ABSTRACT
In 1997, the general lack of debugging tools was termed “the debugging scandal” [7]. Today, as new languages are emerging to support software evolution, once more debugging support is lagging.

The powerful abstractions offered by new languages are compiled away and transformed into complex synthetic structures. Current debugging tools only allow inspection in terms of this complex synthetic structure; they do not support observation of program executions in terms of the original development abstractions.

In this position paper, we outline this problem and present two emerging lines of research that ease the burden for debugger implementers and enable developers to debug in terms of development abstractions. For both approaches we identify language-independent debugger components and those that must be implemented for every new language.

One approach restores the abstractions by a tool external to the program. The other maintains the abstractions by using a dedicated execution environment, supporting the relevant abstractions. Both approaches have the potential of improving debugging support for new languages. We discuss the advantages and disadvantages of both approaches, outline a combination thereof and also discuss open challenges.

Categories and Subject Descriptors
D.2.5 [Testing and Debugging]: Debugging aids; D.3.2 [Language Classifications]: Very high-level languages

General Terms
Languages

Keywords
Language-independent debugging, next generation languages, multiple abstraction debugging

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

1. INTRODUCTION

New languages with advanced support for software evolution start being used in practice. Such languages, like domain-specific languages (DSL), AspectJ, Scala, Compose*, or JPred, improve evolvability by offering advanced modularization techniques, which also increases the comprehensibility of code.

However, while they aim at improving the maintenance task, especially in the maintenance phase, dedicated tool support for these languages is missing. Almost 25% of maintenance is carried out for repairing faults [8], which includes locating defects in the software code, often called debugging. Thus, appropriate debugging support is important in the life cycle of software. In this position paper, we discuss what problems for debugging the new languages bring and how we aim to solve them.

The typical approach of implementing new programming languages, especially domain-specific ones, is to transform the source code to code of an established language, the so-called host language. While this facilitates the use of tools like debuggers that already exist for the host platform, developers are stuck with debugging their code at the level of abstractions of the host language: They see a complex, synthetic composition of low-level abstractions [10].

After Henry Lieberman proclaimed the “Debugging Scandal” [7] in 1997, we are facing the next generation of the “Debugging Scandal” today: To access the program state in terms of abstractions of the actually used language, the developer has to invasively modify the code, to make it output all relevant state. Otherwise, the developer has to manually map the low-level abstractions presented by the host language’s debugger to the source language. This requires intimate knowledge of the compilation strategy that transforms the source code to a synthetic host language program.

![Figure 1: Transformation of different abstractions in the software life cycle](image-url)
What is more, traditional approaches only facilitate observing a program execution in terms of the source programming language’s abstractions—and, as we have seen, for new, emerging languages and DSLs, this is not even the case. But the code level is already a low-level abstraction compared with other code representations created during the software development process like architectural or detailed design (Figure 1). While these high-level code views may not be the best abstraction for traditional debugging, observing the program execution at this level is nevertheless useful for tasks like profiling, testing, test coverage analysis or verification. When inspecting the run-time state, each stakeholder should be presented with information in terms of the abstractions he is most familiar with [3].

What we claim in this position paper is that with a new generation of programming languages and with increasing tool support for early development phases, the “Debugging Scandal” rekindles. New tools and techniques are required that allow observing and interacting with program executions in terms of abstractions natural to the developer. These abstractions must correspond to the language the developer used to define program elements, be it using an architectural or detailed design language or a programming language.

We outline two approaches to alleviate the next-generation “Debugging Scandal” which we deem applicable in different situations.

A first approach involves keeping track of the transformations performed during compilation. This trace of the compilation, the abstraction mapping, can be used to negotiate between a debugger front-end dedicated to the source abstractions and a back-end provided by the host platform. With this approach, the developer can keep thinking in terms of the abstractions he/she is used to and the host platform can be re-used without modifications. Only the compiler has to be enhanced to create an appropriate mapping. Multiple levels of such mappings can be created during a transition from one development phase to the next. The advantage of this approach is that it can be easily applied to languages with an existing compilation tool chain. The downside is that only those program elements can be observed which correspond to program elements supported by the host language debugger.

An alternate approach is to introduce a dedicated intermediate language to which a next generation language is compiled and which keeps the source-level abstractions explicit. In this approach, the source code and the intermediate representation of a program have the same structure, thus debugging in terms of the source-level abstractions is possible. As the structure of the detailed or architectural design language is in general different form that of the source language, this approach cannot support multiple levels of abstractions. Nevertheless, it can offer a richer set of features for a single abstraction, including run-time modification of source code. This approach requires using a dedicated compiler and execution platform, but enables full access to all relevant language elements during debugging.

