An Integrated Algorithm and Software Debugging Tool for Signal Processing Applications

Bernd Geiser [1], Stefan Kraemer [2], Jan Weinstock [2], Rainer Leupers [2], Peter Vary [1]

[1] Institute of Communication Systems and Data Processing (ind)
RWTH Aachen University, Germany
geiser@ind.rwth-aachen.de

[2] Institute for Integrated Signal Processing Systems (ISS)
RWTH Aachen University, Germany
kraemer@iss.rwth-aachen.de

Abstract

Today's software debugging tools are generally tailored to the needs of programmers and software developers. However, the needs of algorithm developers are only poorly supported. In this paper, it is proposed to enhance traditional software debugging tools with powerful algorithm analysis functionality, as e.g. provided by Matlab. Thereby, the focus is on applications in digital signal processing where new algorithmic debugging tools which operate on the C level can significantly simplify the development process. This idea has been implemented for the integrated development environment (IDE) Eclipse and the GNU project debugger GDB. The enhanced debugger provides, among others, the following features: breakpoints with data transfer to / from Matlab, nonintrusive data sampling, support for user-defined processing scripts. The efficacy of the proposed tool is demonstrated in an example debugging session. As a representative algorithm in digital signal processing, the Adaptive Multirate (AMR) speech codec which is widely used in GSM cellular networks has been selected.

1 Introduction

The typical development process for algorithms in digital signal processing (DSP) can generally be divided into four subsequent steps:

(1) Algorithm design with specialized numerical tools, e.g., *Matlab* [5] or *Octave* [3],

(2) Implementation in a high level programming language (mostly *C*),

(3) Conversion to fixed point arithmetic and

(4) Optimization for the target DSP platform,

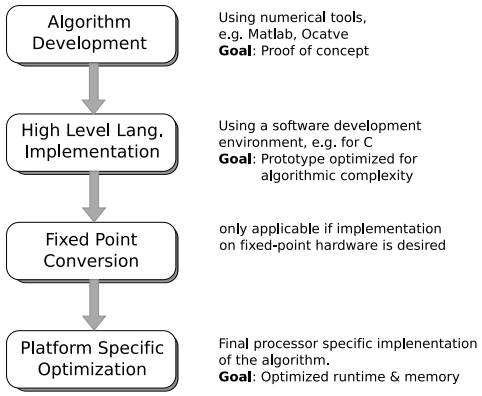


Figure 1. Development steps for applications in digital signal processing.

as illustrated in Fig. 1. Considering the *development tools* utilized for each step, a discontinuity can be observed in particular between steps (1) and (2), i.e., with the migration from the numerical tools to a classical high level programming language like C.

In step (1), i.e., from the algorithm designer's perspective, the use of tools such as Matlab is very attractive because of their powerful language features which allow convenient and rapid prototyping. Moreover, Matlab is often highly appreciated for its algorithm analysis capabilities. Due to its interpretive nature the designer can apply advanced analysis tools (e.g. statistical and spectral analyses) at arbitrary positions within the code. However, with the C implementation, i.e., after the transition from step (1) to step (2), the advanced analysis and debugging capabilities of Matlab are lost. Of course, this is also true if the algorithm design is directly conducted in C, i.e., if steps (1) and (2) are merged. As a matter of fact, the available debugging tools for high level languages like C have an entirely different focus. Clearly, these tools are geared towards soft-

ware developers and programmers instead of algorithm designers. Prominent examples are Microsoft Visual Studio, Eclipse, the GNU project debugger (GDB) [2] and memory inspection tools like Purify and Valgrind.

In this paper, we present a new tool that combines algorithm and software debugging capabilities. It is especially tailored towards signal processing applications. Our approach is to reuse the powerful algorithm design and analysis functionality provided by Matlab within widely used software debugging tools. This idea has been implemented for the integrated development environment (IDE) Eclipse [1] and the GNU project debugger GDB [2].

