Extensible Debuggers for Extensible Languages

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

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

Abstract

Language workbenches significantly reduce the effort for building extensible languages. However, they do not facilitate programmers with built-in debugging support for language extensions. This paper presents an extensible debugger architecture that enables debugging of language extensions. This is established by defining mappings between the base language and the language extensions. We show an implementation of our approach for the mbeddr language workbench.

1 Introduction

Debuggers for general purpose languages (e. g., C or Python) can be hand-crafted specifically for the constructs provided by the language, due to a *fixed* set of language contructs. In contrast, modern language engineering allows the development of *extensible* languages [3], such as mbeddr [2], where users can add new constructs in an incremental and modular way. The constructs introduced by a language extension are usually translated to semantically equivalent base language code during compilation. For example, a `foreach` statement that supports iterating over a C array without a counter variable, would be translated to a C `for` statement. Vice versa, a `for` statement can be reengineered to a `foreach`.

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 and the generated counter variable would not be shown in the watch window.

This paper contributes a framework for building debuggers for extensible, imperative languages, where each language extension is debugged at its particular abstraction level.[1] We illustrate the approach in mbeddr, an extensible version of C implemented with JetBrains MPS (`http://jetbrains.com/mps`).

[1]This work is developed as part of the LWES project, supported by the German BMBF, FKZ 01/S11014.

2 Requirements on the Debugger

Debuggers for extensible languages should provide the same functionality as the corresponding base language debugger. In particular, this includes debugging 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 debugger for the base language (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. Figure 1 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.

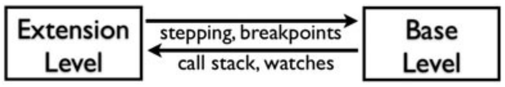

Figure 1: Debugging interactions

GR1 Modularity: As language extensions, debugger extensions must be modular and composable.
GR2 Genericity: New language extensions must not require changes to the framework.
GR3 Ease of Extensibility: Implementing debugging support should be relatively simple as developing language extensions with language workbenches.

In addition, our domain (embedded software) leads to the following requirements:
ER1 Limited Overhead: To reduce runtime and memory overhead, the amount of debugger-specific code generated into the executable should be limited.
ER2 Debugger Backend Independence: Due to target device dependent compilers and debuggers, different C debuggers should be supported.

The next sections of this paper describe how the proposed framework and its implementation for mbeddr addresses these requirements.

3 Debugger Framework Architecture

The architecture can be separated into the *specification* aspect: declarative description of language debug behavior and *execution* aspect: generic and reusable implemention of the extensible debugger framework.

3.1 Specification Aspect

The debugger specification is based on four groups of abstractions: *breakpoints*, *stepping*, *watches* and *stack frames*. To achieve modularitry (GR1) for each debugger extension, language constructs are mapped to these abstractions (GR2). In mbeddr, this mapping is realized with a debugger specification DSL (GR3).

3.2 Execution Aspect

The execution aspect relies on traces and the AST to provide debug support. This means, no debugger-specific code is generated into the executable (ER1).

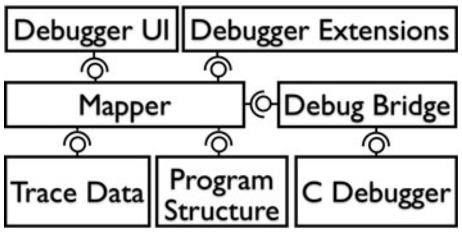

Figure 2: Execution Aspect

Figure 2 shows the components, which implement this AST-based approach: `Program Structure` provides access to the AST. `Trace Data` is used to find out the AST node (base and extension-level) that corresponds to a segment or line in the generated base-language code, and vice versa. `Debug Bridge` provides a common API for different C Debuggers and is implemented by the `Eclipse CDT Debug Bridge`[1] (ER2). Languages contribute `Debugger Extensions` (GR3), based on the abstractions discussed in Section 3.1. Users interact with the debugger via the `Debugger UI`, which serves as a frontend and is integrated into MPS. Finally, `Mapper` contains the algorithms for mapping program state and debug commands, thus integrating the other components.

4 Implementing Debugger Extensions

In this section we discuss some scenarios from mbeddr where we have implemented debugger extensions using the previously described framework.

4.1 Polymorphic Calls

Performing a *step into* on a function call suspends the debugger within the called C function. In mbeddr, language extensions can provide additional constructs with function semenatics (a *callable*) e. g., components or state machines, which require the same behavior. However, when performing a *step into* on a callable, it is statically hard to determine which actual callable will be executed.

In mbeddr, the *components* extension provides interfaces with operations, as well as `components` that `provide` and `require` these interfaces. The `component` methods that implement these operations are generated to base-level C functions. An interface can be implemented by *different* `components`, each implementation ending up in a *different* C function. In order to enable *step into* behavior, we have to set breakpoints in each possiblly called C function.

4.2 Mapping to Multiple Statements

In many cases an extension-level statement is mapped to several statements or blocks on the base-level. So *stepping over* the extension-level statement must step over the whole block or list of statements in terms of C. An example is the `assert` statement (used in test cases) 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`.

4.3 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 `boolean` type 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 instance variables (*color*) and provided/required ports (interfaces *tl* and *driver*). The code generator translates `components` (*TLights*) to `struct` declarations (*TLMod_comp_TLights*) with members: (*field_color* and *port_driver*) for the declared instance variables (*color*) and for each required port (*driver*).

```
component TLights {                      struct TLMod_comp_TLights {
  provides ITrafficLights tl                 /* fields */
  requires IDriver driver                    TLMod_TLC field_color;
  TLC color;                                 /* required ports */
  void setColor(TLC color) {                 void* port_driver;
    color = color;
  }
}                                        };
```

When debugging a `component` instance on the extension-level, we expect the debugger to provide watches for the fields, but with 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 repsective `struct` instance and map the names and their respective values.

5 Future Directions

In the future, we will investigate support for multi-staged transformations and to what extent multiple alternative transformations for a single language construct require changes to the framework architecture.

References

[1] Eclipse Foundation. Eclipse CDT (C/C++ Development Tooling), http://www.eclipse.org/cdt/, 2012.

[2] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an Extensible C-based Programming Language and IDE for Embedded Systems. In *SPLASH 2012*.

[3] M. Voelter and E. Visser. Language extension and composition with language workbenches. In *SPLASH/OOPSLA 2010*.