Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

An Integrated Approach to Source Level Debugging and Compile Error Reporting in Metaprograms

Yannis Lilis[a] Anthony Savidis[ab]

a. Institute of Computer Science, FORTH

b. Department of Computer Science, University of Crete

Abstract Metaprogramming is an advanced language feature enabling to mix programs with definitions that may be executed either at compile-time or at runtime to generate source code to be put in their place. Such definitions are called metaprograms and their actual evaluation constitutes a compilation stage. As metaprograms are also programs, programmers should be supported in handling compile-time and runtime errors, something introducing challenges to the entire tool chain along two lines. Firstly, the source point of a compile error may well be the outcome of a series of compilation stages, thus never appearing within the original program. Effectively, the latter requires a compiler to track down the error chain across all involved stages so as to provide a meaningful, descriptive and precise error report. Secondly, every compilation stage is instantiated by the execution of the respective staged program. Thus, typical full-fledged source-level debugging for any particular stage should be facilitated during the compilation process. Existing implementations suffer in both terms, overall providing poor error messages, while lacking the required support for debugging metaprograms of any staging depth. In this paper, we outline the implementation of a compile-time metaprogramming system offering all aforementioned facilities. Then, we detail the required amendments to the compilation process and the necessary interaction between the compiler and the tool-chain (IDE). Finally, we discuss how similar functionality could be achieved in systems offering runtime metaprogramming.

Keywords Metaprograms; compile-time metaprogramming; run-time meta-programming; staged languages; source-level debugging; error messages.

1 Introduction

The term metaprogramming is generally used to denote programs that generate other programs and was originally related to the existence of a macro system like the *C*

Preprocessor (*CPP*) [KR88] or the *Lisp* macro system [Baw99] that would allow program fragments to be built up at compile-time. Lexical systems like the *CPP* are recognized as being inadequate for metaprogramming as they operate on raw text, unaware of any context information, while most languages do not share *Lisp*'s syntactic minimalism to provide an equally powerful facility with seamless integration. In modern languages, metaprogramming is closely coupled with functions that operate on some abstract syntactic form, like an abstract syntax tree (AST), and can be invoked during compile-time to change existing code or produce and insert additional code in the source being compiled. Such functions are called meta-functions and they as a whole constitute the metaprogram. The compilation of a program that contains a metaprogram requires it to be executed at compile-time to produce a possibly changed source file. If the resulting source contains additional metaprograms they are executed in the same way until we reach a final source with no metaprograms that will be compiled into the final executable. This iterative process may involve multiple steps of metaprogram evaluations called *compilation stages*. Languages that support such a compilation scheme are called *multi-stage languages* [SBM00, TS00] with *MetaML* [She98] and *MetaOCaml* [CLT$^+$01] being two typical examples. Multi-stage programs are essentially programs whose source code is finalized through a sequence of evaluations defined in the program itself. They use special annotations to specify the order of their computations, with respect to the compilation stage they appear in. These annotations are called *staging annotations*. Staging annotations however are not limited to multi-stage languages. For example, *C++* [Str00] is a two-stage language where the first stage is the interpretation of the templates (denoted by the <> tags) and the second stage is the compilation of the non-template code.

Metaprogramming can help achieve various benefits [She01], the most typical of which is performance. It provides a mechanism for writing general purpose programs without suffering any overhead due to generality; rather than writing a generic but inefficient program, one writes a program generator that generates an efficient solution from a specification. Additionally, by using partial evaluation it is possible to identify and perform many computations at compile time based on a-priori information about some of the program's input, thus minimizing the runtime overhead. Another application is the reasoning about object-programs that allows analyzing its properties for performance improvement or program validation. Finally, metaprogramming can achieve code reusability at a macroscopic scale by implementing parameterized design patterns and instantiating them based on the given parameters.

Context As with normal programs, when writing metaprograms errors are bound to happen, so having the proper tools to understand the origin of an error is important. In normal programs, there are two main error categories: compilation and execution errors. Compilation errors are easier to resolve as compilers can identify exactly where something went wrong and why. On the other hand, execution errors involve runtime state that may be different between executions and is not directly visible to the programmer, making them harder to resolve. Fortunately, debuggers can provide the required information by allowing inspection of runtime values and call stack, tracing the program execution and adding breakpoints, thus facilitating the error resolution.

Problem The same principles regarding errors and their management apply for metaprograms as well. However, both metaprogram compilation and execution may involve code that was never part of the original program. This means that compilation errors are not that easy to deal with anymore as the error reported no longer reflects code that the programmer can see and understand. Moreover, metaprogram execution errors are even harder to face since there is no actual source that can be used for debugging. It becomes obvious that error handling in metaprograms requires more sophisticated tools, without which the programmer's ability to write, understand and maintain metaprograms may be severely hindered. Clearly, compilation errors due to staging should encompass sufficient information to identify their cause while stage execution errors should be detectable using typical source-level debugging.

Contributions In this paper, we discuss the implementation details of a compile-time metaprogramming system addressing the previous issues based on:

- Generating source files for compilation stages and their outputs (original source transformations) and incorporating them into the IDE project manager, associated with the main source being built. These files are integrated in the workspace and can be used for code review, error reporting and source-level debugging.
- Recording the chain of source locations involved in the generation of erroneous code fragments for precise error reporting purposes. This chain includes the original source with the generated stage sources and their respective outputs, and can be easily navigated within the IDE to identify the root of the error.
- Mapping breakpoints in the original source to breakpoints of generated stage sources and enabling compile-time source-level debugging directly on stages.

We also discuss how a similar approach can be used to provide error-reporting and metaprogram debugging functionality for languages with runtime metaprogramming.

2 Related Work

Our work targets the field of metaprogramming and focuses on delivering an integrated system able to support debugging of metaprograms as well as providing precise and meaningful messages for compilation errors originating within meta-code. In this context, the topics relevant to our work are compile-time debugging and error reporting.

We do not study preprocessor systems that operate on raw text as they lack essential support for metaprogramming since no iteration, query, manipulation and creation on individual source fragments is possible. This is due to the fact that preprocessors operate merely on text blocks rather than on syntactic elements. For instance, the *CPP* does not support iteration and macros that generate other macros neither it allows programmers perform conditional preprocessing according to the syntactic type of the manipulated text. From those only iteration can be addressed using custom add-ons like the *Boost.Preprocessor* [AG04] library, although iteration is a standard feature in other preprocessors like *M4* [Tur94]. In any case, all preprocessors are ignorant of the language syntax and semantics and their application is separated from the main language as a distinct preceding stage. In fact, they are not limited to a specific language, the same being true even for *CPP* which originally gained its name from the *C* language. As such, we consider that any facilities for error-handling, and possibly debugging, should be separately incorporated as part of the tools themselves.

2.1 Compile-Time Debugging of Stages

C++ [Str00] support for metaprogramming is based on its template system that is essentially a functional language interpreted at compile time [Vel96, AG04]. There are *C++* debuggers (e.g. *Microsoft Visual Studio Debugger*, *GDB*) that allow source level debugging of templates, but only in the sense of tracing the execution of the template instantiation code and matching it to the template definition sources. However, there is no way to debug the template interpretation during compilation. A step towards this end is *Templight* [PMS06], a debugging framework that uses code instrumentation to produce warning messages during compilation and provide a trace of the template instantiation. Nevertheless, it is an external debugging framework not integrated into any development environment and relies on the compiler generating enough information when it meets the instrumented code. Finally, there is no programmer intervention; the system provides tracing but not interactive debugging.

The *C++11* standard [Bec11] adds an extra metaprogramming approach through const expressions; the keyword `constexpr` can be used on functions that meet some requirements, allowing them to be invoked during compilation if their arguments are constants. However, there is no support for debugging their execution at compile time.

D [Ale10] is a statically typed multi-paradigm language that supports metaprogramming by combining templates, compile time function execution, and string mixins. *Descent* [des], an *Eclipse* plugin for *D* code, provides a limited compile-time debugging facility for simple templates and compile-time functions. However, the debugging process does not involve the normal execution engine of the language; instead it relies on a custom language interpreter for both execution and debugging functionality.

Nemerle [SMO04] is a statically typed object oriented language that supports metaprogramming through its macro system. *Nemerle* and its IDE, *Nemerle Studio*, support debugging macro invocations during compile time. *Nemerle* macros are compiler plug-ins that have to be implemented in separate files and modules and are loaded during the compilation of other files that invoke them. Since they are dynamically linked libraries with executable code, they can be debugged by debugging the compiler itself; when a macro is invoked, the code corresponding to its body is executed and can be typically debugged. However, the development model is restrictive, requiring macros to be separated, and the debugging process is rather cumbersome.

There are a lot more compiled languages, both functional and imperative, that support metaprogramming. Some examples include *MetaOCaml*, *Template Haskell* [SJ02], *Dylan* [BP99], *Metalua* [Fle07] and *Converge* [Tra05]. However, none of them provide any support for debugging metaprograms during compilation.

2.2 Compile-Error Reporting

Most compiled meta-languages provide very limited error reporting for compilation errors originating from generated code. Typically, an error is reported directly at the generated code with no information about its origin or the context of its occurrence. Below we examine the few cases that offer more sophisticated error reporting.

C++ compilers (e.g. *Microsoft Visual Studio*, *G++*) provide fairly descriptive messages regarding compilation errors occurring within templates. Using these messages, a programmer may follow the instantiation chain that begins with the code that caused the error (typically user code) and ends with the code that actually triggered the error (probably library code). Essentially, these error messages represent the execution stack of the template interpreter. While potentially informative and able to provide accurate information to experienced programmers, template error messages are quite cryptic for average programmers and require significant effort to locate the actual error. Unfortunately, this is the common case for nontrivial metaprograms and applies to libraries with multiple template instantiations (e.g. *Boost* [AG04]).

Converge [Tra05] provides some error reporting facilities related to metaprogramming by keeping the original source, line and column information for quoted-code. It allows associating a given virtual machine instruction with multiple source code locations [Tra08] so it is possible to combine such information from quasi-quotes and associated insertions to provide a detailed message that allows tracking down the origin of an error. However, it fails to maintain such information across splice locations that involve staging execution. Thus, any error originating in generated code cannot be properly traced back to its origin. Finally, errors are reported only in the original source, with no context about the temporary module executed to perform the splice.

3 Language

Our work has been carried out in the *Delta* programming language [Sav05, Sav10], which is imperative, untyped, object-based, compiled to byte code and run by a virtual machine, belonging to a language family relative to *Self*, *Lua* and *Javascript*. To support metaprogramming facilities, several extensions were made to the language, its compiler and its *Sparrow* IDE [SBG07]. Having access to language and IDE source code to implement these extensions was a significant factor in choosing Delta over existing meta-languages. However, our proposition can be directly applied in more mainstream compiled meta-languages like *Converge* and *MetaLua* as well as languages with runtime metaprogramming like *MetaML*, *Metaphor* [NR04] or *Mint* [WRI$^+$10].