This paper is organized as follows: Sec. 2 summarizes state-of-the-art software debugging techniques and analyzes their usability in the context of DSP algorithm development. Based on the identified deficiencies, our new debugging concept is proposed in Sec. 3. Realization and implementation aspects are presented in Sec. 4. A case study demonstrates the viability of the proposed concept (Sec. 5) before the paper is concluded.

2 Software Debugging: The Algorithm Designer's View

Traditionally, software debugging tools (SDTs) for high level programming languages (like C) help the developer to pinpoint programming mistakes and errors. In that respect, SDTs are essential for a smooth and efficient development process. Debuggers typically provide a well-defined set of capabilities, including:

- *Controlled execution* (stepwise program execution, breakpoints, conditional breakpoints, source code inspection),
- *Memory and variable inspection* (context-dependent value printing, watchpoints),
- *Callstack inspection* (stack backtrace),
- *Program and variable modification.*

With this functionality, the software developer is well-equipped to tackle numerous common problems such as illegal memory access. However, in contrast to these classical programming mistakes, errors on the *semantic* level are comparatively hard to pinpoint and require expert knowledge about the expected behavior of the program. Ideally, the information provided by the debugging tool in conjunction with the expert knowledge suffices to identify and fix the respective problem in a reasonable time frame.

However, in the domain of signal processing, the raw data presented by the debugger is insufficient and even the DSP expert can not comprehend essential features and characteristics. Therefore, in most cases, it is hard to come to a final conclusion.

```
 0: void fft(int n, double *A_re, double *A_im,
 1:          double *W_re, double *W_im)
 2: {
 3:   double w_re, w_im, u_re, u_im, t_re, t_im;
 4:   int m, g, b, i, mt, k;
 5:
 6:   for (m=n; m<=2; m=m>>1) {  /* Bug 1 */
 7:     mt = m >> 1;
 8:
 9:     for (g=0,k=0; g<n; g+=m,k++) {
10:       w_re = W_re[k];
11:       w_im = W_im[k];
12:
13:       for (b=g; b<(g+mt); b++) {
14:         t_re = w_re * A_re[b+mt] -
15:                w_im * A_im[b+mt];
16:         t_im = w_re * A_im[b+mt] +
17:                w_im * A_re[b+mt];
18:
19:         u_re = A_re[b];
20:         u_im = A_im[b];
21:         A_re[b] = u_re + t_re;
22:         A_im[b] = u_im - t_im; /* Bug 2a */
23:         A_re[b+mt] = u_re + t_re; /* Bug 2b */
24:         A_im[b+mt] = u_im - t_im;
25:       }
26:     }
27:   }
28:   permute_bitrev(n, A_re, A_im);
29: }
```

Figure 2. Erroneous radix-2 FFT implementation

A very simple example is shown in Fig. 2. This piece of code is used to illustrate the above mentioned problem. The first mistake in this radix-2 FFT implementation can be easily detected: The loop condition in line 6 (`m<=2` instead of `m>=2`) is wrong since the entire FFT computation is bypassed. This can be easily detected with a common debugging tool by step by step execution. However, the second mistake in lines 22 and 23 is much harder to locate, since it only affects the actual values produced by the function.

A makeshift solution which is frequently applied in such situations is manual instrumentation of the C application for data logging purposes. The obtained log files are then post-processed with suitable tools (Matlab). In the case of the FFT example, the developer would manually instrument the FFT in order to locate the problem (see Fig. 3). Fig. 4 shows real (a) and imaginary (b) part of the erroneous FFT (array variables `A_re` and `A_im`) as well as the reference FFT (c), (d) as computed with an external tool based on the captured input signals. Owing to the improved presentation of the data, it is immediately obvious that the symmetry conditions for real-valued FFTs are violated. Based on this knowledge it is possible for the algorithm designer to locate the problems in lines 23 and 24 (wrong signs).

