As shown in the listing below, `components` can contain declarations for fields (instance variables e. g., *color*) and provided/required ports (interfaces e. g., *tl* and *driver*). The code generator translates each `component` (e. g., *TrafficLights*) to a `struct` declaration (e. g., *CompModule_compdata_TrafficLights*). This `struct` declaration contains members (e. g., *field_color* and *port_driver*) for the declared fields (e. g., *color*) and for each required port (e. g., *driver*).

```
component TrafficLights extends nothing {
  provides ITrafficLights tl
  requires IDriver driver
  TLC color;
  void setColor(TLC color) op tl.setColor {
    color = color;   }
}
```

```
struct CompModule_compdata_TrafficLights {
    /* fields */
    CompModule_TLC field_color;
    /* required ports */
    void* port_driver;

};
```

When debugging a `component` instance on the extension-level, we expect the debugger to provide watches for the fields. They should have their respective extension-level values and names. However, the members for the ports should not be displayed. In the mapping implementation of `component` we must therefore extract the fields from the respective `struct` instance and map the names and their respective values.

8 Discussion

This section revisits the requirements outlined in Section 4 to evaluate to what extent they are fulfilled:

8.1 Revisiting the Requirements

GR1 Modularity Our approach 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 specifications. MPS' capability of incrementally including language extensions in a program *without defining a composite language first* is preserved in the face of debugger specifications.

GR2 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 querying the type (e. g., consider our `boolean` type example).

GR3 Simple Debugger Definition This challenge is solved by the debugger definition DSL. It supports the declarative specification of stepping behavior and watches. However, it does not concern the user with implementation details of the framework or the debugger backend.

ER4 Limited Overhead Our solution requires no debugger specific code at all (except debug symbols added by compiling the C code with debug options). Instead, we rely on trace data to map the extension-level to the base-level and ultimately to text. This is a trade-off since the language workbench must be able to provide trace information. Also, the generated C code cannot be modified by a text processor before it is compiled and debugged. This would invalidate the trace data (the C preprocessor works, it is handled correctly by the compiler and debugger). On the contrary, we are not required to change existing transformations to generate debugger-specific code. This keeps the transformations independent of the debugger.

ER5 Debugger Backend Independence We use the Eclipse CDT Debug Bridge [6] to wrap the particular C debugger. This way, we can use any compatible debugger without changing our infrastructure. Our approach requires no changes to the native C debugger itself. However, since we use breakpoints for stepping, the debugger must be able to handle a reasonable number of breakpoints. The debugger also has to provide an API for setting and removing breakpoints and for querying the currently visible symbols and their values. In addition, the API should allow us to query the code location where the debugger is suspended. Most C debuggers support all of this, so these are not serious limitations.

8.2 Other Evaluation Criteria

In addition to the specific requirements form Section 4, our approach can be evaluated with regards to additional criteria.

Sizes and Efforts The `ForeachLanguage` consists of 70 lines of code. 17 of them (25%) are related to the debugger. For the much more complex components extension, the ratios are similar, although the language is ca. 2.500 lines of code. We do not have numbers for the default extensions itself, since their debugging implementation was mixed with the development of the debugger framework. From these numbers we conclude that we have reached the goal of the debugger specification for extensions not requiring significantly more effort than the definition of the extension itself.

Multi-Staged Transformations The extensions described so far are *directly* transformed to C. However, extensions may also be transformed to other language extensions. Thus, forming a multi-level stack with high-level languages on top and low-level languages at the bottom. Our current framework implementation provides debug support for such multi-level extensions as well. However, for high-level extensions, the debugger must be specified relative to C and *not* relative to the next lower-level extension. This is a limitation, since all transformations between the extension and the base language must be considered.

Use for other Base Languages The framework was designed and implemented for mbeddr C. However, it contains no dependencies on mbeddr or on the C implementation. The framework only assumes that the base language and its extensions use the imperative paradigm with statements that can be nested and embedded into callables. Consequently, debug support for other imperative base languages can be implemented using our framework.