3.1 Staging Constructs

To support multi-stage metaprogramming, *Delta* has been extended with staging annotations similar to the ones of *MetaOCaml*.

- ***Quasi-quotes*** (written `<<...>>`) can be inserted around most language elements to enclose their syntactic form into an AST, creating a value that contains code.
- ***Escape*** (written `~(expr)`) can be used on an expression within quasi-quotes to escape the syntactic form and interpret the expression normally. It allows combining existing AST values in the AST being constructed by the quasi-quotes.
- ***Inline*** (written `!(expr)`) can be used on an expression to evaluate it at translation time and insert its result (an AST or AST convertible value) directly into the program code by substituting itself. It is used for program transformation.
- ***Execute*** (written `&stmt`) can be used to execute a statement at translation time. An addition to the *MetaOCaml* annotations, execute performs the computation without modifying the main AST. It is used for non-expression computations (e.g. loops) but also to specify code available only during compilation.

The following is a simple example (adopted from [COST03]) showing the staging constructs of Delta. `ExpandPower` creates the AST of its `x` argument multiplied by itself `n` times, while `MakePower` creates a specialized power function AST.

```
&function ExpandPower (n, x) {
    if (n == 0) return <<1>>;
    else return <<~x * ~(ExpandPower(n - 1, x))>>;
}
```

```
&function MakePower (n)
   { return <<(function(x) { return ~(ExpandPower(n, <<x>>)); })>>; }
power3=!(MakePower(3)); //power3=(function(x){return x * x * x * 1;});
```

3.2 Compilation Process

The *quasi-quotes* and *escape* annotations are used to create and combine code segments and involve no staging computation on their own. Staging occurs due to the existence of the *inline* and *execute* annotations, which are therefore also referred to as *staging tags*. Essentially this means that any program containing these staging tags cannot be compiled until all of them are translated first. However, staging tags can be nested or their evaluation may introduce additional staging tags, so the whole compilation process requires multiple translation and execution stages (Figure 1). To support this scheme, we extend the compilation process as follows. Initially, we parse the original source program to produce the main AST. If it contains staging tags, we traverse it to collect the nodes for the next compilation stage based on the two following properties: (i) nested staging tags should always be evaluated at an earlier stage than outer ones; and (ii) staging tags of the same nesting level should be evaluated in the same stage. Thus, the staging tags for a given compilation stage are the innermost. After collecting the appropriate nodes, we assemble them to create the compilation stage AST. By construction, this contains no staging tags so it can be normally compiled and executed. During the execution, the original source *inline* tags that were translated to virtual machine instructions will modify the source by inserting code into the main AST. When the execution completes, the main AST is fully updated and ready for the next stage. This process continues iteratively until no more staging tags exist in the main AST (i.e. it is the final AST), when it is normally compiled to the final binary.

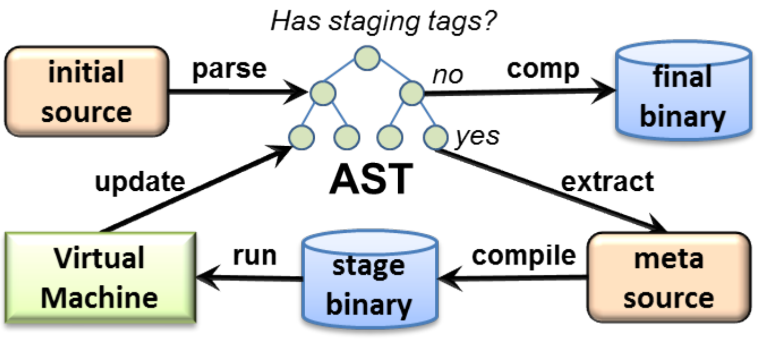


Figure 1 – High level overview of a multi-stage compilation process.

3.3 IDE Features

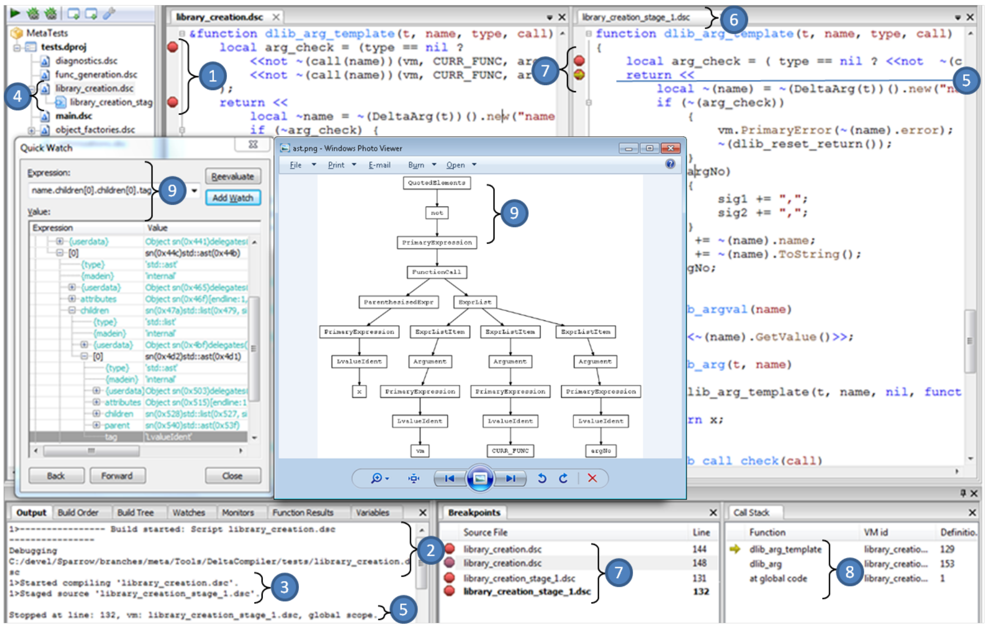
Metaprogramming especially for multiple stages is a quite demanding task, so our system aims to facilitate the development of metaprograms as much as possible.

Any source code transformations performed by the compilation stages are handled internally by the compiler and are therefore transparent to programmers so special attention is given to providing meaningful, descriptive and precise error reports in case some compilation stage raises a compilation error. Such reports provide the full error chain across all stages involved in the generation of the erroneous code. Within

Sparrow, programmers may easily navigate back and forth across this error chain. This feature is a significant aid in tracking the error's origin and ultimately resolving it.

Additionally, every compilation stage involves staged program execution that should be subject to typical source-level debugging. *Sparrow* offers such functionality supporting typical debugging facilities like expression evaluation, watches, call stack, breakpoints and tracing. In particular, a compile-time debug session (Figure 2) involves the following actions: The programmer initially sets breakpoints within a meta-function in the original source file (Figure 2:1) and then builds it with debugging enabled. This launches the compiler for the build and attaches the debugger to it for any staged program execution (Figure 2:2). During compilation, the IDE is notified about any compilation stage sources (Figure 2:3) that are added in the workspace associated with the main source being built (Figure 2:4). When a breakpoint is hit, execution is stopped at its location (Figure 2:5) and the source corresponding to the breakpoint hit is opened within the editor to allow further debugging operations such as tracing, variable inspection, etc. (Figure 2:6). Notice that the breakpoints in the generated stage source (including the one hit) were automatically generated based on the breakpoints set in the original source file (Figure 2:7). While execution is broken, it is possible to navigate across active meta-function calls through the call stack (Figure 2:8) or inspect variables containing code segments as AST values (Figure 2:9).

The compilation stage sources as well as their output (main AST transformation stages) are actually created and inserted into the workspace even when performing a non-debugged build. This allows programmers to review the assembled and the generated code of each stage along with the effect it has on the final program even after the build is completed, thus allowing for a better understanding of their code. Figure 3 highlights this functionality showing all sources related to the power example.



```
&function dlib_arg_template(t, name, type, call)
    local arg_check = (type == nil ?
        <<not ~(call(name))(vm, CURR_FUNC, ar
        <<not ~(call(name))(vm, CURR_FUNC, ar
    );
    return <<
        local ~name = ~(DeltaArg(t))().new("name
        if (~arg_check) {
```

```
function dlib_arg_template(t, name, type, call)
{
    local arg_check = ( type == nil ? <<not  ~(c
    return <<
        local ~(name) = ~(DeltaArg(t))().new("n
        if (~(arg_check))
            {
                vm.PrimaryError(~(name).error);
                ~(dlib_reset_return());
            }
argNo)
            {
                sig1 += ",";
                sig2 += ",";
            }
 += ~(name).name;
 += ~(name).ToString();
gNo;

b_argval(name)

<~(name).GetValue()>>;

b_arg(t, name)

lib_arg_template(t, name, nil, funct

rn x;

b call check(call)
```

Expression	Value
{userdata}	Object sn(0x441)delegates(
[0]	sn(0x44c)std::ast(0x44b)
{type}	'std::ast'
{madein}	'internal'
{userdata}	Object sn(0x465)delegates(
attributes	Object sn(0x46f)[endline:1,
children	sn(0x47a)std::list(0x479, si
{type}	'std::list'
{madein}	'internal'
{userdata}	Object sn(0x4bf)delegates(
[0]	sn(0x4d2)std::ast(0x4d1)
{type}	'std::ast'
{madein}	'internal'
{userdata}	Object sn(0x503)delegates(
attributes	Object sn(0x515)[endline:1
children	sn(0x528)std::list(0x527, si
parent	sn(0x540)std::ast(0x53f)
tag	'LvalueIdent'