There are several drawbacks of these "ad-hoc" approaches (manual instrumentation). First, they are mainly developed "in-house" and they tend to be too specialized to

```
 0: void fft(int n, double *A_re, double *A_im,
 1:          double *W_re, double *W_im)
 2: {
 3:   /* INSTRUMENTATION for input */
 4:   write_double("A_re_input.dbl", A_re, n);
 5:   write_double("A_im_input.dbl", A_im, n);
      ...
 9:   for (m=n; m>=2; m=m>>1) {
        ...
12:     for (g=0,k=0; g<n; g+=m,k++) {
          ...
16:       for (b=g; b<(g+mt); b++) {
          ...
28:       }
29:     }
30:   }
31:   permute_bitrev(n, A_re, A_im);
32:
33:   /* INSTRUMENTATION for output */
34:   write_double("A_re_output.dbl", A_re, n);
35:   write_double("A_im_ouptut.dbl", A_im, n);
36: }
```

Figure 3. 'Ad-hoc' code instrumentation

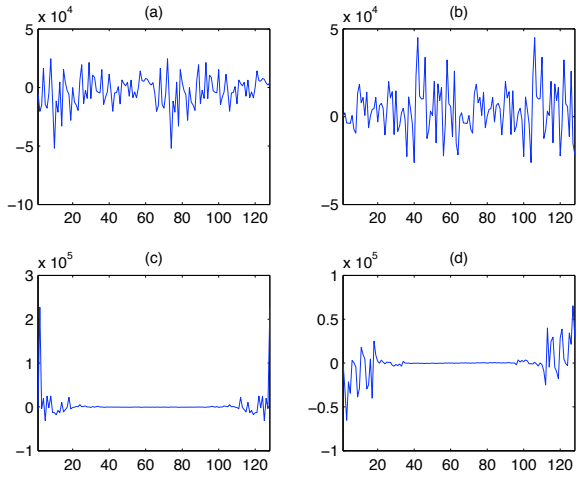


Figure 4. Real part of the erroneous FFT a) Imag. part of the erroneous FFT b) Real part of the reference FFT c) Imag. part of the reference FFT d)

be applied in another application. Then, they are intrusive, i.e., the application source code is augmented by unnecessary overhead. Potentially, new bugs could be introduced. In the following section we describe the proposed debugging concept to overcome such problems.

3 Proposed Debugging Concept

As described previously there mainly exist two different views on an application; the algorithm view and the software view. Traditional debuggers are developed to satisfy the needs of software developers but they are neglecting the algorithm centric view on the application.

Our proposal is a step towards a unified development environment targeted at the software itself as well as its algorithmic aspects. Therefore, a connection between the existing software debugger and the existing algorithm analysis and design tool (e.g. Matlab) is established based on the available application interfaces. For such a tool, the following functionality is desirable from the algorithm developer's view :

(1) Non-intrusive algorithm analysis.

(2) Data transfer between both systems with automated mapping of data structures.

(3) Data sampling (collection for gathering statistics and evaluation at the end of the debug session).

(4) Easy visualization and presentation of (post-processed) data.

(5) A rich library of analysis methods.

(6) Conditional breakpoints triggered by the outcome of the algorithmic analysis.

The non-intrusiveness is already guaranteed by common software debuggers which allow to inspect the software at run-time. Thereby, the debugger provides a common interface to various debug targets such as programs on the host PC or even on external DSP hardware. The latter case opens up the possibility to inspect an algorithm running on this DSP within the familiar Matlab environment.

In order to keep the suggested link between debugger and algorithm design framework as general as possible, changes to the software debugging tool should be kept at a minimum level. Therefore, two operation modes for the augmented debugging environment are suggested:

- A classical breakpoint that automatically transfers selected variables to the connected algorithm design tool.
- A data logging mode based on conditional breakpoints (inspection points)

When hitting an extended breakpoint, the transferred data is immediately visible within the connected tool and can be freely manipulated, post-processed and visualized. Additionally, the outcome of the analysis can serve as a condition to trigger the breakpoint. This way, as opposed to classical conditional breakpoints, it is possible to interrupt the software execution based on *derived* characteristics and parameters (and not only based on memory content).

