Multi-Level Debugging for Extensible Languages

Domenik Pavletic[1] and Syed Aoun Raza[2]

[1]itemis AG, pavletic@itemis.de
[2]itemis AG, raza@itemis.de

Abstract

Multi-level debugging of extensible languages requires lifting program state to the extension level while translating stepping commands to the base-level. Implementing such bi-directional mappings is feasible for languages with a low abstraction level (e. g., C). However, language workbenches support language stacking with a bottom-up approach from low- to high-level (e. g., domain-specific) languages. This way, generation of code written with these high-level languages is incremental. However, languages can have more than one generator, which is selected depending on the execution environment. On the other hand, provision of such flexibility makes multi-level debugging much harder. In this paper, we present an approach on how to enable debugging for such multi-staged generation environments. The approach is illustrated by mbeddr, which is an extensible C language.

1 Introduction

In domain-specific and model-driven software development several different abstractions come into play, where each involved entity might not want to deviate from their abstraction level. Mixed-language environments or environments with language extensibility fulfill this requirement i.e., each entity can use notations from its respective abstraction level and is therefore not forced to implement everything with a low-level language.

We discussed the significance of extensible debuggers for extensible languages in [1] with an implementation based on mbeddr [1]. Further, we described the requirements and the architecture of the debugger: it is designed for extensibility and supports debugging of mixed-language programs. However, in mbeddr users can build multiple levels of language extensions. As shown in Figure 1, programs written with these extensions are incrementally generated to some base language (e. g., C).

Currently, the mbeddr debugger supports single-level debugging of programs written with language extensions. Furthermore, debugger extensions are always implemented relative to the extension- and base-level. However, if at any level a generator for language extension is changed, then the corresponding debugger extension must be changed as well. Changes to generators happen on a frequent basis due to bug fixes or implementation of additional requirements.

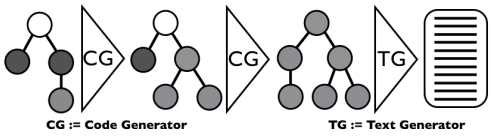

Figure 1: Incremental generation from extension- to base-level

This paper contributes an approach on how to enable multi-level debugging and reducing the effort to support generator changes. We illustrate the approach with examples based on mbeddr.

2 Requirements

The flexibility to switch between generators is an important feature of language workbenches which allow language extension. Accordingly, they must also support debuggers to provide debugging functionality for such language extensions. This introduces further requirements in addition to those defined in [1]. After analysing the application scenario, we have come to the following further requirements:

GR1 Multi-Level Debugging: Because of the semantical gap, some errors cannot be analyzed on the extension-level. Debugging the generated code is possible, however, this involves more effort because of the missing semantical richness. Hence, debugging support on different extension-levels is essential.

GR2 Seamless Integration Support: Languages can have different generators. The debugger must provide capabilities for integrating corresponding debugger extensions.

GR3 Scalability: An arbitrary number of generators can be involved during code generation. This can slow down debugging experience, however, there should not be a restriction on how many code generators can be involved.

[1]mbeddr is an extensible language [2], build with the Meta Programming System (MPS).

3 Implementation Proposal

To support multi-level debugging of extensible languages, we propose an incremental approach based on the work described in [1]. In this approach, we propose lifting program state bottom-up, whereas stepping commands are translated top-down. Figure 2 illustrates the approach: the white box represents our initial mixed-languages program, which is stepwise translated by different generators to intermediate programs (grey boxes) until it finally results in a pure base language program (black box). This representation is in contrast to our initial approach [1], which required significant re-implementation in debugger extensions if any of the generators is replaced. Because debugger extensions are always implemented in correspondence to the extension- and base-level.

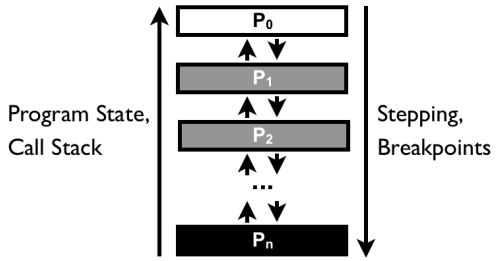

Figure 2: Bi-directional flow of debugging and generation information

Each generation step will have a corresponding debugger extension, which provides program state and propagates stepping commands with necessary information to lower debugger extensions. This way, the approach facilitates multi-level debugging (GR1). The framework will provide APIs for easily plugging in new debugger extensions (GR2). In order to construct program state or translate stepping commands, the approach will have to traverse all related debugger extensions. This way, number of extensions will only be limited by the number of generators involved during transformation (GR3).

4 Discussion

In below listings we show a multi-level transformation of a language extension that sums up a range of numbers. This extension is stepwise translated to C. Listing 1 contains the code of the high-level language extension for summarizing numbers from 0 to 10.

```
1 void main() {
2   int sum = 0;
3   sum = [0 to 10];
4 }
```

Listing 1: First Level

Listing 2 shows the generated code after the first transformation to a *loop* language extension.

```
1 void main() {
2   int sum = 0;
3   loop [0 to 10] { sum += it; }
4 }
```

Listing 2: Second Level

The listing 3 shows the complete unrolled form of `loop` after the final transformation step as a pure C program (the base language).

```
1 void main() {
2   int sum = 0;
3   sum += 0;
4   ...
5   sum +=10;
6 }
```

Listing 3: Base Level

For supporting multi-level debugging for the above described scenario a debugger extension is required for each language. In this example, if a user wants to *step over* the `sum statement` in listing 1, it involves the following steps: first, `sum statement` debugger extension sets a breakpoint on the `loop` in listing 2. Next, `loop` debugger extension sets a breakpoint on the second statement of listing 3. Finally, this information is propagated to the base-level debugger (here, `gdb`).

The previously described scenario clearly defines that it is possible to debug on different levels in a multi-level transformation. Further, this is accomplished by mapping debug information stepwise. The combination of such debugger extensions will provide multi-level debugging from highest to the base language (GR1). Additionally, it is possible to scale this approach by combining an arbitrary amount of transformation levels (GR3).

Generating a different structure requires introducing a new generator, but also a new debugger extension for this generator (GR2). Nevertheless, changes to existing generators only require re-implementation in the respective debugger extension.

5 Conclusion

This paper discusses the requirements and an implementation strategy for supporting multi-level debugging for extensible languages in mbeddr. Depending on the language workbench there can be additional requirements. However, we have discussed the general requirements which are necessary to implement basic functionality of multi-level debugging.

6 Future Directions

In the future we will investigate how debugger extensions can be implemented inside generators. Also we will analyze how much information (e. g., variable names) can be reused this way. Finally, we will investigate to which extent debugger extensions can be derived from transformation rules.

References

[1] D. Pavletic, S. A. Raza, M. Voelter, B. Kolb, and T. Kehrer. Extensible debuggers for extensible languages. *Softwaretechnik-Trends*, 33(2), 2013.

[2] M. Voelter. Language and IDE Development, Modularization and Composition with MPS. In *GTTSE 2011*, LNCS. Springer, 2011.