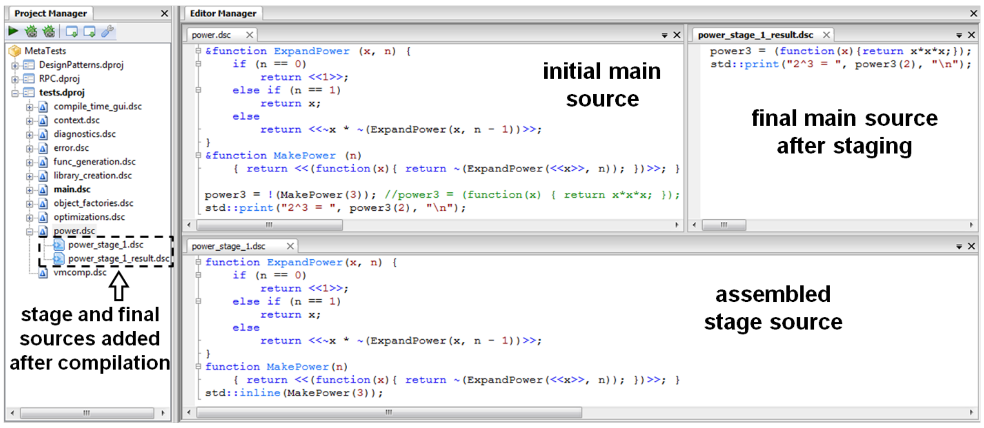
```
1>---------------- Build started: Script library_creation.dsc
----------------
Debugging
C:/devel/Sparrow/branches/meta/Tools/DeltaCompiler/tests/library_creation.d
sc
1>Started compiling 'library_creation.dsc'.
1>Staged source 'library_creation_stage_1.dsc'.

Stopped at line: 132, vm: library_creation_stage_1.dsc, global scope.
```

Source File	Line
library_creation.dsc	144
library_creation.dsc	148
library_creation_stage_1.dsc	131
library_creation_stage_1.dsc	**132**

Function	VM id	Definitio.
dlib_arg_template	library_creatio...	129
dlib_arg	library_creatio...	153
at global code	library_creatio...	1

Figure 2 – A compile-time debugging session in Sparrow; items 1-9 are discussed in text.



power.dsc

```
&function ExpandPower (x, n) {
    if (n == 0)
        return <<1>>;
    else if (n == 1)
        return x;
    else
        return <<~x * ~(ExpandPower(x, n - 1))>>;
}
&function MakePower (n)
    { return <<(function(x){ return ~(ExpandPower(<<x>>, n)); })>>; }

power3 = !(MakePower(3)); //power3 = (function(x) { return x*x*x; });
std::print("2^3 = ", power3(2), "\n");
```

power_stage_1_result.dsc

```
power3 = (function(x){return x*x*x;});
std::print("2^3 = ", power3(2), "\n");
```

power_stage_1.dsc

```
function ExpandPower(x, n) {
    if (n == 0)
        return <<1>>;
    else if (n == 1)
        return x;
    else
        return <<~x * ~(ExpandPower(x, n - 1))>>;
}
function MakePower(n)
    { return <<(function(x){ return ~(ExpandPower(<<x>>, n)); })>>; }
std::inline(MakePower(3));
```

Figure 3 – Compilation sources in Sparrow: project manager (*left*), initial main source (*middle*), assembled stage source (*bottom*), final main source after staging (*right*).

4 Compile Errors

4.1 Storing the Source Code of Every Stage and its Output

The ASTs assembled for each compilation stage are temporary and only used for code generation. To support reporting compile errors for stage sources or debugging their execution, these ASTs can be further utilized to create source files containing the code they represent, a process known as *unparsing*. Since these files are meant to be shown to programmers, they have to be as readable as possible, so their code must span across multiple lines and be properly indented. Additionally, user written code should clearly be preferred over automatically unparsed code (different indentation, empty lines, comments, etc.), so any code segment originating in the initial source should maintain its original form. To support this efficiently, AST nodes contain their starting and ending character positions in the original source to retrieve their text segments. This way, the unparsing algorithm will combine original and generated text segments to produce a complete source for each compilation stage.

To obtain the source code for a specific compilation stage, we apply the unparsing algorithm on its AST and store the result using some naming convention, for example adding a suffix along with the current stage number. To allow programmers review not only the compilation stages, but also the code they generate and the modifications they perform on the main AST, we also unparse the updated main AST after the successful execution of each compilation stage. Essentially, this means that for an execution involving n compilation stages, there will be $2*n$ source files generated. The final program being compiled into executable code is actually the output of the last compilation stage, so it will also be available as the last generated source (Figure 4).

Once a stage source or stage output has been generated during compilation it is sent to the IDE and associated with the main source being built. The nature of the involved communication relies on the way the compiler is invoked by the IDE. In case the compiler is available as an IDE service invoked during the build process, it is possible to directly provide callbacks for events like compilation errors or generation of stages source. If it is implemented as a separate executable spawned by the IDE, the communication channel between them is typically a memory pipe using standard

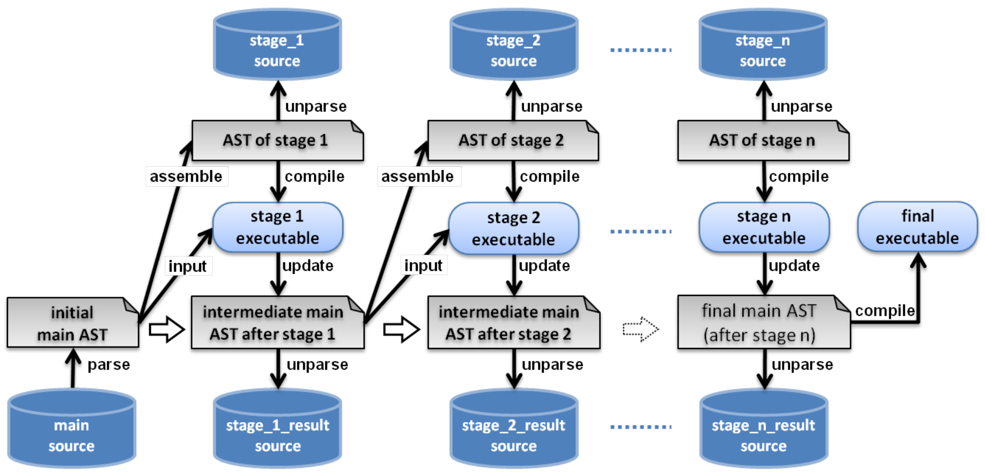


Figure 4 – Storing the source code of all compilation stages and their outputs.

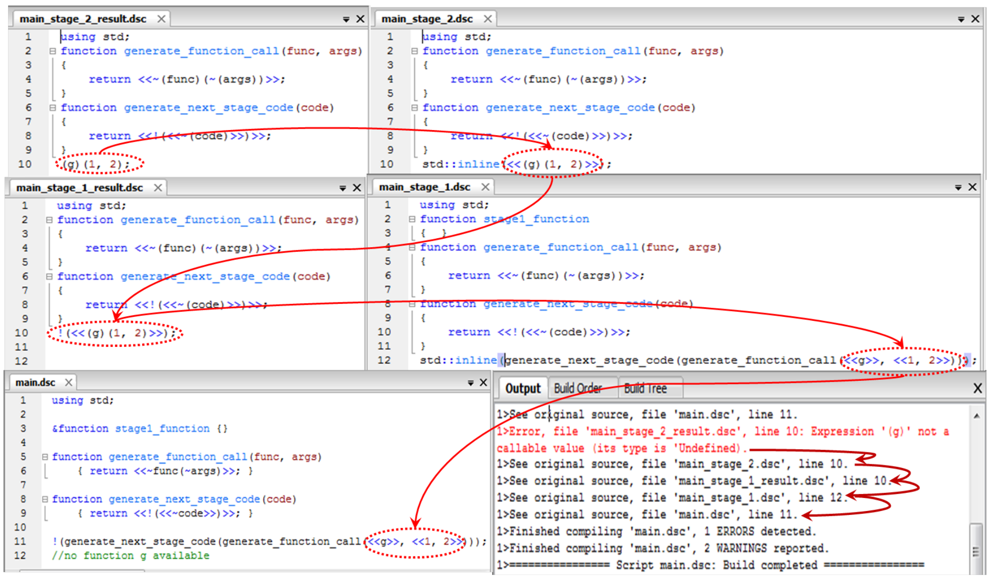
text input and output. This requires establishing a protocol for communicating the compiler events to the IDE using some text representation. For example, to notify the IDE about the existence of the stage sources, the compiler may use a special message containing resource identifiers (e.g. file paths). Finally, there is the option of network based communication. The IDE acts as a server for receiving stage sources and supplies the host and port information as compiler invocation parameters. Once launched, the compiler will use these parameters to establish a connection with the IDE and use it for supplying it with any stage sources generated during the compilation process.

4.2 Tracking the Compile-Error Chain across Stages and their Outputs

Any compilation stage (as well as the final program) is the outcome of a series of previous compilation stages and may never appear in the original source. As a result, to provide a precise compile error report, the compiler has to track down the error chain across all involved stages and combine all relevant information in a descriptive message. To provide such functionality, each AST node is enriched with information about its origin, thus creating a list of associated source references. The source references for each node are created using the following rules: (i) nodes created by the initial source parsing have no source reference; (ii) when assembling nodes for a compilation stage, a source reference is created, pointing to the current source location of the node present in the main AST; and (iii) when updating the main AST, the source locations of the modified nodes are mapped to the latest stage source, creating a source reference.

Rules (i) and (iii) along with the fact that the main AST can be modified only through compilation stage execution guarantee that the main AST nodes will always either be a part of the original source or be generated by some previous stage and have a source reference to it. Furthermore, rule (ii) and the fact that compilation stages are created using only nodes from the main AST guarantee the same property for all compilation stages as well. This means that any AST being compiled, either for some compilation stage or the final program, will incorporate for each of its nodes the entire trajectory of the compilation stages involved in their generation. Figure 5 provides a sample visualization of this information upon the occurrence of an error.