The *data logging functionality* is motivated by the fact that virtually all signal processing algorithms process their

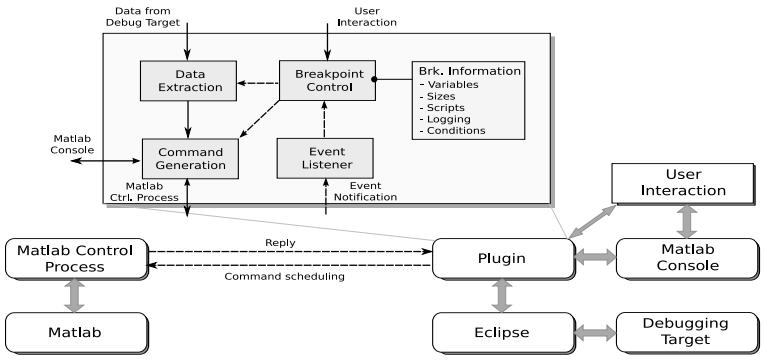


Figure 5. Eclipse / Matlab coupling: Process overview and interaction.

data in a block based manner. In that case, the observation of the instantaneous variable value for a particular time instance does not reveal the underlying statistics. Therefore, the proposed non-intrusive data logging approach is an adequate way to collect and evaluate the temporal evolution of variables and derived parameters (e.g. statistics).

With the above proposed unified development environment, software and algorithm debugging can be conducted simultaneously in a convenient way. For instance, the following use cases are targeted:

(1) Development of signal processing algorithms in C.

(2) Classical software debugging of signal processing applications.

(3) Algorithm verification based on reference data or on a reference implementation. Algorithmic verification helps to ensure that no mistakes have been introduced while implementing the application in C, e.g., based on a Matlab reference.

(4) Migration to a different type of arithmetic, e.g. fixed point conversion. For this task, it is essential to observe the statistics of introduced rounding errors compared to the floating point reference.

(5) Algorithm evaluation and characterization. This use case is particularly important if entire algorithm development has been performed at the C level and no reference is available.

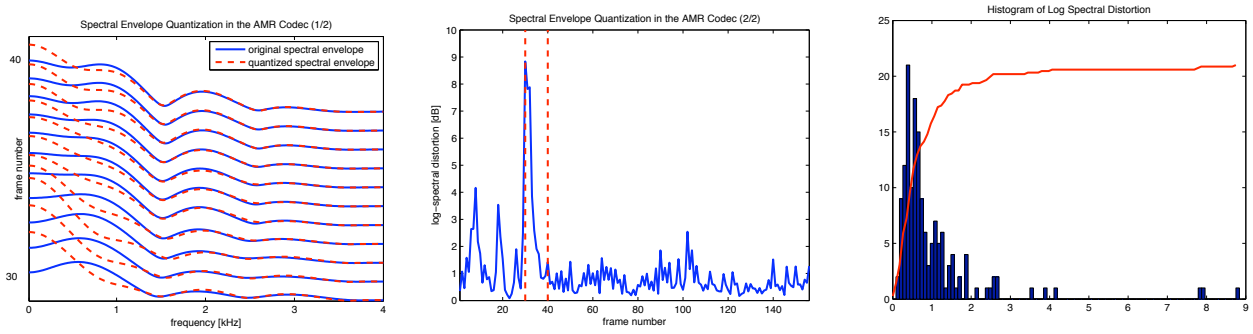
4 Realization

A prototype of the above mentioned idea has been realized. The implementation is based on the Eclipse Integrated Development Environment (IDE) [1] whose debugging facilities rely on the GNU debugger [2]. The main components of the coupling with Matlab are summarized in this section.

