Exploring the design of compiler feedback

Thibault Raffaillac

School of Computer Science and Communication
KTH, Royal Institute of Technology, Stockholm
traf@kth.se

Keywords: POP-I.B. transfer of competence, POP-II.B. maintenance, POP-III.D. compiler feedback

Abstract

Nowadays, programmers willing to start optimising their code must undergo a lengthy interaction with dedicated profiling tools. This paper proposes as an alternative to make compilers generate feedback messages aimed at explaining how they understand the code, and how it could be improved. The study aims at foreseeing the technical integration of feedback notifications in modern compilers, as well as sketching how Integrated Development Environments (IDE) would display them.

A first comparison of three related works enables the core differentiators to be highlighted: letting the compiler inform where code is actually fine and does not need any refinement, displaying the notifications along the relevant source lines rather than in a separate interface, insisting on the absence of artificial intelligence, and evaluating the importance of each message to filter in a handful. Then, a preparatory field study is carried to observe different programmers and poll their receptiveness to a compiler feedback. The findings relate the usefulness of optimisations' suggestions to fit where users lack expert knowledge, the existence of dormant interrogations calling for serendipitous information retrieval, and the avoidable mistakes inherent to Message of the Day windows.

Three prototypes are designed to embody three distinct approaches, using Web tools to provide an appearance close to code editors along with decent interactivity. With the help of a new user study with the prototypes, a final set of refinements is discussed so as to shape a coherent result and differentiate it further: users can create and share sets of feedback messages to supplement the ones included in their compiler, a list of rules is provided to help designers compose the messages, an emphasis is laid on transparency to help exhibit the absence of artificial intelligence, and the heuristic used to evaluate the importance of messages is sketched.

1. Introduction

Compiling a program nowadays is simple. Once the source code itself is written, a single action is needed to turn it to an executable file. With an IDE such as Eclipse or Microsoft Visual Studio, it is synonym with clicking on a "Build" button. With a command-line compiler such as the GNU Compiler Collection (GCC), it is computed with a single command. Past the errors and warnings, as soon as a working executable is output the compiler will not provide any more interaction.

When it comes to optimization, however, the procedure becomes trickier. By setting the proper options and command-line flags, it is normally handled transparently by the compiler, yet in practice this support is irregular (see Aho, Lam, Sethi, & Ullman, 2006, for a technical overview of modern compilers). While instructions scheduling and register allocation are decently achieved nowadays[1], improvements such as making parallel loops or exploit the locality of memory accesses *will* require tuning specific options, or the source code itself[2].

On the other hand, a significant knowledge gap separates the programmer from the compiler. For the

[1] With the example of GCC, see http://gcc.gnu.org/wiki/InstructionScheduling and http://gcc.gnu.org/wiki/RegisterAllocation (both accessed 03.09.2012).
[2] See the example of GCC at http://gcc.gnu.org/onlinedocs/libgomp/Enabling-OpenMP.html, and for Intel at http://software.intel.com/articles/automatic-parallelization-with-intel-compilers/ (both accessed 03.09.2012).

former, the language syntax[3], the diversity of architectures and systems, and the basic skills required for a software engineer (Kreeger, 2009; Lethbridge, 2000), are potentially overwhelming. For the latter, as quoted from Bose (1988), *the compiler is not designed to fully "understand" the high-level, algorithmic intentions expressed by the user in his (or her) source code*.

As a consequence, it is believed that users do not obtain the performance and security they should expect from their programs. This paper is based on a Degree Thesis which investigated the design and integration of feedback messages from the compiler, to leverage opportunities of tuning and improve the user's mastery in software programming. The method was structured based on Saffer (2009) proposed methodology, with problem framing as Introduction, competitive analysis and differentiators as Related work, design research and structured findings as Preparatory study, prototyping in the dedicated chapter, and testing as User study and future work. My own experience being that of a C/C++ programmer with a passion for performance, I focused on C++, as this general-purpose language is widely used in industry nowadays.

1.1. Problem definition

The compiler should extend interaction *after* the creation of a working executable, to suggest opportunities of improvement, for example. Software such as Intel VTune Amplifier, SmartBear AQtime Pro, or Microsoft Visual Studio Analyzer, can typically provide such functionality. However, they are rather a collection of tools, and require dedicated learning, yet the knowledge gap is not intended to be *dug*. The suggestions should not require the user to learn a convoluted interface, hence the idea of mere feedback messages.

Such notifications are not limited to suggestions; fundamentally they provide feedback on the compiler's operation while parsing the source code, on how it was "understood". They could assure that a hand-coded optimisation is unnecessary when it is transparently done by the compiler. Furthermore, they would allow the removal of conditional compilation features such as preprocessing and generics from programming languages. Indeed, nowadays compilers are capable of evaluating certain run-time expressions at compile-time. With lack of information, however, users still resort to the dedicated syntax to enforce early evaluation. Now, if a user is aware of *which conditions* trigger the former behaviour, the language can be trimmed from its compile-time semantics.