We should note that our approach focuses on providing the source locations involved


main_stage_2_result.dsc

```
1    using std;
2    function generate_function_call(func, args)
3    {
4        return <<~(func)(~(args))>>;
5    }
6    function generate_next_stage_code(code)
7    {
8        return <<!(<<~(code)>>)>>;
9    }
10   (g)(1, 2);
```

main_stage_2.dsc

```
1    using std;
2    function generate_function_call(func, args)
3    {
4        return <<~(func)(~(args))>>;
5    }
6    function generate_next_stage_code(code)
7    {
8        return <<!(<<~(code)>>)>>;
9    }
10   std::inline(<<(g)(1, 2)>>);
```

main_stage_1_result.dsc

```
1    using std;
2    function generate_function_call(func, args)
3    {
4        return <<~(func)(~(args))>>;
5    }
6    function generate_next_stage_code(code)
7    {
8        return <<!(<<~(code)>>)>>;
9    }
10   !(<<(g)(1, 2)>>);
11
12
```

main_stage_1.dsc

```
1    using std;
2    function stage1_function
3    {  }
4    function generate_function_call(func, args)
5    {
6        return <<~(func)(~(args))>>;
7    }
8    function generate_next_stage_code(code)
9    {
10       return <<!(<<~(code)>>)>>;
11   }
12   std::inline(generate_next_stage_code(generate_function_call(<<g>>, <<1, 2>>)));
```

main.dsc

```
1    using std;
2
3    &function stage1_function {}
4
5    function generate_function_call(func, args)
6        { return <<~func(~args)>>; }
7
8    function generate_next_stage_code(code)
9        { return <<!(<<~code>>)>>; }
10
11   !(generate_next_stage_code(generate_function_call(<<g>>, <<1, 2>>)));
12   //no function g available
```

```
1>See original source, file 'main.dsc', line 11.
1>Error, file 'main_stage_2_result.dsc', line 10: Expression '(g)' not a
callable value (its type is 'Undefined').
1>See original source, file 'main_stage_2.dsc', line 10.
1>See original source, file 'main_stage_1_result.dsc', line 10.
1>See original source, file 'main_stage_1.dsc', line 12.
1>See original source, file 'main.dsc', line 11.
1>Finished compiling 'main.dsc', 1 ERRORS detected.
1>Finished compiling 'main.dsc', 2 WARNINGS reported.
1>================ Script main.dsc: Build completed ================
```

Figure 5 – Precise error reporting for compilation stages with a chain of generated sources.

in a compile error but does not affect the error message itself. Since the same compiler executable is used for both normal programs and compilation stages, the same messages are naturally reported upon errors regardless of their origin. We believe that such error messages are chosen by the compiler to provide all relevant information based on the error context and are not related to the meta-compilation process. By providing the complete error chain across all stages and outputs, we essentially provide the *missing information context* required to fully understand the error report. For instance, the error reported in Figure 5 concerns an undeclared symbol called as a function. Looking only at the original source (bottom left) there is no evidence of such a call so the error message makes no sense; however looking at the stage result that actually caused the error (top left) we can spot the erroneous call and understand the error report.

4.3 Discussion: Error Reporting for Runtime Meta-Programming

The above discussion focuses on a system offering compile-time metaprogramming; however error reporting is equally important for runtime metaprogramming. Code assembled at runtime may also generate errors during its translation, and thus require tracking down their origin to be resolved. In this context, we discuss how a methodology similar to that discussed previously could be deployed in languages supporting runtime metaprogramming, using either existing features or possible extensions.

Let's consider multi-stage languages that generate code during runtime based on staging annotations. While it may be possible to ensure the type-correctness of a generated program based on the type-correctness of its generator (for instance this is the case in *MetaML*), there may also be semantic errors that cannot be reported before translating the code at runtime. For such cases, staging annotations should keep track of their locations and combine this information upon AST creations and combinations or insertions into the main program. For a single stage this would resemble the approach used by *Converge* for compile-time error reporting. For instance, consider the following example written in *Mint* [WRI+10], a multi-stage extension of *Java*.

```
Code<Void> code = <|{break;}|>;
for (int i = 0; i < 10; ++i) code.run();
code.run();
```

The above code creates a delayed computation for a break statement and runs it both inside and outside of a loop. When the computation is run inside the loop it should normally execute the break statement and exit the loop. However, when it is run outside of a loop it should produce an appropriate error message referring to the origin of the erroneous computation. In this sense, the code object `<|{break;}|>` should maintain the line information of its origin, possibly combine it with line information from any involved escapes (none in this case), and finally use it during the execution of the run operator to provide an error message similar to the following: *"In line 3, run introduces a 'break' outside of a loop. See original delayed computation at line 1."*.

However, if we have multiple stages or if the AST generation involves combining multiple quasi-quotes present in the original source, reporting just the original source locations of the code segments that generated the erroneous AST would probably be insufficient as any context of the final AST being translated is not available. In this case, we could unparse each of the involved ASTs to generate source segments (as separate files or parts of a file containing all the relevant information) that can then be used for linking the source references of the error message, thus providing the full context of any AST combination and insertion in the entire code generation chain.

Runtime metaprogramming can also be achieved through *reflection*, a language facility allowing to examine or modify the structure and behavior of program code at runtime. This is typically supported by providing the compiler and loader as runtime libraries. For example, Figure 6 shows how two mainstream languages, *C#* [HTWG10] and *Java* [AGH05], can offer metaprogramming through their reflection API. For *C#* in particular, another option is to use the forthcoming *Roslyn* technology [NWGH12].

In this context, the source code to be generated is just typical program data (e.g. dynamic text) and has no special source or line information associated with it. As such, the only way to provide an improved error reporting mechanism in this case, would be to manually insert any source and line information for the generated code along

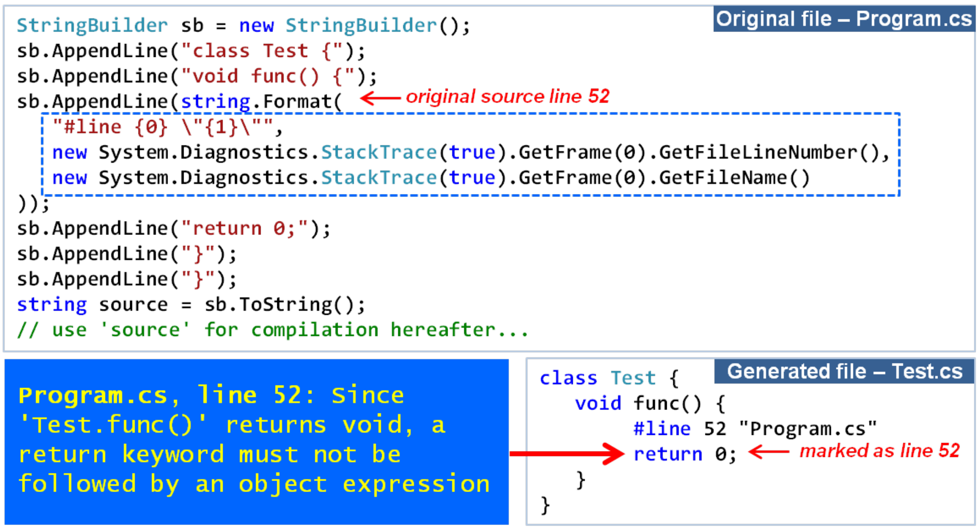
C#

```
string source = "class Test { public void func() { System.Console.WriteLine(\"Hello World\"); } }";
CodeDomProvider provider = CodeDomProvider.CreateProvider("CSharp");
CompilerParameters cp = new CompilerParameters();
cp.GenerateInMemory = true;
CompilerResults result = provider.CompileAssemblyFromSource(cp, source); // invoke the compiler
Assembly assembly = result.CompiledAssembly; // get the compiled assembly
Type type = assembly.GetType("Test");        // get generated class information
Object o = Activator.CreateInstance(type);   // create an instance of the generated class
type.GetMethod("func").Invoke(o, null);      // get and invoke ‘func’ method, printing Hello World
```

Java

```
String source = "public class Test { public void func() { System.out.println(\"Hello World!\"); } }";
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
StringSourceJavaObject src = new StringSourceJavaObject("Test", source);
Iterable<? extends SimpleJavaFileObject> fileObjects = Arrays.asList(src);
compiler.getTask(null, null, null, null, null, fileObjects).call();      // invoke the compiler
Class<?> clazz = ClassLoader.getSystemClassLoader().loadClass("Test");   // load generated class
Object o = clazz.newInstance();              // create an instance of the generated class
clazz.getMethod("func").invoke(o);           // get and invoke ‘func’ method, printing Hello World
```

Figure 6 – Runtime code generation and execution through reflection in *C#* and *Java*.



```
StringBuilder sb = new StringBuilder();
sb.AppendLine("class Test {");
sb.AppendLine("void func() {");
sb.AppendLine(string.Format(
   "#line {0} \"{1}\"",
   new System.Diagnostics.StackTrace(true).GetFrame(0).GetFileLineNumber(),
   new System.Diagnostics.StackTrace(true).GetFrame(0).GetFileName()
));
sb.AppendLine("return 0;");
sb.AppendLine("}");
sb.AppendLine("}");
string source = sb.ToString();
// use 'source' for compilation hereafter...
```

```
Program.cs, line 52: Since
'Test.func()' returns void, a
return keyword must not be
followed by an object expression
```

```
class Test {
   void func() {
        #line 52 "Program.cs"
        return 0;
   }
}
```

Figure 7 – Instrumenting #line directive for better error reporting in *C#*: The compilation error occurs in the generated file but is reported in the original file.

with any potential references to other source locations, if of course such a construct is supported by the language. For example, in *C#* we can utilize the `#line` directive along with `System.Diagnostics.StackTrace` instances to instrument the dynamic source code so that it reports any of its compilation errors directly on the original source lines that generated the error. This functionality is depicted under Figure 7.

If the reflection API supports compiling code based on some AST value, it would be possible to enrich the AST structure with custom source file, line, or source reference information and have the programmer explicitly write code that sets this information in the nodes of the AST to be translated. This way, and considering that the compiler is also extended to utilize such AST information, it would be possible to generate a more detailed error report relating the generated code to the original program instructions producing it, thus facilitating the identification of the error cause.

5 Stage Debugging

For source-level debugging of stages we require: (i) initiating a compile-time debug session utilizing the generated stage sources; (ii) introducing breakpoints for the stage code both before and during this session; and (iii) enabling inspection of AST values.

5.1 Compile-time Debug Session

Stage programs are essentially normal programs, so it is possible to use the standard language and IDE infrastructure to support them with typical source level debugging.