Figure 5 shows three major building blocks that are involved: The target process which is to be debugged, the Eclipse IDE and the Matlab environment. In addition, there is an auxiliary component (Matlab control process) to launch Matlab and to enable direct interaction with the Matlab programming interface. On the Eclipse side, a plugin is provided to extend the debugging functionality. The first task of the plugin is to instantiate the Matlab control process and to establish the connection to Matlab. Second,the breakpoints within Eclipse are augmented with additional information concerning Matlab interaction:

- A list of transferable variables. By default, the complete list is forwarded to Matlab. However, the user may specify a customized list to reduce the amount of transferred data.
- An associated Matlab m-script to be triggered upon break. The default behavior is to pass the control to an interactive Matlab session. A Matlab console has been integrated into the Eclipse GUI for this purpose.
- A name for a Matlab structure (“namespace”) to store the transferred data. This structure is reset for each breakpoint hit, but the rest of the Matlab workspace is persistent.
- A condition in Matlab syntax which will be evaluated at the respective code location based on the Matlab workspace. This way, conditional program interception based on derived parameters can be realized.
- An option is provided to exploit the breakpoint for data logging purposes. In this case, the target program is not stopped, but only the associated m-script is executed. This script is mainly intended to collect data over several hits which is particularly important for block based signal processing.

If an augmented breakpoint is hit, the variable transfer to Matlab is initiated and a suitable Matlab command is generated and scheduled for execution by the Matlab control



(a) (b) (c)

Figure 6. Analysis plots of the AMR encoder for 12.2 kbits/s— generated by Matlab based on captured information from the debug target — (a) magnitude of the spectral envelope and its quantized version for 11 speech frames — (b) logarithmic spectral distortion (LSD, integrated quantization error) per frame — (c) histogram and cumulative distribution function (CDF) of the LSD

process. Scalar data types and statically defined arrays can be automatically transferred without any further user interaction. However, if pointers shall be dereferenced, neither the allocated field size nor the underlying Matrix dimensions are known to the debugger. Therefore, these details have to be provided by the user for each breakpoint.

5 Case Study

To demonstrate the efficacy of the proposed tool, the Adaptive Multirate (AMR) speech codec [4] has been selected as a representative algorithm of digital signal processing. The AMR codec, which is based on the principle of Code Excited Linear Prediction (CELP), is widely used for speech transmission in GSM networks. Its bit rate modes, ranging from 4.75 up to 12.2 kbit/s, are usually set by the operator according to the current radio link quality.

The C implementation of the encoder component has been executed on a PC while being inspected by the modified Eclipse debugger. To visualize and analyze the operation of the algorithm, data logging has been realized by an augmented breakpoint. In particular, the quantization of the spectral envelope of the speech signal to be coded is of interest. This spectral envelope is computed and quantized for each speech sub frame of 5 ms length. Concretely, the C arrays describing the computed and the quantized spectral envelope parameters are transferred to Matlab each time the respective breakpoint is triggered, i.e., for each speech frame. The breakpoint has been configured for data logging mode and a specific m-script has been used to collect the data. At the end of the execution run, the collected data has been analyzed and visualized as shown in Fig. 6. From this data presentation, the speech coding expert can easily spot problematic signal portions where the encoder exhibits suboptimal performance. The identified critical signals could be used within further testing sessions and help to improve the algorithm.

6 Conclusions & Outlook

The presented debugging tool alleviates the algorithm developer from certain time-consuming debugging tasks. A uniform framework based on the Eclipse IDE and on Matlab has been realized for this purpose. Of course, this concept can in principle be transferred to other debugging tools. Moreover, the integration of algorithm design tools like Matlab into the software development process allows debugging on a semantic level. Here, this has been shown for the domain of digital signal processing. Nevertheless, the general idea to combine software development tools with information from other sources and the incorporation of semantics is expected to gain interest.

For our concrete realization, a couple of future extensions are imaginable. For instance, support for user defined data types and the related mapping to the Matlab workspace would be interesting. Also, the applicability of our tool in multi-core environments is still an open question.

References

[1] Eclipse Integrated Software Development Environment. `http://www.eclipse.org`.

[2] GDB: The GNU Project Debugger. `http://www.gnu.org/software/gdb/`.

[3] GNU Octave. `http://www.octave.org`.

[4] ETSI Recommendation GSM 06.90. Digital cellular telecommunications system (phase 2+); adaptive multi-rate (AMR) speech transcoding. version 7.2.1, release 1998, Apr. 2000.

[5] The MathWorks. Matlab. `http://www.mathworks.com`.