As we have observed, many new languages share common core concepts. Thus it is possible to define a common intermediate language which embodies these concepts. Then the execution platform, including the debugger, only has to be implemented once and is shared between all supported languages.

Combining both approaches, it is also possible to create external mappings from, e.g., design-level abstractions to source code, and to compile the program code to a dedicated intermediate language keeping the source-language abstractions. Such an approach may even improve the observation of program executions in terms of higher-level views, as next generation languages or DSLs often aim to keep the program code’s structure closer to the design. Thus, advanced debugger features may also benefit the power of mappings created for early design artifacts.

In the following two sections, we will first elaborate on the two approaches for enabling debugging in terms of advanced and higher-level abstractions. In Section 4 we will conclude our position statement and outline the potential of improving debugger support by combining both approaches and the remaining challenges. For a preliminary discussion of work related to the two presented approaches as well as early prototype implementations, we refer to our previous publications [5, 11], for brevity.

2. ABSTRACTION TRANSFORMATION MAPPINGS

Abstraction mappings create a view of the run-time as if the program was executed in terms of the source code abstraction, even though these abstractions were lost during compilation. To create this abstract view of the run-time, the transformation performed by the compiler is reverted.

More precisely, the relation between the source language and the host language is modeled by a model-to-model transformation [4]. For each program that is compiled, the compiler emits sufficient information to make this model-to-model transformation invertible.

This principle is depicted in Figure 2. A program, written in a high-level language, is compiled to a host language. The resulting program is executed in a run-time environment that can be debugged by a host-language debugger. The information presented by the host-language debugger is in terms of the host-language abstractions and not in terms of the original language abstractions. The high-level language abstractions are restored by the high-level debugger, which implements the inverse model-to-model transformation.

As input, it takes the host-level debug information and the extra debug information emitted by the compiler.

Figure 2: Principle of abstraction transformation mappings

This design is not new, as it is already used by most existing debuggers. A debugger like GDB for example consumes information from a lower-level infrastructure (the operating system and CPU) and combines it with external debugging
information to restore the abstractions of higher-level languages like C, Fortran or C++. Even though this design is common, it is generally not well understood. No body of literature or design guidelines exist. Therefore it is not often applied to languages with a complex compilation process.

The advantage of this approach is that it puts no constraints on the host language or compilation. The mapping and related infrastructure are completely separate from the system to be debugged, which has important consequences:

1. There are no constraints on the design of the language to which this approach should be applied. The approach can be applied to any language, even if it already exists.

2. When not debugging, there is no overhead. No part of the infrastructure is even present. This allows debugging on resource constrained devices.

3. It is possible to use multiple views on the same system. Views can be stacked to create higher-level views. This may for example enable architecture level representations, where the interaction between architectural components can be examined. Views can also be combined: If multiple source languages are present, each can be presented in terms of its own abstractions.

4. The approach is not limited to explicit abstractions. It can also create abstractions that were not explicit in the original program. For example: if a developer consistently uses an object-oriented programming style in a procedural language, an abstraction mapping can be created that presents this procedural program in terms of object-oriented abstractions, even though these abstractions were not explicit in the source.

The disadvantage of this approach is its complexity. It is not easy to build this type of debugger. An accurate model of the source language, the host language and the compilation is required. To design these, the debugger designer, must have intimate knowledge of the language’s structure. While this is preferable to the current situation where all developers must have this knowledge, it requires significant effort and experience. Also application of the model-to-model transformation is technically complex.

This complexity is however not insurmountable. Further research can make abstraction mappings more applicable. In practice, there is a lack of design guidelines, patterns, tools and reference materials. Existing research [5] and emerging tools [9] already move in this direction. In a theoretical sense, no framework exists to define the limits of this approach and to support the analysis of the compilation structure. In the future we plan to focus on a more disciplined approach to debugger construction. One that is explicitly based on model-to-model transformation, enabling more automation in the process of creating a debugger and allowing more systematic reasoning. It will reduce the time required to build debuggers and allow more rapid experiments.

3. DEDICATED INTERMEDIATE LANGUAGE

Instead of compiling a new language to an unsuitable host intermediate representation and restoring abstractions by means of a mapping, high-level source code can be compiled to a dedicated intermediate language (IL) that has the same modularity concepts and expressiveness as the high-level language. We observed that transformation from one model of language abstractions to another, like the host intermediate language, is difficult when inconsistency exists between the two abstraction sets. The reason is that an abstraction in one set cannot be fully expressed by either a single abstraction or a cohesive group of several abstractions from the other set. As an example, consider the language AspectJ, which is compiled to Java bytecode; AspectJ pointcuts—

independent syntactic elements in the source language—are partially evaluated by the compiler and, thus, do not have a bytecode counterpart in full [6].