Use outside of MPS Our *implementation* cannot be used outside of MPS since it depends on MPS' APIs. However, the general *approach* can be adapted to other language workbenches. According to Fig. 5, the tool has to provide the following services: a debugger UI that lets users interact with the debugger, access to the program and the language definitions as well as trace data.

9 Related Work

This section provides an overview of related research. We look at compile-time debuggers, extensible debuggers and DSL debuggers.

Compile-time Debuggers Our approach animates and inspects the execution of a program. Other debuggers inspect the compilation or macro expansion phase. Examples include Porkoláb's debugger for C++ template metaprogram compilation [11]. Furthermore, Culpepper's stepper for the macro expansion process in Scheme [4]. In mbeddr, the equivalent of compile time meta programs or macros is the transformation of extensions to base language code. While not discussed in this paper, MPS provides various ways of debugging the transformation process. This includes retaining all intermediate programs for inspection as well as a debugger for stepping through the transformation process itself.

Cornelissen [3] describes an approach to enable debugging at two meta-levels at the same time. Their work is based on the TIDE debugger framework [2] and the ASF+SDF Meta-Environment [5]. One debugger debugs the execution of a program written in the Pico language. At the same time, the interpreter defining the semantics of the Pico language can be debugged as well. This is different from our approach. We support integrated debugging of programs expressed at different *abstraction* levels (C base language and the various extensions). In contrast, Cornelissen supports debugging at different *meta*-levels.

Extensible Debuggers Extensibility can address different aspects of a debugger, not just the extensibility of the base language as discussed in this paper. Vraný and Píse describe a debugger that integrates information from multiple different debuggers into a common debugger UI. Consequently, providing an integrated execution flow [15]. This approach is similar to ours in that it considers several languages. However, in our case, the languages are mixed within the same program, extend a common base language and run in a single debug process.

The debugger described by Al-Sharif and Jeffery [1] can be extended with user-defined program execution monitors. These monitors run concurrently, listen to debug events, such as `breakpoint hit`, and process them in user-definable ways. May et al. introduce a debugger for parallel applications [10]. They provide an API to integrate custom views. Those views can also access debug sessions, but additionally, they contribute components to the debugger UI. Both works address extending the *debugging functionality* for a fixed language. However, our debugger provides a fixed functionality, but for extensible *languages*.

DSL Debuggers Wu et al. introduce a debugging framework for imperative, declarative and hybrid DSLs [17, 16]. They integrate `print` statements into the generated code that output debug symbols, such as values of watches or the current source line. Based on this output, the debugger renders the UI. Our debugger stores the mapping between high-level nodes and generated code in a trace data structure. This information is created during the transformation process. With this approach, we avoid instrumenting the program code, an important requirement for us (ER4). In the same work, Wu et al. also describe a debugger for a simple assembler-like language. To implement step *in* and step *over* in this language, they introduce the idea of using breakpoints. While they use a mix of native stepping and breakpoints, we adapted their approach to use *only* breakpoints. In addition, we add support for *step out*, based on a call stack.

Lindeman et al. introduce a generic debugger framework for DSLs [9]. Using a debugger specification DSL, developers create *event types* and map the syntax and semantics of a language onto an execution model. This is similar to our approach, as we also provide a DSL to map language concepts to a execution model (see Section 5.1). In Lindemann's approach, a preprocessor integrates debug symbols into the DSL program, based on the specified event types. In contrast, our debugger uses external trace data to relate extension-level programs to base-level programs, and ultimately to lines in the generated code.

10 Summary and Future Work

Extensible languages are a good compromise between general-purpose and domain-specific languages because they can grow towards a domain incrementally. In addition, they enable validation and verification on the problem-level, by reducing propagation of possible errors into base language code. To alleviate rest of the errors, extension-level debugging is helpful. In this paper, we have introduced a debugger framework for an extensible language. Furthermore, we have demonstrated the feasibility by implementing debugging support for non-trivial extensions for the mbeddr C language. The requirements regarding extensibility, modularity and limitation of overhead outlined at the beginning of the paper have all been satisfied. Further, efforts for implementing debugger extensions fit well with the efforts for building the language extensions themselves.