Beyond feedback, with a little more work the compiler could be able to directly query the programmer, to suggest local tunings when the language semantics show their limits. Think about the problem of specifying the underlying search tree structure of a `set` object. Since the C++ standard library does not offer this choice, users must cope with the default implementation shipped with their compiler. If the latter provides several implementations though, it could probably be specified through *pragmas* (the compiler-specific preprocessing instruction in C/C++) but again this would require learning the particular syntax. An alternative here would be to generate a query with radio buttons for the programmer, and automatically insert the correct corresponding `#pragma` call.

1.2. Challenges

Many potential issues were identified ahead of this work, with the help of the reactions to simulated intelligent help highlighted in Carroll (1988). The biggest difficulty would be to properly relate the messages to their context, that is to output messages which actually *interest* the user, so as not to be disabled like a *Message of the Day* in IDEs. Technically, suggesting improvements, guessing the critical parts of a program, querying the programmer and binding a low-level transformation to the source code are challenging tasks. In order to avoid calling for unmanageable artificial intelligence, the system will aim for a simple implementation. Finally, for a reasonable project the amount of notifications are expected to become tremendous, hence the need to filter them, as discussed further. With real-world compilers in mind, the work was conceived to be a realisable project amidst the acknowledged difficulties. See the differentiators and future work for discussion on how they are tackled.

[3] Refer to the C++ specification, for example (http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372, accessed 03.09.2012).

2. Related work and differentiators

I started by analysing three previous similar attempts (Table 1), then identified a few differentiators so as to make this thesis a conceivable choice for real-world development environments.

	VISTA (Zhao et al., 2002)	EAVE (Bose, 1988)	Matlab Code Analyzer
Learning the interface	Moderate: GUI separate from the editor	Lengthy: text interface separate from the editor	Easy: advisory messages under the relevant lines
Preliminary knowledge	Transformations (cited by name) and RTL	Transformations and capabilities of the machine	None required, everything is explained, technically
Amount of information	Huge: showing step-by-step optimisations on RTL	Decent since limited to loops	Decent, but limited to critical advices
Limits	No binding to original source	Only vectorisation, need to *request* advice	No simple feedback or querying for tuning

Table 1 – Comparison of three related works

2.1. The compiler queries the programmer

A mere advice solicits the user where code needs updating; here we introduce the possibility to simply inform where code was properly understood, as well as enable direct querying of the programmer. More specifically, interaction with the programmer can be divided into four distinct tasks (Figure 1). The compiler should be able to:

- *inform*: tells how well a portion of code was compiled, relates a compilation technique, promotes a coding practice, introduces a feature from the standard, etc.
- *alert*: incites the user to correct a supposed flaw, notifies about a potential vulnerability which could arise with further lack of attention, recommends a performance tweak. This is the task at hand in Code Analyzer. As opposed to the previous task, here we request some code to be fixed. Also, the difference with compiler warnings is that the latter concern code which might not execute with the intended meaning, though here only tuning is involved.
- *ask*: inquires a clarification about a portion of code. Sometimes an alert is insufficient, when the clarification cannot be expressed by updating the code. In cases like choosing the implementation of a tree structure (`set` or `map` in C++) or the character set (Latin-9, UTF-8, etc.) of a `string` object, the interface could trigger radio buttons to query the programmer. This task would further benefit from languages being designed with a supplemental *detailed* semantic, accessible with *pragmas* or menus in the IDE to avoid burdening the main syntax.
- *answer*: responds to a direct interrogation from the user. The requirement for artificial intelligence is not discussed in this paper, though an attempt to test its design was made further in the second prototype.

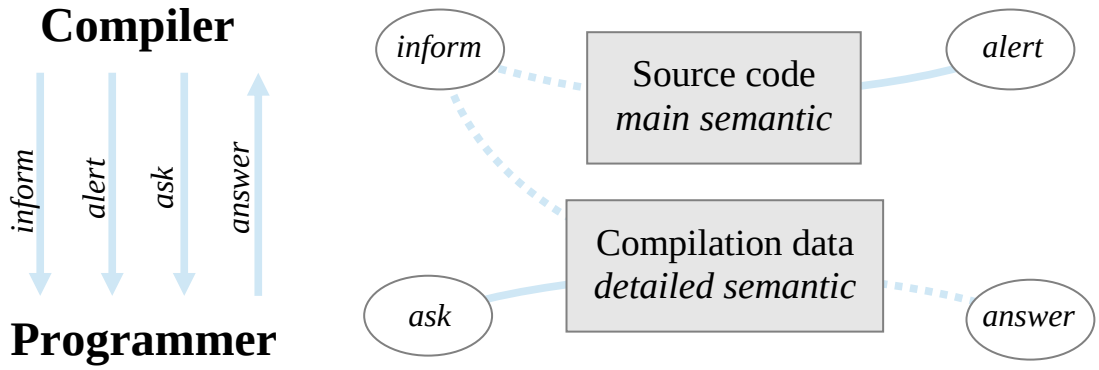

Figure 1 – On the left image, arrows indicate who initiates the communication. On the right one, lines bind each task to which data is discussed (a dotted line indicating the data is not to be modified).