Generally, the debugging functionality is split in two distinct parts, the debugger backend responsible to interoperate with the executing program (i.e. the debuggee) and the debugger frontend responsible for the visual presentation of the process and the user interaction. The former is typically incorporated into the execution system being debugged, and the latter is typically a part of the IDE, while they interoperate

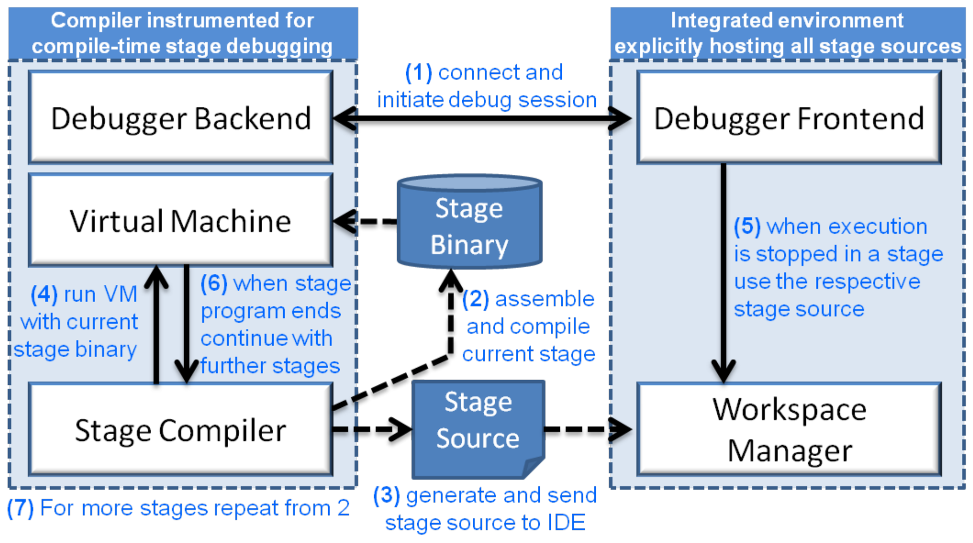

Compiler instrumented for compile-time stage debugging
Integrated environment explicitly hosting all stage sources
Debugger Backend
(1) connect and initiate debug session
Debugger Frontend
Virtual Machine
Stage Binary
(5) when execution is stopped in a stage use the respective stage source
(4) run VM with current stage binary
(6) when stage program ends continue with further stages
(2) assemble and compile current stage
Stage Compiler
Stage Source
Workspace Manager
(7) For more stages repeat from 2
(3) generate and send stage source to IDE

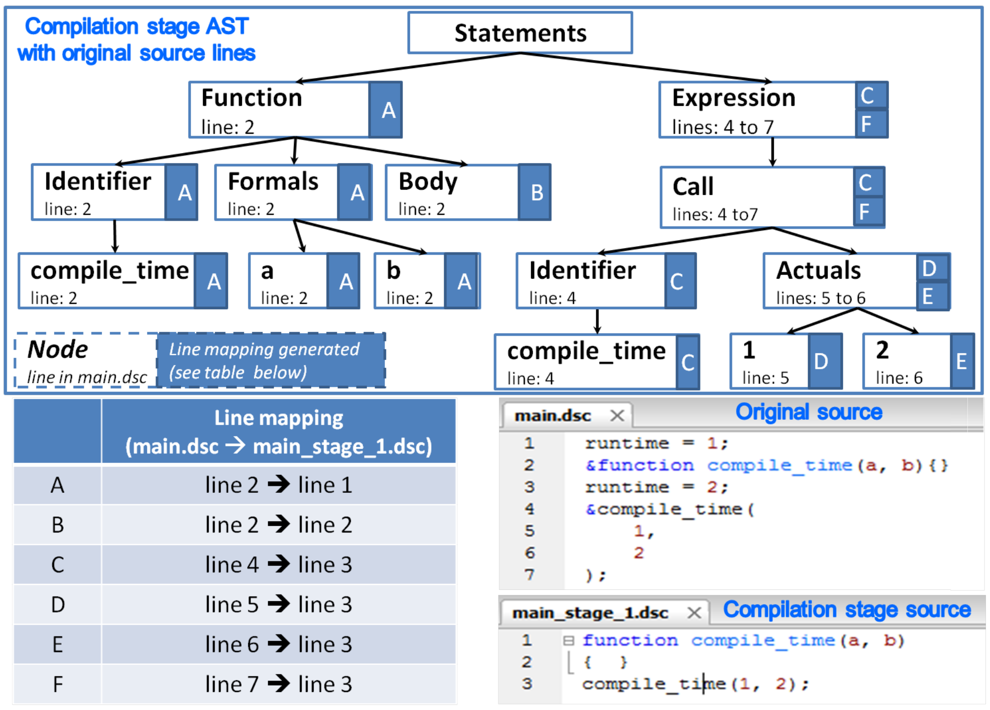
Figure 8 – Interaction between the compiler and integrated development environment for supporting compile-time source-level stage debugging.

with each other through remote communication. In our case, the execution system is the compiler executable, so apart from the functionality related to assembling and executing stages, it also incorporates the debugger backend. Consider that a source is selected to be built with debugging enabled. This will launch the compiler, which in effect activates the debugger backend, connects with the debugger frontend (Figure 8), and the debugging session proceeds as for normal programs. When a stage source is generated, the IDE is notified about its existence and incorporates it into the project management thus allowing its source to be used for debugging purposes. The stage is then translated to binary form and executed by the virtual machine. Upon ending execution, the compiler normally continues with the next stage. During this entire process, the debugger backend is active, so the execution of any compilation stage can be typical debugged. Additionally, the assembly and translation of a compilation is transparent to the debugger meaning that it is possible to debug the entire process in a seamless way; stepping from the last instruction of one stage will stop execution at the first instruction of the next stage without requiring a new debug session.

From the IDE perspective, another requirement for proper compile-time debugging is to properly orchestrate any facilities previously targeted only for build or debug sessions. Essentially, IDEs provide different tools for build sessions (e.g. error messages, build output, etc.) and debug sessions (e.g. call stack, watches, threads, processes, loaded modules, etc.), while applying different visual configurations for each activity. Compile-time debugging involves both a build and a debug session, so the provided facilities should be combined in a way that maintains a familiar working environment.

5.2 Supporting stage breakpoints

Once a stage source has been generated and incorporated into the IDE workspace, we can normally add or remove breakpoint for its code, however that only happens during a meta-compilation round. Prior to that there are no stage sources and no way to



	Line mapping (main.dsc → main_stage_1.dsc)
A	line 2 → line 1
B	line 2 → line 2
C	line 4 → line 3
D	line 5 → line 3
E	line 6 → line 3
F	line 7 → line 3

```
1   runtime = 1;
2   &function compile_time(a, b){}
3   runtime = 2;
4   &compile_time(
5       1,
6       2
7   );
```

```
1   function compile_time(a, b)
2   {  }
3   compile_time(1, 2);
```

Figure 9 – Extracting line mappings for a compilation stage: assembled stage AST (*top*); original source snd compilation stage source (*bottom right*); line mappings generated by each AST node (*next to the nodes, referring to elements of the bottom left table*).

associate breakpoints with their execution. The only information available is limited to any breakpoints associated with the original source being compiled. However, these breakpoints can be utilized to automatically generate breakpoints for the stage source.

The stage AST is composed of main AST nodes and, as previously discussed, main AST nodes are either part of the original source or recursively generated by some previous stage. This means that each stage node originates from one or more nodes of the original AST. Viewing this from a different perspective, the original AST nodes can be associated with the stage nodes they generate, and the same applies for their line information. To achieve this, we extend the unparsing process to associate each node line of the AST being traversed to the current line of the source being generated, taking into account the lines introduced by unparsing (Figure 9). Each AST node generates two (not necessarily distinct) line associations, one upon entering and one upon exiting the traversal. The result is a list with line associations that can be used to transform breakpoints placed in the original source into stage breakpoints.

As shown in Figure 9, the generated line associations are not necessarily one-to-one; effectively this means that a single original source breakpoint may end up generating multiple stage source breakpoints (e.g. a single-line expression that generates a multi-line function) while multiple source breakpoints may end up generating the same stage source breakpoint (e.g. a complex multi-line expression that generates a single line of code). Nevertheless, this is in fact the functionality that a programmer would expect supposing that any code modifications occur directly at the line they appear in within the original source. For instance, an expression generating a function can be seen as substituting itself with a single line containing the function definition. A breakpoint set on the single line function would be hit during the execution of any statement within the function; likewise, the breakpoint of the original source will generate breakpoints for all lines the function expands to, achieving the same functionality.

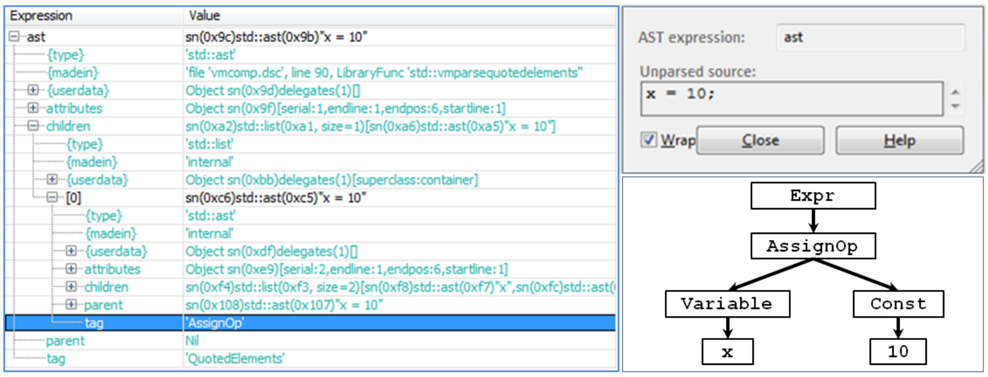
Our original approach for generating stage breakpoints, discussed in [LS12], involved passing the original breakpoints as arguments in the compiler invocation, applying the computed line associations on them within the compiler and finally supplying the resulting breakpoints directly to the debugger backend (present within the compiler executable) that would in turn forward them to the frontend. The idea was to minimize the data communicated to the IDE by keeping the line associations as internal compiler data available only during compilation. However, line associations may also be required by the IDE after compilation, for example to provide better navigation across stage sources and their outputs or to generate breakpoints for the runtime debugging of the final source deploying a similar method with the that used for stage breakpoints. To support this functionality, we adopt the original approach as follows. The line associations for a stage source (or a stage source output) are normally calculated within the compiler during the unparsing and are then propagated to the IDE as meta-data accompanying the stage source. Upon receiving a stage source, the IDE is then responsible to apply the line associations to generate the stage breakpoint based on the breakpoints present in the original source. Of course, the latter only applies to a compilation initiated with debugging enabled. Finally, as in the original approach, any breakpoints generated this way are essentially transient, so they are only kept during the execution of their respective stage and discarded afterwards.

Apart from allowing the IDE to be aware of the line associations, the new approach also provides better modularity and separation of concerns. The compiler's only responsibility is now the generation and provision of the stage sources and their line associations. In the IDE, the line associations now become part of the stage source meta-data and are maintained by the workspace management. Also, the generation and bookkeeping of the stage breakpoints is now part of the typical breakpoint management of the IDE. Finally, the debugger communication between the frontend and the backend now requires no extensions with any breakpoints posted typically by the frontend.

5.3 Enabling AST inspection

The execution of a compilation stage typically targets the modification of the original source being compiled by manipulating code segments in AST form. To debug such operations, we require inspecting such runtime values and browsing through their contents. For example, using a typical expression tree-view we should be able to navigate across a tree hierarchy representing an AST node and inspect any of its attributes (e.g. type, name, value, etc.) or its related tree nodes (e.g. children, parent).