When a dedicated intermediate language is designed for a new source language, also a dedicated run-time is required to perform the execution in the way expected by the language designer and at the granularity defined by the IL. The principle of this approach is depicted in the Figure 3. The compiler takes source programs as input and generates an intermediate representation (IR) of the programs in terms of the IL. The generated IR is sent to a dedicated run-time which conforms with the IL. Debug information stored in the IR is accessible to the high-level debugger, which can present the program state in terms of the high-level language abstractions.

The next generation languages introduce new syntactic abstractions with new models of computation and new kinds of events. Thus, debugger users want to be able to refer to the computation in terms of the higher-level abstractions and new event kinds, e.g., by setting a breakpoint to the computation of such an abstraction, or by stepping over the whole computation. A debugger is not fully capable of handling this if the higher-level computations are not compiled to host-language computations observable by the host debugger or if the boundaries of the host-level computations do not correspond to the boundaries of the higher-level computation. Frequently, compilers of next generation languages perform partial evaluation and optimization of computations. Computations and state that are optimized away can, thus, not be observed.

The advantages of the approach of keeping all source-level abstractions in a dedicated IR mainly come from the following aspects:

1. For debugger designers, the mapping between the source code and the IR becomes very simple since every abstraction in the source language has a counterpart in the IR.

2. The dedicated run-time facilitates debuggers with abilities, such as modification of behavior of the executing
program at the granularity of the source-level abstractions.

3. Additional kinds of events which are specific to the high-level language can be explicitly observed, e.g., by means of breakpoints.

Since several parts of a language implementation depend on the intermediate language, this approach requires the implementation of a dedicated tool chain: A compiler mapping the high-level language to the dedicated intermediate language; a run-time executing the resulting intermediate representation of a program; and a debugger communicating with that run-time. But for many next-generation languages, the tool chain does not have to be implemented from scratch: Because these languages extend established languages, large parts are mapped straightforward to the host language such that the host compiler, run-time and debugger can be reused for these parts.

From our past experience, we observed that some languages can be grouped into families sharing core concepts [2]. For such a family it is possible to design a common IL which contains the superset of abstractions of all the family’s languages. For example, many languages offer means to control late binding of functionality to, e.g., method calls, by means of expressions over the program state; the languages differ in, e.g., the concrete syntax, the expressiveness of the expressions and verifications performed by the compiler. Implementing a run-time and debugger for such a common IL supports debugging in terms of the source language and allows reusing the tools for multiple languages at the same time [11, 1].

In order to fit all languages, the IL must be more powerful and fine-grained than each individual language, and the terminology used in the IL cannot always correspond to the flavor of the individual language. Thus, programmers see abstractions with a structure comparable to the source code, but they may be presented with a different terminology or granularity. This may still be confusing and requires getting familiar with the common IL. Nevertheless, the gap between source-level abstractions and those presented by the debugger is small and much less hindering than is the case in conventional approaches where high-level languages are compiled to an unsuitable host IL. Furthermore it is our goal to also close this remaining gap, as will be outlined in Section 4.

4. CONCLUSION

Both approaches have the potential to solve the next generation debugging scandal for languages with advanced support for software evolution. In [5, 11], we discuss our experience in designing and implementing debuggers by following the two approaches respectively. Both approaches have different properties: The abstraction mappings can offer inspection features without changing the run-time structure; while a dedicated intermediate language additionally provides modification and simulation features but requires a custom run-time environment.

In order to increase the reusability and make both approaches more applicable, it is beneficial to identify language families and define a common IL respecting the superset of the member languages’ concepts.

In the long term, when both approaches are more mature, a combination would lead to the richest results. For a language family, a shared execution environment with a shared debugger and shared visualization of the program execution can be implemented. An extra mapping can restore the language specific terminology and the mental model. Additional mappings can support higher-level abstractions like design languages. Thus finally, to overcome the debugging scandal, we put forward the goals to:

1. support debugging in terms of the abstractions of the programming or design language actually used by the developer,
2. support interacting with program execution in terms of high-level abstractions,
3. maximize the efforts that can be reused across multiple different languages (which are sufficiently similar), and
4. simplify the efforts that are still required to make debugging or program observation respect the peculiarities of the actually used language.

5. ACKNOWLEDGMENTS

This work is partly funded by a CSC Scholarship (No.2008613009).

6. REFERENCES