2.2. No separate interface

VISTA, EAVE, and to a greater extent profiling tools in general provide the performance analysis and suggestions of improvement as an interface distinct from the editor. This requires users to learn how to use it, and this knowledge acts as a disincentive on their commitment to start profiling. Here we will bind the messages to the relevant source line, much as in Code Analyzer.

No interface has to be learned here, the only limit to the user's will to tune the program is the messages' clarity, which is discussed further in this paper. Moreover, such an interface can easily be implemented: in practice a compiler like GCC already outputs a line number along with every warning and error, and an IDE such as Eclipse is able to display warnings and errors in relation to the targeted line. This also helps the context to be accurately identified, as opposed to one of the challenges exposed earlier.

2.3. Cooperation instead of assistance

VISTA, EAVE and to a lesser extent Code Analyzer illustrate what an *assisting* agent is: it waits for the user to request help, it does not query him/her, and it focuses on helping the user fix an issue rather than improving his/her knowledge. By contrast, a *cooperation* is similar to a discussion, in which both interlocutors can engage the conversation, ask questions and answer them. Moreover, there must be no assumption that the programmer is familiar with any of the concepts involved. Contrary to VISTA and EAVE which refer to the optimization techniques by their names, here the tasks will give a short explanation and always cite their source, so that the user is never responsible for owning the proper reference.

This is actually not meant as a human-machine cooperation, since no artificial intelligence is intended. Instead, it is the designer behind the interaction tasks who is cooperating with the user. Neutrality needs not be sought then, as the feedback messages and the importance heuristic will embody the designer's point of view on how to improve a program. *Users Need Rationales*, as Carroll and Aaronson (1988) state, and a liberty for arguing is a decent mean to satisfy it.

2.4. The filtered notifications

In a simple technical design such as the first prototype shown further, or in Code Analyzer, a single pass on the code generates notifications to be displayed in the development environment – for simplicity, let us call messages the questions from the third task *ask* too. In EAVE and Code Analyzer, the suggestions generated all have critical importance, however we want to consider *every* possible feedback here. To avoid burdening the programmer with countless notifications, only a handful should be selected to be displayed at each build, hence the need to filter them. A simple solution proposed here is to evaluate a coefficient of importance for each message generated, then display the few highest rated ones. See the User study and future work for a technical description of the formula.

Besides, for the sake of transparency and to allow users to retrieve missed feedback, the full list of published messages must be kept available, that is all the notifications generated before they were filtered to keep a handful. The integration of this list in code editors is not covered in this paper, though.

3. Preparatory study

At this point, a series of interviews was necessary in order to evaluate the users' preferences regarding the contents of the messages, and ensure there would be no clear rejection of "improvements" to compilers. I chose to start with a simple algorithmic task. Being in familiar working conditions the interviewees could share their interrogations through a *think aloud,* and explain afterwards how they could optimise their code further. In need for participants with experience in programming, I selected KTH peers whom I knew had such experience. Each interview would be conducted on a platform the participant would be familiar with, be it his/her own laptop or a school desktop computer. I would sit next to the interviewee and give the instructions while he/she had the IDE in sight.

Three problems were written, covering a broad range of expertise, on three *distinct* typical goals in programming: *performance*, *security* and *maintainability*/*extensibility*. The problems would be given in any order, usually two in an interview, so as to fit in 40 minutes. Five interviews were carried over a month. This rather limited number of participants was fortunately mitigated by the range of their core fields: Numerical Analysis, Networking, Software Engineering, Cryptography, and Robotics.

It was first observed that *all* participants had interrogations or misconceptions regarding optimisations the compiler can perform, which was pounded by their average experience of 9 years in programming, and 2.5 years in their current language. This incomplete knowledge covered for example: where objects are stored in memory, the unrolling of loops, or the replacement of a multiplication by 2^n with a binary shift. One could argue that asking a quick working draft then its optimization induced the production of sub-efficient code in all three problems. This is however meant as a reflection of the IT industry, where delivery of working software under tight schedule is a core target[4]. The interviews showed that people perform hardly well at optimizing code, the interface should thus provide help to produce *efficient working code* at first draft.

It also appeared that proper optimization is not barely a matter of time. One *has to be* an expert in the specific field corresponding to the specific aspect targeted. This leads to projects centred around one aspect, the others becoming sub-efficient. A perfect example is the BSD family of operating systems: FreeBSD (performance), OpenBSD (security), NetBSD (portability)[5]. The interface should thus help to compensate where users lack expertise.

Furthermore, the participants often showed interest in the answers to questions they had previously been wondering. The existence of such unanswered interrogations which somehow *haunt* the users, calls for serendipitous information retrieval (De Bruijn & Spence, 2008). The interface should output several different messages at each execution and cite a source in each one, so as to expose the user to much information, that potentially answers a dormant interrogation. Moreover, for the same purpose the messages should be the shortest, and the number of sources limited to one.

Finally, when observing those using an IDE, I noticed they were all disabling the *Message of the Day* tooltip at startup. The reasons they gave were that much of the information displayed would document basic functionalities, the first few messages were not teaching anything, and they were neither contextual nor relevant. The first note motivated the requirement for technicality of the messages, that is they should always seem relevant, not worth being disabled, even if they would be quite complex. A source link could then provide the necessary details to users willing to follow it. As for the third note, it instructed to avoid citing what the compiler *can do* in general, in favour of informing what it *will do* on a particular line of code. Thereby, the feedback is closest to the context which triggered it.