In future, we will investigate the extent to which multiple alternative transformations for a single language concept require changes to the debugger framework. We will explore synergies between the debugger and other language definition artifacts such as transformations (watches) and the data flow graph (stepping). Finally, we will investigate improved support for multi-staged transformations.

References

[1] Al-Sharif, Z., Jeffery, C.: An Extensible Source-Level Debugger. In: Proceedings of the 2009 ACM Symposium on Applied Computing. pp. 543–544. ACM, Honolulu, Hawaii, USA (2009)

[2] Van den Brand, M.G.J., Cornelissen, B., Oliver, P.A., Vinju, J.J.: TIDE: A Generic Debugging Framework - Tool Demonstration. In: Electronic Notes in Theoretical Computer Science. vol. 141, pp. 161–165. Edinburgh, UK (2005)

[3] Cornelissen, B.: Using TIDE to Debug ASF+SDF on Multiple Levels. Master's thesis, University of Amsterdam, Netherlands (2005)

[4] Culpepper, R., Felleisen, M.: Debugging Macros. In: 6th International Conference on enerative Programming and Component Engineering. pp. 135–144. ACM, Salzburg, Austria (2007)

[5] van Deursen, A., Dinesh, T.B., van der Meulen, E.: The ASF+SDF Meta-Environment. In: Proceedings of the 3rd International Conference on Methodology and Software Technology. pp. 411–412. Springer, Enschede, Netherlands (1993)

[6] Eclipse Foundation: Eclipse CDT (2015), `http://www.eclipse.org/cdt`

[7] International Organization for Standardization (ISO): ISO C 99 Standard (1999), `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`

[8] JetBrains: Meta Programming System (2015), `http://www.jetbrains.com/mps`

[9] Lindeman, R.T., Kats, L.C., Visser, E.: Declaratively Defining Domain-specific Language Debuggers. In: 10th Conference on Generative Programming and Component Engineering. pp. 127–136. ACM, New York, NY, USA (2011)

[10] May, J., Berman, F.: Panorama: A Portable, Extensible Parallel Debugger. In: 3rd Workshop on Parallel and Distributed Debugging. pp. 96–106. ACM, San Diego, California, USA (1993)

[11] Porkoláb, Z., Mihalicza, J., Sipos, A.: Debugging C++ Template Metaprograms. In: 5th Conference on Generative Programming and Component Engineering. pp. 255–264. ACM, New York, NY, USA (2006)

[12] Voelter, M.: Language and IDE Development, Modularization and Composition with MPS. In: 4th Summer School on Generative and Transformational Techniques in Software Engineering. pp. 121–140. Springer, Braga, Portugal (2011)

[13] Voelter, M., Ratiu, D., Schaetz, B., Kolb, B.: Mbeddr: An Extensible C-based Programming Language and IDE for Embedded Systems. In: Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12. pp. 121–140. ACM, New York, NY, USA (2012)

[14] Voelter, M., Visser, E.: Language Extension and Composition with Language Workbenches. In: Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA. pp. 301–304. ACM, New York, NY, USA (2010)

[15] Vraný, J., Píse, M.: Multilanguagee Debugger Architecture. In: 36th International Conference on Current Trends in Theory and Practice of Computer Science. pp. 731–742. Springer, Špindlerův Mlýn, Czech Republic (2010)

[16] Wu, H.: Grammar-Driven Generation of Domain-Specific Language Testing Tools. In: Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005. pp. 210–211. ACM, San Diego, California, USA (2005)

[17] Wu, H., Gray, J.G., Roychoudhury, S., Mernik, M.: Weaving a Debugging Aspect into Domain-Specific Language Grammars. In: Proceedings of the 2005 ACM Symposium on Applied Computing. pp. 1370–1374. ACM, Santa Fe, New Mexico, USA (2005)