This view may provide all relevant information, but becomes difficult to use as the tree size grows. This is because the AST is visualized only through its root node and specific nodes within the hierarchy can only be viewed after navigating across all other intermediate nodes. For a better visual representation that directly illustrates the entire AST we propose using graphical tree visualizers. This way, programmers may simultaneously observe all AST nodes, the connectivity and relations among them as well as their specific attributes (annotated on the tree or available as tooltip information). Since ASTs represent source code, another viable solution would be to unparse the AST into source code and provide it to the programmer with proper formatting and syntax highlighting. It is even possible to have multiple alternative



Expression	Value
ast	sn(0x9c)std::ast(0x9b)"x = 10"
{type}	'std::ast'
{madein}	'file 'vmcomp.dsc', line 90, LibraryFunc 'std::vmparsequotedelements''
{userdata}	Object sn(0x9d)delegates(1)[]
attributes	Object sn(0x9f)[serial:1,endline:1,endpos:6,startline:1]
children	sn(0xa2)std::list(0xa1, size=1)[sn(0xa6)std::ast(0xa5)"x = 10"]
{type}	'std::list'
{madein}	'internal'
{userdata}	Object sn(0xbb)delegates(1)[superclass:container]
[0]	sn(0xc6)std::ast(0xc5)"x = 10"
{type}	'std::ast'
{madein}	'internal'
{userdata}	Object sn(0xdf)delegates(1)[]
attributes	Object sn(0xe9)[serial:2,endline:1,endpos:6,startline:1]
children	sn(0xf4)std::list(0xf3, size=2)[sn(0xf8)std::ast(0xf7)"x",sn(0xfc)std::ast(
parent	sn(0x108)std::ast(0x107)"x = 10"
tag	'AssignOp'
parent	Nil
tag	'QuotedElements'

AST expression: ast

Unparsed source:

```
x = 10;
```

Figure 10 – Alternative views for inspecting ASTs in *Sparrow*: an expression tree view (*left*), an unparsed source code view (*right, top*) and a graphical view (*right, bottom*).

visualization schemes that can be selected based on the specific program being debugged. Figure 10 provides alternative visualizations for inspecting the AST `<<x = 10>>`.

5.4 Discussion: Staged Debugging for Runtime Meta-Programming

Again, the above discussion was focused towards compile-time metaprogramming. We continue by exploring how stage debugging can be achieved in a language with runtime metaprogramming and discuss directly applicable existing practices, their limitations and possible extensions. As with the compile-time case, we focus on the requirements for stage debugging (i.e. source and debug information) as well as the support for setting breakpoints in the context of the metaprogram. We begin with languages that support metaprogramming through reflection and again use *C#* and *Java* as examples.

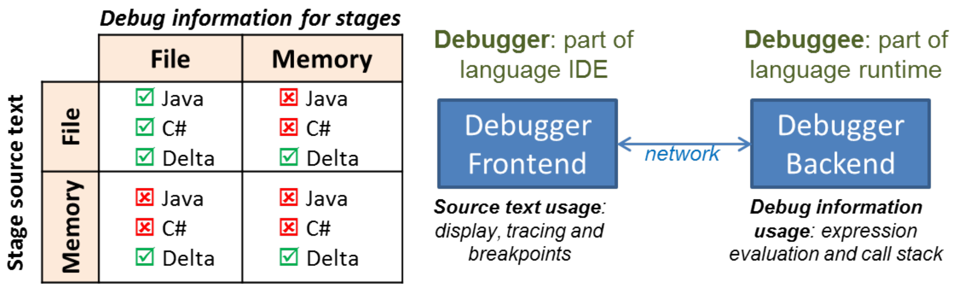
Stage debugging As already discussed, in both *C#* and *Java*, the compiler can be used at runtime to compile any dynamically created source text. The compilation outcome is typically a binary file (dll or exe file in *C#*, class file in *Java*) that can be loaded and executed using the reflection API. Deploying such code within a typical debug session and ensuring that the proper options for debug information are supplied to the compiler invocation, we end up with a normally executing binary that can be debugged. However, we are missing the source information required for true source-level debugging. In *C#*, compilation is performed through a temporary file containing the dynamic source text. Using a compiler option it is possible to specify that the temporary file should be retained after the end of the compilation. This way it can be automatically used by the debugger to trace the execution within the generated code.

In *Java* there is no similar option to automatically generate a source file for debugging purposes. Nevertheless, an existing source file matching the generated binary file can be automatically loaded and used for source-level debugging. This means that manually storing the dynamic source text in an appropriate source file before loading the generated binary file achieves the desired effect. To automatically support this functionality with no programmer intervention, we propose utilizing the information present in the class file to generate a source file through reverse engineering, i.e. using a *Java* decompiler (e.g. [jav]). To this end, the *Java Platform Debugger Architecture* [Ora] would have to be extended with an extra command allowing the debugger frontend to ask the backend for missing source information, while their interplay would be as follows. The backend initially issues a stop-point at a specific source location. The frontend, typically part of the IDE, checks if the target source is present in the project management and if not, it asks the backend to supply a source file. The backend that has access to the generated class file, invokes the decompiler to generate a matching source and sends it back to the frontend. Finally, the frontend receives the source and opens it in an editor to support source-level debugging. From that point, the source file can be used normally, e.g. to allow adding extra breakpoints.

Apart from using the compiler on a dynamic source text, it is also possible to directly emit code in intermediate or final form. In *C#* there is the `Reflection.Emit` namespace containing functionality for generating intermediate language code. When emitting code this way, it is possible to associate it with a source file to be used for debugging. However, this involves manually specifying a source location for each emitted instruction and explicitly providing names for any generated symbols. Standard *Java* libraries do not provide a similar functionality; however it is possible to generate byte-code using a third party library like the *Byte Code Engineering Library* [Com] or *ASM* [Bru07]. The generated binaries can be loaded for execution during a typical debug session, but they do not provide an associated source representation. Nevertheless, with the class file present, we can also use the proposed decompiler methodology to create a source file that can be used for debugging purposes.

A reflection infrastructure is also offered by the *Delta* language. To provide source information for debugging purposes when the original code is stored in a buffer or when only a respective syntax tree is available for translation we use the following approach: The source text is incorporated into the debug information of the generated binary. Once the binary is loaded for execution, the source text from the debug information is extracted by the debugger backend and is posted to the frontend when a breakpoint is hit in a statement of the dynamic source code. Then, the frontend opens an editor for the dynamic source code enabling users to review it and add breakpoints.

In general, the data required to carry out stage debugging consist of the stage source text and the respective debug information. Typically, they are utilized by different components during the debug process and target different tasks; the stage source is used for displaying code and tracing through it as well as setting breakpoints while the debug information is used for expression evaluation and call stack information (Figure 11, right part). As such, they may be stored independently, both either in memory or as files within the filesystem, effectively presenting different options regarding their availability. Supporting each of these options is important towards allowing source-level debugging for all cases; however existing languages typically provide only partial support. For example, in *C#*, it is possible to compile a source present in memory as well as a source present in the filesystem but the generated binary and debug information file (pdb file) is always stored in the filesystem. Even if we explicitly set that generation should take place in memory (using the `CompilerParameters.GenerateInMemory` property), temporary files are always created in the filesystem and are disposed after compilation. This intermediate step may in fact cause unexpected errors to occur if for example the disk quota is exceeded or the application lacks access privileges. Even worse, during a subsequent debug session, the debugger will request these intermediate files to provide source and debug information, however they have been already disposed after compilation (unless explicitly otherwise specified), meaning no proper debugging is possible. Similarly, in *Java*, the debug information is inserted into the generated class file that is always stored in the filesystem and the debugger always requires a source file in the filesystem to match that generated class file. In both cases, the only possibility for a proper debug session is when all required information is available in the filesystem.



Debug information for stages

Stage source text	File	Memory
File	☑ Java ☑ C# ☑ Delta	☒ Java ☒ C# ☑ Delta
Memory	☒ Java ☒ C# ☑ Delta	☒ Java ☒ C# ☑ Delta

Debugger: part of language IDE
Debuggee: part of language runtime
Debugger Frontend
network
Debugger Backend
Source text usage: display, tracing and breakpoints
Debug information usage: expression evaluation and call stack

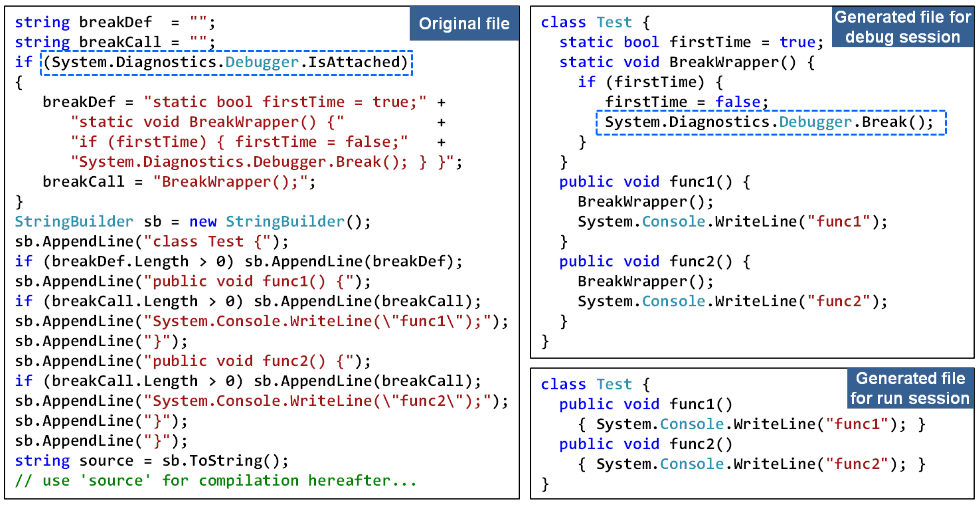
Figure 11 – Left: source-level debugging support for runtime stages in *C#*, *Java* and *Delta* regarding the availability options of stage source text and debug information; Right: the split responsibility in using source text and debug information for debugging.

In *Delta*, where the dynamic source is part of the debug information that is stored within the generated binary it is possible to have that binary stored in both filesystem and memory. In particular, regardless of the stage source text or generated binary presence (either memory or filesystem), *Delta* provides full source-level debugging. The left part of Figure 11 summarizes the different availability options for stage source text and its debug information and the debugging support offered by each language.

Stage breakpoints Once a source file corresponding to the metaprogram being executed becomes available, it is possible to use it for setting any breakpoints. However, the previous methods for providing the source file imply that execution is already stopped within the metaprogram context. Effectively, we need a way to set breakpoints in the generated code *before* it becomes available, or an entry point for breaking execution within generated code and then placing breakpoint directly on its source.