4. Three prototypes

As advocated in Dow et al. (2010), I chose to design several prototypes in parallel, each embodying a distinct approach to the solution. This work does not include their iterations though, which are simply discussed in the next chapter.

4.1. First prototype: a stripped version for GCC

This prototype (Figure 2) emerged after an unsuccessful proposal for a Google Summer of Code for GCC[6], and corresponds to the interaction tasks *inform* and *alert*. Confronting the design of a compiler's feedback to the internal workings of GCC helped identify the difficulties, and overcome them with a simple technical scheme. HTML5, CSS3 and JavaScript were used for the neat look and interactivity they provide. Note that the sample feedback messages presented in all three prototypes are not meant to be true for any particular compiler; they simply look technical and precise.

[4] Refer to the first, third and seventh principles of the Agile development method at http://www.agilealliance.org/the-alliance/the-agile-manifesto/the-twelve-principles-of-agile-software/ (accessed 07.09.2012).
[5] For a brief history and comparison of the three major BSD systems, see http://www.freebsdworld.gr/freebsd/bsd-family-tree.html (accessed 07.09.2012).
[6] Available at http://www.google-melange.com/gsoc/proposal/review/google/gsoc2012/traf/2001 (accessed 13.09.2012).

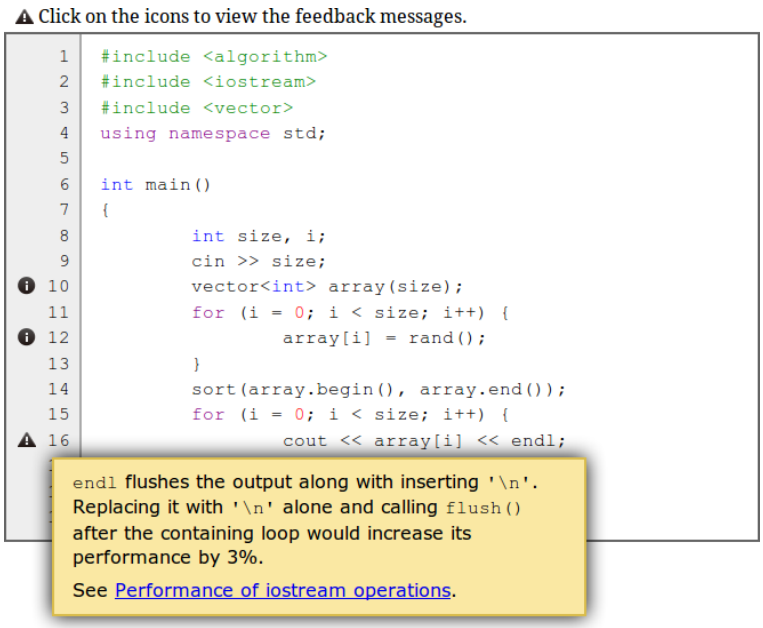

Figure 2 – Screen from the first prototype[7]

The original proposal involved generating the messages from every compilation unit in GCC, entangling the messages in its source code. The proposed approach here is to use files to store all possible messages, as pairs {*text*, *trigger*}, the latter being a condition on each instruction processed which enables the output of the corresponding *text* as a feedback. Though outside the scope of this thesis, the standardisation of a *trigger* syntax would allow messages to be written for several compilers, and be shared among users thanks to their storage in files.

The most important task along with designing the interface was to provide an exhaustive list of possible messages, to give a clearer idea of its usefulness. In order to manage the enumeration, I focused on the standard optimizations performed among most modern software:

- Register Allocation (explaining how faster an operation is performed in registers, telling whether for an inner loop or a function all automatic variables could be stored in registers or spilling happened, citing the allocation technique used, informing which conditions allow a data structure to be stored in registers)
- Strength Reduction (indicating when a multiplication by a loop index was carried with an addition, warning about the use of floating point functions on integers and propose alternatives, showing the replacement of multiplications with powers of 2 by binary shifts)
- Common Sub-expression Elimination (informing where an expression was found to be redundant and how the code was replaced, enumerating which operations are taken into account in CSE)
- Value Range Propagation (telling when a constant has been properly propagated, showing that the detected range of values of a variable leads to a performance gain, enumerating the types which can be propagated by the compiler, citing the Static Single Assignment technique)
- Branch Prediction (informing when a dead section was detected and will not be compiled, showing how branch probabilities translate into code and how performance improves)
- Functions Optimizations (telling when and why a particular function was inlined, informing about the compilation flags toggling inlining, warning when too many variables are passed to a function as the registers are limited, telling when Tail Recursion could be applied on a recursive function, describing how complex objects like classes are passed as arguments and returned)

[7] The first prototype is available online at http://www.csc.kth.se/~traf/thesis/proto1.html.