The simplest scenario is to locate the first invocation targeting generated code and use that as an entry point. Within the debug session, we can place a breakpoint at the already present source that will invoke the generated code and then step into that invocation to break execution in the desired context. However, this may not always be possible as the generated code is not invoked necessarily directly after loading. For instance, the generated code may contain callbacks that are only registered upon loading and whose invocation occurs at an unspecified time later in the program execution. In such cases one possible option is to instrument the generated code with explicit directives for breaking system execution (granted of course that the language and execution system provide such a facility). For example, such functionality is provided in *C#* through `System.Diagnostics.Debugger.Break`. In case of a debugged execution (checked at runtime through `System.Diagnostics.Debugger.IsAttached`) it is possible to insert such break invocations where needed in the code to be generated. For the callback example mentioned earlier, that would involve placing a break invocation as the first instruction in each of the callbacks present in the generated code. In general, the same would have to be done for every executable piece of code (e.g. function, method, etc.) present in the generated binary. The latter may result in multiple breakpoints being issued with subsequent invocations of generated code, while the original intent was only to break execution in the first invocation. However, as shown in the code example of Figure 12 this is easily resolved by substituting the original call with a wrapper that only breaks execution in its first invocation.

While the above is an adequate solution to the problem, it requires introducing additional code just for debugging purposes. The original problem narrows down to


Original file
Generated file for debug session
Generated file for run session

```
string breakDef  = "";
string breakCall = "";
if (System.Diagnostics.Debugger.IsAttached)
{
   breakDef = "static bool firstTime = true;" +
      "static void BreakWrapper() {"          +
      "if (firstTime) { firstTime = false;"   +
      "System.Diagnostics.Debugger.Break(); } }";
   breakCall = "BreakWrapper();";
}
StringBuilder sb = new StringBuilder();
sb.AppendLine("class Test {");
if (breakDef.Length > 0) sb.AppendLine(breakDef);
sb.AppendLine("public void func1() {");
if (breakCall.Length > 0) sb.AppendLine(breakCall);
sb.AppendLine("System.Console.WriteLine(\"func1\");");
sb.AppendLine("}");
sb.AppendLine("public void func2() {");
if (breakCall.Length > 0) sb.AppendLine(breakCall);
sb.AppendLine("System.Console.WriteLine(\"func2\");");
sb.AppendLine("}");
sb.AppendLine("}");
string source = sb.ToString();
// use 'source' for compilation hereafter...
```

```
class Test {
  static bool firstTime = true;
  static void BreakWrapper() {
    if (firstTime) {
      firstTime = false;
      System.Diagnostics.Debugger.Break();
    }
  }
  public void func1() {
    BreakWrapper();
    System.Console.WriteLine("func1");
  }
  public void func2() {
    BreakWrapper();
    System.Console.WriteLine("func2");
  }
}
```

```
class Test {
  public void func1()
    { System.Console.WriteLine("func1"); }
  public void func2()
    { System.Console.WriteLine("func2"); }
}
```

Figure 12 – *C#* example instrumenting the generated code with debugger break instructions to stop execution in the context of the generated code during a debug session.

breaking execution after loading the binary and before executing any of its code, so it should be addressed as a breakpoint issue. Apart from normal breakpoints, there are also conditional breakpoints, breaking execution when a condition is met, data breakpoints, breaking execution when the value of some data is changed, and exception breakpoints, breaking execution when an exception is thrown. In the same sense, we propose introducing a breakpoint category dedicated for breaking execution upon loading a binary file. With such a breakpoint, when a generated binary is loaded, execution will be stopped and the debugger will generate a matching source file using one of the earlier discussed methods. This source file can then be utilized to add any normal breakpoints directly in the generated code before it is executed, thus solving the initial problem without requiring code instrumentation and with minimal debugging effort from the programmer (merely enabling the breakpoint when necessary).

Multi-stage Languages Multi-stage languages that support runtime code generation through staging annotations (e.g. *MetaOCaml*, *Metaphor*, *Mint*, *MetaML*) could also use a similar approach to the one discussed for compile-time metaprogramming. In such languages, the stage code is typically available in some AST form that can be unparsed to provide a source file to be used for debugging. Regarding breakpoint support, it should also be possible to associate lines from the original source to lines of the generated source to automatically generate breakpoints for the stage program. Nevertheless, there is a significant difference compared to the compile-time case. Since breakpoints are typically targeted for runtime execution, they do not interfere with the compilation of the original program; however, if code generation occurs at runtime the same breakpoints may be triggered by the normal execution flow of the program even if the intent was to use them just for generating breakpoints for the stage program. The latter essentially demonstrates the need for disambiguating between breakpoints targeted for normal program execution and breakpoints targeted for stage metaprograms. In this sense, we propose introducing a new breakpoint category for *explicit stage breakpoints*. Such breakpoints may be set in the original source just like normal breakpoints but they do not cause execution to break; instead they are only used in

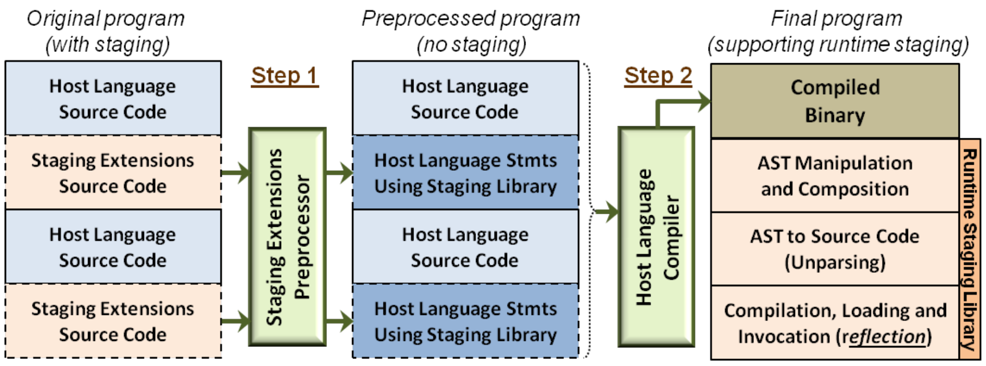

Original program (with staging)
Host Language Source Code
Staging Extensions Source Code
Host Language Source Code
Staging Extensions Source Code
Step 1
Staging Extensions Preprocessor
Preprocessed program (no staging)
Host Language Source Code
Host Language Stmts Using Staging Library
Host Language Source Code
Host Language Stmts Using Staging Library
Step 2
Host Language Compiler
Final program (supporting runtime staging)
Compiled Binary
AST Manipulation and Composition
AST to Source Code (Unparsing)
Compilation, Loading and Invocation (reflection)
Runtime Staging Library

Figure 13 – Sample implementation of a multi-stage language with runtime metaprogramming relying on: (i) staging extensions preprocessing; and (ii) runtime code generation, loading and invocation via the language reflection facilities.

the context of staging and allow generating normal breakpoint for all stage programs based on the previously discussed line associations.

In particular, multi-stage languages based on top of a language that supports reflection, like *Mint* (based on *Java*) or *Metaphor* (based on *C#*) may utilize the reflection mechanism and the approach mentioned previously to support stage debugging. Such a language could be implemented using a stage preprocessor, an AST composition library and the pure base language along with its support for dynamic compilation (Figure 13). Initially, the source file that includes staging annotations is provided to the stage preprocessor that translates them into the appropriate AST composition invocations, while associating them with information about the source and line they originate from. If the run construct is syntactic (e.g. `run code` as in *MetaML*) it is preprocessed as well, translating them into invocation of the reflection API that will dynamically generate code from the constructed AST (directly or by first unparsing them to source text). Otherwise, if the run construct is a normal language invocation part of the AST library (e.g. `code.run()` as in *Mint*), then the invocation of the reflection API is already present in the implementation of the run method of the AST library. In both cases, the reflection API will be used at runtime to dynamically generate and execute the stage code, thus allowing it to be debugged with the existing language infrastructure and extensions we discussed in the previous sections.

6 Evaluation

To evaluate the effectiveness of our system, we created a suite of metaprograms with errors and asked a total of 8 programmers to resolve them. All users were familiar with metaprogramming concepts and had similar skill level but only two of them had previous experience with the features of *Delta* in particular. The metaprograms were divided into two groups, each containing 4 separate scenarios with varying levels of difficulty: easy (1 test), medium (2 tests) and hard (1 test). All users were asked to work on both groups; in one group with the support of our system and in the other one without it. To avoid any bias, half users began with the tool support on the first group and then continued to the second group without it, while the other half began the first group without the tool and then proceeded to the second group using it. The two

Tests	Errors	Difficulty	Average time without tool	Average time with tool	Speedup
T1, T5	1	Easy	9′15″	8′44″	6%
T2, T6	2	Medium	9′24″	6′18″	33%
T3, T7	2	Medium	11′29″	7′51″	32%
T4, T8	3	Hard	18′55″	11′25″	40%

Table 1 – Evaluation results: tests groups were T1-T4 and T5-T8; average values calculated by combining similar difficulty tests per use case (tool/no tool) for all users

users familiar with *Delta* metaprogramming features were put into different groups.

At the beginning of each evaluation session, the evaluator briefly explained the *Delta* metaprogramming features and demonstrated the debugging and error-reporting facilities of the system. During the evaluation, the evaluator outlined each of the metaprograms encountered, briefly describing their purpose and functionality, and provided any necessary assistance to the users. A second evaluator was present to time the user progress and document any comments or opinions they expressed. When the evaluation was completed, users were asked to complete a questionnaire, rating specific aspects of our system on a scale from 1 to 5, 1 being low and 5 being high.

Results show that the average time required for solving the errors using our tool was significantly less than the time required without it. In particular, as shown in Table 1, the speedup increases as the metaprograms become more difficult. The reason for this is that the simpler the metaprogram, the easier it is to understand its behavior just by observing the source code. In fact, for the easy cases some users focused on manually locating the error even when they could have used the tool instead. On the contrary, the longer and harder the metaprogram, the more difficult it is to correctly predict its behavior without additional support. As such, in the more difficult cases, users didn't try to fully understand the code, but instead relied on viewing the metaprogram result to check its correctness and receive hints towards erroneous code generation, as well as debugging the metaprogram execution to find the exact error locations. For the same test and use case, experienced users performed better than the rest. However, there were also cases where a less experienced user with the tool performed better than a more experienced without the tool. This highlights the positive impact of our tool.

A notable comment from a user was that without the tool the reported errors were typically out of context with little or no helpful information. Also, most users noted that our debugging features allowed them to handle even complex errors quite easily.