- Data Alignment (explaining why the size of a structure can be bigger than the sum of its fields' sizes, telling in which case padding was added inside a data structure, informing about the performance penalty when accessing unaligned data)
- Stack Layout (pointing which variables are stored on the stack, giving figures as to how performance increases with such storage, proposing buffer overflow protection techniques and informing which flags enable them, giving the typical stack size on the target system)
- Vectorisation / Parallelisation (indicating whether and why a loop could be vectorised on a SIMD-capable architecture, describing how to best control vectorisation through flags and tools, providing figures as to how the performance of a loop increased with the use of SIMD instructions, telling whether several similar operations could be packed in a single instruction, suggesting parallelisation libraries to execute simultaneous iterations on parallel threads)

I also focused on the various aspects of a language design (mainly C++) to enumerate a few more topics for feedback messages:

- Manipulation of files (explaining the difference in performance/cache use/security between the various input/output functions, pointing out risks for unsafe/unchecked input)
- Time management (warning about the year 2038 bug and discussing means to circumvent it)
- Strings and characters (providing a comparison between null-terminated and sized strings, explaining how the fast comparison and copying functions translate into code)
- Assertions (giving figures as to how enabling assertions impacts performance, telling whether assertions are used for value range analysis)
- Style and formatting (warning when a function is too big that it would not fit in a cache, telling which character set was detected for the source file and how strings are stored in the output, here the messages can greatly depend on the designer/community)
- Run-time checks (enumerating the list of available checks and the flags enabling them, informing when such checks have been inserted and their cost)
- Classes (showing when constructors and destructors are inlined, giving the number of system calls involved in the use of dynamically sized objects, describing the actual implementation of standard complex classes like bit-fields or hash tables, telling which operations will leave a certain iterator stable)
- Functions (introducing the overhead of a function call, telling which registers are saved/used)
- Data storage (explaining where in memory the standard instructs a particular static/automatic variable to be stored, informing where in memory constants are saved, introducing endianness and why one should care about it, suggesting faster initialization methods like `memset`)
- Floating point types (warning about the use of equality with such variables, describing the range of acceptable values including subnormal numbers and the expectable precisions)
- Operators (telling how certain ambiguous operations like integer division behave with negative operands, comparing the speed of an addition versus a multiplication on the target architecture, warning when an apparently small operation like a norm has a non-negligible cost, informing about the possibility of integer overflow and proposing various means to avoid it)
- Control flow structures (explaining how switch statements are converted into fast code)
- Exception handling (describing how this mechanism is translated into code, citing which types of exceptions are the most easily dealt with by the compiler)

These lists are certainly not exhaustive but already give a strong basis of feedback messages the interface could implement.

4.2. Second prototype: a communicative compiler

The second prototype (Figure 3) was designed to complement the feedback side with the possibility to query and discuss with the programmer, corresponding to the interaction tasks *ask* and *answer*. After a successful compilation, the second frame displays a set of spontaneously generated queries. As with the first prototype, these notifications will change each time a compilation is run. Answering them is never required; they will default to safe values.

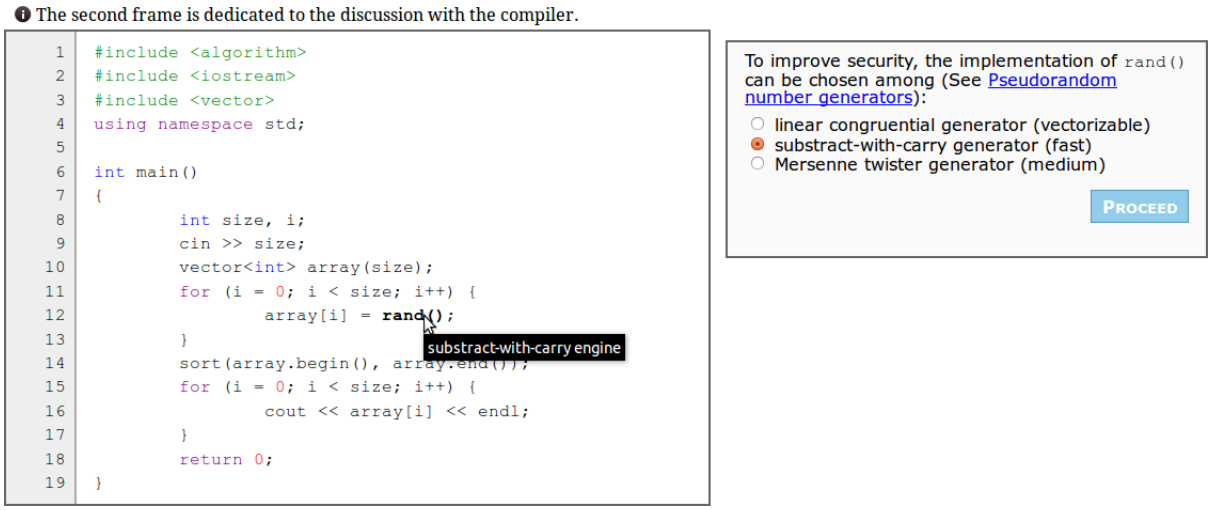

Figure 3 – Screen from the second prototype[8]

Technically, this prototype requires a compiler-specific semantic to express the answers to queries. As an example, clicking "Proceed" on the question in Figure 3 would add `#pragma rand substract_with_carry` before line 12, effectively giving this hint for the next build run.

Triggering these messages would then be similar to the first prototype. They would become triplets {*text*, *trigger*, *pragma*}, where *text* would be formatted to generate a query, and *pragma* would contain the text added to the source code after answering the question.

Enumerating the messages to include in such an interface is not as straightforward as for the first prototype. It requires seeking the aspects of a language semantics which are incomplete. For C++, I focused on the aspects which would benefit from the increased expressiveness without burdening the main semantic:

- Choosing the character encoding of a string or stream, which will influence functions such as `strlen` or `isspace`
- Setting the locale of the program, since the current C standard dedicates a single library to it
- Asserting that a certain variable will never overflow, which could enable certain optimizations
- Choosing the algorithm behind certain mathematical operations such as computing the inverse square root, while giving the precision of each
- Querying the expectable branching probabilities in a critical portion of code
- Asking whether to maintain an assertion in release mode when sample cases show its failure
- Gathering the properties of an variable read from a stream, to enable the use of faster routines
- Selecting the algorithm to sort an array, the default being usually Quick Sort
- Setting the precision for the storage of time or delays
- Selecting the implementation method to generate random numbers (in C++ a library is dedicated to it, though in C it is a single function)

[8] The second prototype is available online at http://www.csc.kth.se/~traf/thesis/proto2.html.

- Choosing the underlying storage of an array of bits (in C++ it is stored as a `bitset` though it sacrifices performance in comparison with an array of integers)
- Setting the properties of a container (the implemented directions of iteration, whether the size often increases, where items are appended, whether stable iterators are required)
- Asking for the implementation method of a binary tree or a hash table
- Proposing the stack protection method against memory corruption including buffer overflows

These semantics being optional, they must not have critical importance on the program. They will rather be set to tune its various aspects, when the program was already proven to work properly. As with the previous prototype, this list is certainly not exhaustive, but gives an insight for the usefulness of the *querying* improvement.

4.3. Third prototype: a far-fetched alternative

This prototype (Figure 4) goes one step further in the coupling between the compiler and its interface. It uses a second frame to graphically represent the compiler's understanding of the various elements found in the code. It was mostly intended as a place for open suggestions from the testers, and is not to be matched with the interaction tasks previously mentioned.

Having noticed in the interviews that users had a will to do good despite their refusal of an embodied interface, I chose to depict the compiler as a living system. The interaction then consists in the user helping the compiler understand the code, a simple colour scheme being used to inform how an element is apprehended. The two frames are representing the code as text, however the right one *should* evolve towards a more suitable representation, such as a coloured dataflow diagram.

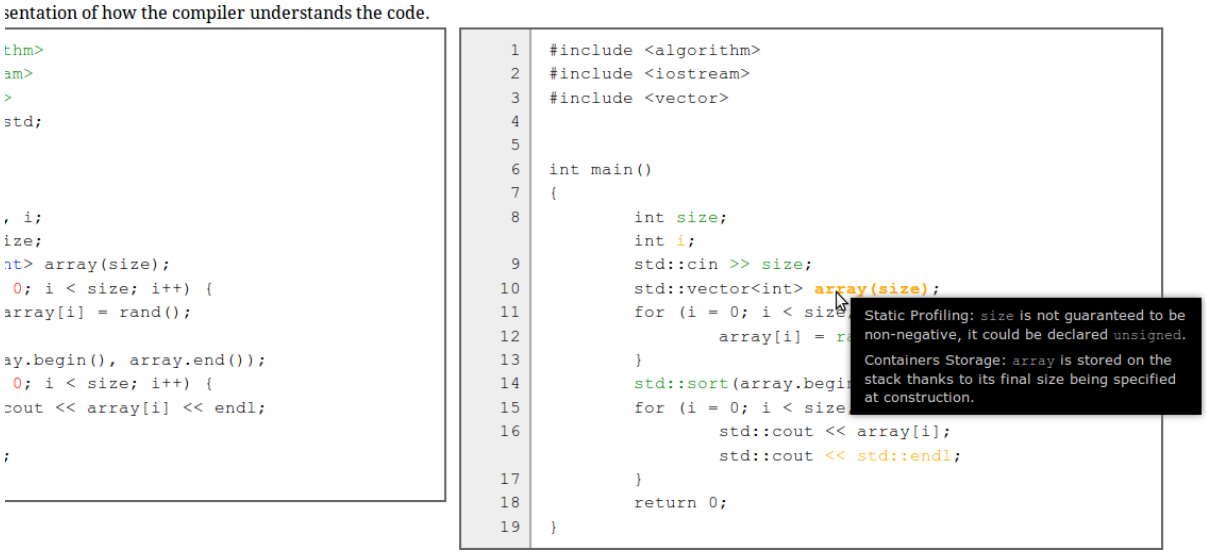

Figure 4 – Screen from the third prototype[9]

5. User study and future work

A new series of interviews was scheduled to ensure the goals set after the first one had been met, and estimate how users would consider the value added. Five interviews of 30 minutes each were conducted over two weeks, with the requirement that the participants had not participated in the previous interviews, had had a previous experience with an IDE, and could understand basic C++. After testing the three prototypes in order, the interviewee would choose his/her favourite and explain it, then answer a few additional questions about the use of *personae* (see the next dedicated section).