During the evaluation we observed that some users required similar times to complete an easy or difficult test and less time to complete a task of the same difficulty regardless of the tool support. This is attributed to a learning curve not in the tool but the metaprogramming features of *Delta* and the errors they involve. Practically, after a couple of tests users became familiar with the common error cases and resolved them more quickly in following tests. This explains cases where users of the first group (began with tool) resolved some simple errors faster without the tool. However, the same learning curve applies for the users of the second group (began without tool) that required even less time, thus compensating for the previous negative speedup. Overall, our measurements are not significantly affected by the learning curve.

Table 2 shows the questions asked in the questionnaire, while Figure 14 summarizes the answers received. As shown, users gave high ratings for most facilities offered by

Q1:	How would you rate the ability to view the source code of metaprograms?
Q2:	How would you rate the ability to view the source code of generated programs?
Q3:	How helpful was the chain of source references regarding compilation errors?
Q4:	Did the offered metaprogram debugging facilities help you to resolve errors?
Q5:	Was associating the breakpoints from original program to metaprogram intuitive?
Q6:	How would you rate the breakpoints for metaprograms?
Q7:	How would you rate the AST visualization as source cod?
Q8:	How would you rate the AST visualization using treeview expression watches?
Q9:	How would you rate the graphical visualization for ASTs?

Table 2 – Questionnaire for rating the various facilities of our metaprogramming system

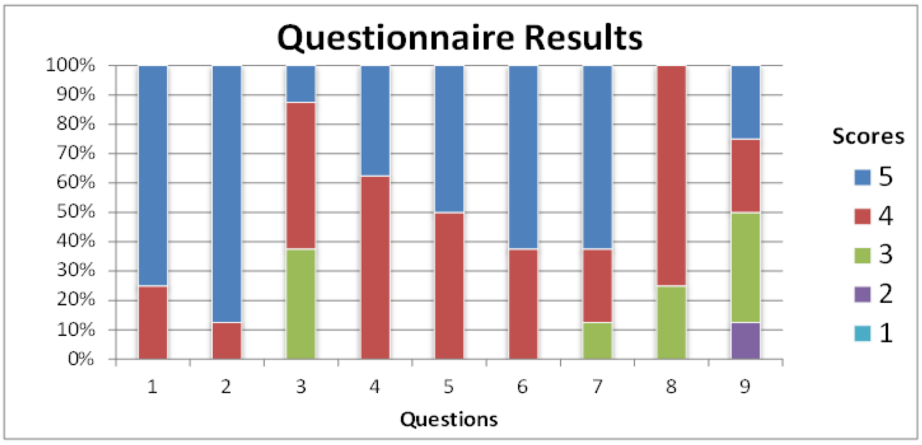


Figure 14 – Results provided by users for the questionnaire shown in Table 2

our system. The lowest score is found in the chain of source references and is due to the fact that most tests contained a single compilation stage, thus making its offered information trivial. For tests with multiple and more elaborate compilation stages we expect it to be much more helpful. Regarding the debugging facilities, users gave higher ratings to the breakpoint support and the AST visualization as source code. Indeed, all users made extensive use of metaprogram breakpoints while they typically utilized the source code view to visualize code segments in a simple and familiar form.

7 Conclusion

In this paper we focused on providing error handling facilities in the context of metaprogramming. Metaprograms are essentially programs, so proper handling is required to resolve errors occurring both during their compilation and execution. However, existing systems provide poor error messages and lack support for metaprogram debugging.

Towards this direction, we implemented a compile-time metaprogramming system that features: (i) precise reports for errors occurring at compilation stages or the final program using source references across the entire code generation chain; and (ii) full-fledged source-level debugging for any stage during compilation. To support these features, we based our implementation on three axes: (i) source files are generated for compilation stages and their outputs and are incorporated into the IDE's project manager associated with the source being built; (ii) the chain of all source locations involved in generating an erroneous code segment is utilized to provide precise error reports; and (iii) original source breakpoints are mapped to breakpoints for stages sources. Since these features are not tightly coupled to metaprogramming, we plan to investigate their application to other program transformation methods like aspect-oriented programming. We evaluated our system by asking programmers to resolve errors in a suite of metaprograms with and without the offered facilities. Results showed that using our system, programmers resolved the errors significantly faster.

Finally, we provided an overview of the amendments required to the compilation process and tool-chain to support such functionality and suggested how it can also be achieved in systems offering runtime metaprogramming. While we mainly focused on an untyped language, the same approach can also be used with typed languages. The difference is that in our case, type errors result into stage execution errors, while a type system could detect them at stage compilation. However, this is a typical trade-off between typed and untyped languages not related to metaprogramming.

Overall, apart from the metaprogramming system we built for the *Delta* language, we believe that our work can provide a basis for extending other metaprogramming systems with similar features, arguably improving the metaprogramming experience.

References

[AG04] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, December 2004.

[AGH05] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley Professional, fourth edition, August 2005.

[Ale10] Andrei Alexandrescu. *The D Programming Language*. Addison-Wesley Professional, first edition, June 2010.

[Baw99] Alan Bawden. Quasiquotation in Lisp. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 4–12, 1999.

[Bec11] Pete Becker. C++ international standard, 2011. Available from: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf.

[BP99] Jonathan Bachrach and Keith Playford. D-expressions: Lisp power, Dylan style. Technical report, 1999. Available from: http://people.csail.mit.edu/jrb/Projects/dexprs.pdf.

[Bru07] Eric Bruneton. Asm 4.0 a Java bytecode engineering library, 2007. Available from: http://download.forge.objectweb.org/asm/asm4-guide.pdf.

[CLT$^+$01] Cristiano Calcagno, Queen Mary London, Walid Taha, Liwen Huang, and Xavier Leroy. A bytecode-compiled, type-safe, multi-stage language. Technical report, 2001.

[Com] Apache Commons. BCEL - Byte Code Engineering Library. Available from: http://commons.apache.org/bcel/manual.html.

[COST03] Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. Dsl implementation in MetaOCaml, Template Haskell, and C++.

In *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 51–72. Springer, 2003. doi:10.1007/978-3-540-25935-0_4.

[des] Descent: An Eclipse plugin providing an IDE for the D programming language. Available from: http://www.dsource.org/projects/descent.

[Fle07] Fabien Fleutot. Metalua manual, 2007. Available from: http://metalua.luaforge.net/metalua-manual.html.

[HTWG10] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Addison-Wesley Professional, 2010.

[jav] Java Decompiler. Available from: http://java.decompiler.free.fr/.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall, second edition, April 1988.

[LS12] Yannis Lilis and Anthony Savidis. Supporting compile-time debugging and precise error reporting in meta-programs. In *TOOLS 2012*, pages 155–170, 2012. doi:10.1007/978-3-642-30561-0_12.

[NR04] Gregory Neverov and Paul Roe. Metaphor: A multi-stage, object-oriented programming language. In *GPCE 2004*, volume 3286 of *LNCS*, pages 168–185. Springer, 2004. doi:10.1007/978-3-540-30175-2_9.

[NWGH12] Karen Ng, Matt Warren, Peter Golde, and Anders Hejlsberg. The Roslyn Project - exposing the C# and VB compiler's code analysis, 2012. Available from: http://go.microsoft.com/fwlink/?LinkID=230702.

[Ora] Oracle. Java SE 7 Java Platform Debugger Architecture. Available from: http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/.

[PMS06] Zoltán Porkoláb, József Mihalicza, and Ádám Sipos. Debugging C++ template metaprograms. In *GPCE 2006, Portland*, pages 255–264. ACM, 2006. doi:10.1145/1173706.1173746.

[Sav05] Anthony Savidis. Dynamic imperative languages for runtime extensible semantics and polymorphic meta-programming. In *RISE 2005, Heraklion, Crete, Greece*, pages 113–128, 2005. doi:10.1007/11751113_9.

[Sav10] Antony Savidis. The Delta Programming Language, 2010. Available from: http://www.ics.forth.gr/hci/files/plang/Delta/Delta.html.

[SBG07] Antony Savidis, Themistoklis Bourdenas, and Yannis Georgalis. An adaptable circular meta-ide for a dynamic programming language. In *RISE 2007, Luxemburg*, pages 99–114, 2007. Available from: http://www.ics.forth.gr/hci/files/plang/sparrow.pdf.

[SBM00] Tim Sheard, Zino Benaissa, and Matthieu Martel. Introduction to multistage programming using MetaML. Technical report, Pacific Software Research Center, Oregon Graduate Institute, 2000.

[She98] Tim Sheard. Using MetaML: A staged programming language. In *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 207–239. Springer, 1998. doi:10.1007/10704973_5.

[She01] Tim Sheard. Accomplishments and research challenges in meta-programming. In *SAIG 2001*, volume 2196 of *LNCS*, pages 2–44. Springer, 2001. doi:10.1007/3-540-44806-3_2.

[SJ02] Tim Sheard and Simon L. Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Notices*, 37(12):60–75, 2002. doi:10.1145/636517.636528.

[SMO04] Kamil Skalski, Michal Moskal, and Pawel Olszta. Metaprogramming in Nemerle, 2004. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.8265&rep=rep1&type=pdf.

[Str00] Bjarne Stroustrup. *The C++ Programming Language: Special Edition.* Addison-Wesley Professional, third edition, February 2000.

[Tra05] Laurence Tratt. Compile-time meta-programming in a dynamically typed OO language. In *Proc. of the 2005 Symposium on Dynamic Languages, DLS 2005*, pages 49–63. ACM, 2005. doi:10.1145/1146841.1146846.

[Tra08] Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.*, 30(6):1–40, 2008. doi:10.1145/1391956.1391958.

[TS00] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000. doi:10.1016/S0304-3975(00)00053-0.

[Tur94] Kenneth J. Turner. Exploiting the m4 macro language. Technical report, University of Stirling, September 1994.

[Vel96] Todd Veldhuizen. Using C++ template metaprograms. In *C++ gems*, pages 459–473. SIGS Publications, Inc., New York, NY, USA, 1996.

[WRI$^+$10] Edwin M. Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. In *PLDI 2010*, 2010. doi:10.1145/1806596.1806642.

Availability information

Our metaprogramming system is fully functional and its complete source and code examples are available for public download through our Subversion repository https://139.91.186.186/svn/sparrow/ using a guest account (username: 'guest' and empty password). A video showing the metaprogramming features of our system is available from: http://www.ics.forth.gr/hci/files/plang/metaprogramming.avi.

About the authors

Yannis Lilis owns an MSc from the Department of Computer Science, University of Crete, being currently a PhD student collaborating with the Institute of Computer Science - FORTH. His e-mail address is lilis@ics.forth.gr.

Anthony Savidis is an Associate Professor of 'Programming Languages and Software Engineering' at the Department of Computer Science, University of Crete, and an Affiliated Researcher at the Institute of Computer Science, FORTH. His e-mail address is `asics.forth.gr`.