[9] The third prototype is available online at http://www.csc.kth.se/~traf/thesis/proto3.html.

Both the first and third prototype were praised, the former for its technical insights, and the latter as a quick overview of the compiler's job. As with the preparatory study, few participants were initially showing interest or added value in a feedback from the compiler. I had to pursue the description to the personae, until the concept of compiler feedback became clear and coherent. Then they all agreed that they would not disable the feedback like a *Message of the Day*, which was my main concern. Some even expressed they were actually looking forward to seeing a working release in the future. Surprisingly though, very few interviewees understood the third prototype's colour scheme at first glimpse; further iterations should probably explore accentuating the areas requiring attention inside the very tooltips.

On the downside, the second prototype was generally little understood. This might have been influenced by the unusual situation of being queried by the compiler. However, in my opinion it was the presentation as a separate frame which made it difficult to spot the context at hand, that is which part of the program the question was dealing with. The integration of queries along source lines as in the first prototype then remains to be tested for future iterations.

5.1. The personae

This idea appeared with the possibility to store messages in files, as discussed in the first and second prototypes. Provided a *trigger* syntax is defined, each notification can be stored aside the compiler, under a triplet {*text*, *trigger*, [*pragma*]}. Sets of notifications can then be stored as files forming categories of similarly related items. The addition of a field along every message of the first prototype could further allow the programmer to be aware of the category at hand, and increase or decrease its further occurrences, in order to receive the most interesting feedbacks.

Categories form the default set of messages shipped with the compiler. To extend and customise this set, users could create their own files and share them. The *persona* here is the idea to bind an author to a file with notifications. Knowing who wrote a certain suggestion could give value to it and mitigate the effect of a poor feedback, particularly if the author is known for being a good programmer. Here, a simple and recommended way to store the category and author's names is though the file's name.

In practice most testers were very receptive to it, with different intents. One tester did not care about the author's name, as long as he/she was a specialist. Another one conceived the sharing of files inside teams of developers, in companies. A last one considered contributing in online communities of developers rather than friends.

5.2. A few rules for composing the messages

During the tests, feedback was sought for the relevancy and clarity of the messages. While the testers were often puzzled with the feedback's technicality, they were very fine with it. Indeed, two actually argued that they were used to this situation. The links to references were intended to balance this complexity, and in practice were praised by all interviewees. The quality of redaction had a great influence on the participants' reception of each message, though. The first query in the second prototype, for example, was systematically deemed too complex, and I always had to explain it. This difficulty motivated further the addition of personae, to let programmers choose a good teacher, and sketch a set of rules to help the redaction of further messages:

- *Technicality*: The feedback should rather be too technical than not enough, and provide a substantial benefit which will be highlighted.
- *Context*: Indicate what the compiler *will do* for a particular line/object rather than what it *can do* in general.
- *Referencing*: Cite one and only one source giving details for the corresponding feedback.
- *Neutrality*: Balance the amount of positive and negative feedback, that is when a line was well understood or when it needs tuning.
- *Clarity*: To be read and understood quickly, each message should receive careful attention and go straight to the point.

5.3. Transparency is crucial

As advocated in Sinha & Swearingen (2002), transparency has been a key design choice along this work. Providing a reference link along each feedback, targeting the programmer's knowledge rather than hacking tips, binding the author to the messages, defining a syntax for notifications and storing them in text files directly accessible to the programmer, were all choices motivated with transparency in mind. It should be noted here that transparency is preferred over translucency – selecting what is to be shown – since no *thought* restriction on the information given was intended.

In my opinion, transparency is the key means to show and insist on the absence of artificial intelligence, or "smart assistant", to govern the suggestions. Giving the qualifier *smart* to the robot could make the users feel it is asserted to be smarter than them. Carroll and Aaronson (1988) bring out many receptiveness issues from interacting with such an agent, which transparency would greatly mitigate. Indeed, with access to the database of possible messages, and knowledge of who lies behind them, users are aware of the bounds of the compiler's intelligence, and will not expect more than it could actually help.

5.4. A formula to evaluate the importance of each message

As hinted in the differentiators, taking into account the huge range of possible feedback messages will require the computation of an importance factor along with every message generated, in order to select a handful to display. For the purpose of being exposed to the programmer and implemented easily, the formula constructed here aims at simplicity.

Let us note f_c a constant criticality factor for the message, n the number of its previous occurrences, f_p a preference factor assigned to its category, and m the number of previous messages displayed on the same line. A simple recommended formula here would be the average of f_c, 2^{-n}, f_p, and 2^{-m}, provided all are bounded by [0;1]. The strength of such a formula is the simplicity to graphically represent an average. As a drawback, it does not allow to completely disable one category, though this is actually possible by simply deleting the corresponding file. Also, the factors averaged might need to receive an additional scale, which estimation is left to implementation.

Note that reinitialisation of the heuristic is to be taken into account. The compiler might be reinstalled often, possibly clearing the memory of previously displayed messages (n and m) and preferences (f_p). The more files/categories are stored, the more time will be needed to reach their previous f_p values (considering the programmer can only increase/decrease this factor on every message received). The number of categories should thus be limited to a dozen on average.

6. Discussion and concluding words

This project started with the idea to have the compiler return performance-helper messages, much as in the first prototype. The expected design was then pretty clear, though the content of the messages remained to be defined. However, most of the time was actually spent on communication tasks. Indeed, the project suffered from the difficulty to clearly state what was intended by a *feedback*, because this word can be interpreted quite at will. Moreover, it evolved vastly to seek a coherent system which could be obviously differentiated from mere compiler warnings, and the dreaded paperclip assistant from Microsoft. Designing the prototypes with the precision of HTML and CSS helped dramatically to show *exactly* what was intended. An actual challenge during this Degree Thesis was the management of the two rounds of interviews. Though in this paper they occupy a marginal space, in practice a sheer amount of time was dedicated to them, especially for preparing the interview plans and the algorithmic tasks.

A few points were left aside during this work, either by lack of time or because they were a matter of debate. Among them, the initialisation of the system should be mentioned. Introducing the feedback is important, so that users do not disable it instinctively like a *Message of the day*. An example for an introductory text could be: *This compiler can output feedback messages telling how it understands the code, as well as technical suggestions. The list of feedback messages it can generate is contained in [folder], and can be extended by adding .cfb files downloaded from trusted authors*.

Also, this paper does not cover how to identify the level of knowledge of the users. Receiving too complex/simple feedback might eventually annoy them. While they can choose and download the notifications' files to add to the compiler, the initial set of categories could be specifically tailored to each one's knowledge, by estimating it with a question along the introductory text, for example.

One last issue which might eventually arise with the possibility to share authored feedback files is the lack of secure signing. If the author's name is stored in the name of the file as suggested in this paper, nothing prevents it to be overwritten, or a wrong set of messages be imputed to the same author. The rationale behind this choice is similar to the availability of coding guidelines on the Web: it is the user's responsibility to fetch the file at the source he/she trusts.

As shown in the tests, the introduction of a feedback from the compiler was well received, either indirectly for the "big picture" overview it would provide, or for its relevant technical insights. Using this interface does require very little learning, if any, which is in my opinion a cornerstone of this work. As for the direction to give to the prospective future iterations of the prototypes, since both the first and third were deemed promising the focus should be laid on implementing the common basis, namely generating the feedback messages. An option could then be available to choose *how* to display them. Indeed, this choice might depend on the progress of the coding project. At the early stages when raw code is written, a few technical messages targeting the most critical aspects would be needed, as in the first prototype. Later in the process when these fragments are assembled, a broader overview like the third prototype would become useful. Giving these tools to the programmer would then make optimisation more accessible, outside the sphere of expert programmers.

7. References

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Prentice Hall.

Bose, P. (1988). Interactive program improvement via EAVE: an expert adviser for vectorization. *Proceedings of the 2nd international conference on Supercomputing - ICS '88* (pp. 119–130). New York, New York, USA: ACM Press. doi:10.1145/55364.55376

Carroll, J., & Aaronson, A. (1988). Learning by doing with simulated intelligent help. *Communications of the ACM*, *31*(9), 1064–1079. doi:10.1145/48529.48531

De Bruijn, O., & Spence, R. (2008). A new framework for theory-based interaction design applied to serendipitous information retrieval. *Transactions on Computer-Human Interaction*, *15*(5). doi:10.1145/1352782.1352787.

Dow, S. P., Glassco, A., Kass, J., Schwarz, M., Schwartz, D. L., & Klemmer, S. R. (2010). Parallel prototyping leads to better design results, more divergence, and increased self-efficacy. *ACM Transactions on Computer-Human Interaction*, *17*(4), 1–24. doi:10.1145/1879831.1879836

Kreeger, M. N. (2009). Security testing. *ACM SIGCSE Bulletin*, *41*(2), 99. doi:10.1145/1595453.1595484

Lethbridge, T. C. (2000). What knowledge is important to a software professional? *Computer*, *33*(5), 44–50. doi:10.1109/2.841783

MathWorks. (n.d.). Using the MATLAB Code Analyzer Report. Retrieved September 7, 2012, from http://www.mathworks.se/help/techdoc/matlab_env/f9-11863.html

Saffer, D. (2009). *Designing for Interaction: Creating Innovative Applications and Devices* (2nd ed.). New Riders Press.

Sinha, R., & Swearingen, K. (2002). The role of transparency in recommender systems. *CHI '02 extended abstracts on Human factors in computer systems - CHI '02*, 830. doi:10.1145/506558.506619

Zhao, W., Jones, D. L., Cai, B., Whalley, D., Bailey, M. W., van Engelen, R., Yuan, X., et al. (2002). VISTA. *ACM SIGPLAN Notices*, *37*(7), 155. doi:10.1145/566225.513857