Abstract
Research on WCET analysis and time-predictable systems faces difficulties when it comes to the evaluation of new ideas. Even conceptually simple improvements need a large and complex infrastructure to validate their usefulness. This infrastructure comprises suitable benchmark suites, program analysis tools, compilers for possibly non-standard instruction set architectures, realistic simulators and/or hardware and, last but not least, a WCET analysis tool supporting the architecture and its experimental extensions. In order to mitigate this problem, we argue for an open and extensible evaluation platform that is centered around a mainstream open-source compiler framework and open computer architectures. We investigate a possible candidate built around the LLVM compiler framework and an open-source processor, and present a case study tying together two existing tools from academia and industry. The experiences so far suggest that such a platform simplifies the integration of different research efforts, and thus has a positive impact on the quality of research and teaching in the WCET analysis community.

1. Introduction
Research on worst-case execution time (WCET) analysis and on building time-predictable systems needs to be evaluated by experiments. For both lines of research, the effectiveness of an approach can only be judged by comparing experimental results of different techniques in similar settings. Any particular technique performs poorly in certain scenarios. Experiments with realistic benchmarks are thus needed to show whether these scenarios are indeed problematic, or if they are merely theoretic issues. Evaluating new strategies is only economically feasible, however, if the new techniques can be easily plugged into an existing framework.

Therefore, we argue that research on time-predictable architectures and WCET analysis needs an open framework facilitating the reproducible evaluation of ideas in realistic settings and a fair comparison of different approaches. The central question when discussing common settings for different researchers and projects is of course how to maximize flexibility while ensuring a common baseline for evaluation and providing good support for implementing prototypes within a reasonable time frame. Concerning flexibility, it should be possible to replace most components by alternative implementations, also supporting proprietary tools and conceptually different approaches.

We provide arguments that the platform should be built around two central, open components. First, an extensible, mature compiler framework, together with a small set of community-maintained plug-ins for timing analysis. This compiler framework should use a standardized, modern intermediate language representation, give fine-grained control over code optimizations and enable the construction of exact mappings between intermediate representations and generated machine code. Secondly, the framework should include one or more open computer architecture designs, which are well supported by the compiler. Amongst other things, the target platform should make it easy to extend the instruction set architecture (ISA) and add new hardware features, as well as perform measurements to validate analyses.

Our prototype system is built using the open-source compiler framework LLVM [12] and the open-source processor LEON3 running on a FPGA. We do not aim to push one particular solution at this point, and both LLVM and the processor are nothing but possible candidates. Instead, we try to illustrate the key ingredients, which allow one to build an attractive platform for research institutes and support the fast and seamless integration of additional tools. As a proof of concept, we developed a showcase demonstrating the integration of two existing tools into the platform. We use the Swedish Execution Time Tool (SWEET) [8] to extract flow facts. For this purpose, we developed an ALF [7] back-end for the LLVM intermediate representation. To obtain the WCET, we use the commercial tool aiT [4] for the LEON3 processor. This is supported by a generator for mappings between the intermediate and machine code representation of LLVM.

1 Institute of Computer Engineering, Vienna University of Technology, Austria
2 http://www.gaisler.com/
Related work  The ALL-TIMES project [9] aimed at the integration of available timing tools into a single framework, and thus shared some of the goals of the proposed platform. To this end, several open interface specifications where defined within ALL-TIMES, to facilitate the communication between different tools. The language ALF, intended as general purpose input language for flow analyses, is of particular interest here. It seems to be a suitable candidate to represent imperative programs from a variety of different sources. However, as the ALF representation is only built a posteriori, the connection between source language and the various binary formats is missing. Performing program analyses on binaries is possible, but a considerable amount of information is lost in the translation process. Additionally, whenever the ISA changes, it is necessary to adapt both the compiler back-end and the ALF representation, duplicating the work of computer architecture researchers.

In Figure 1, the current situation regarding the integration of the most important components influencing or taking part in WCET analysis are shown. Black-box compilers have well-known drawbacks, as valuable information on control flow, types and memory references is lost during compilation. As the details of the compilation process are hidden, relating the source language and the compiled binaries is only possible to a limited extend. Consequently, the modification of existing compilers for WCET analysis has been investigated. The WCC compiler [3] uses a WCET analysis tool (aiT) to decide which optimizations should be performed in order to minimize the WCET. It also supports the translation of source-code level flow facts, and integrates a loop bound analyzer. It is, however, intended for performing focused compiler research, and seems tailored towards a single architecture. Additionally, it has less widespread support than mainstream compiler frameworks like LLVM, and is not easily modifiable by other research groups.

A mainstream open-source compiler that has already been used for WCET centered research (e.g., in [11]) is GCC.3 It has been successfully adapted to various input languages and target platforms, and thus would be a possible candidate for a compiler framework around which the proposed platform can be built. The major drawback of GCC is that it is lacking a well-documented intermediate language, on which analysis tools can rely on. Moreover, personal experience has led us to the conclusion that LLVM is more modular, and easier to understand and work with.

The Chronos tool [14] aims at providing a platform for (low-level) WCET analysis research. It is open source under the GPL license, and analyzes binaries for the PISA variant of SimpleScalar [1]. This well-known instruction set simulator can be configured to simulate a wide variety of architectural features. The proposed project differs from Chronos in two aspects: First, we attribute a central role to the compiler and its intermediate representation. Second, we are also interested in the analysis of hardware-implemented computer architectures, which poses additional challenges compared to higher-level instruction set simulators. Nevertheless, given a suitable LLVM back-end for PISA, or adding support for the more recent ALPHA variant of SimpleScalar to Chronos, its integration into our proposed platform seems plausible. As Chronos is primarily a low-level timing analysis tool, it would indeed benefit from this integration, being able to use the results of program analysis tools.

3http://gcc.gnu.org/
For safety-critical embedded systems, a formally verified compiler is being developed within the CompCert project [13]. However, due to the complexity of formal verification, we believe that modifying and extending CompCert for evaluation purposes requires too much effort.

Outline The rest of the paper is organized as follows. The requirements and expected benefits of an open timing analysis platform are presented in Section 2. Section 3 discusses modular compiler frameworks, exemplified by LLVM, and how they can support WCET analysis. The benefits of open target platforms, and their key role in the framework are outlined in Section 4. In Section 5 we present a prototypical instantiation of the timing analysis platform, integrating SWEET for program analysis, and the commercial tool aiT for calculating a WCET bound. Finally, Section 6 concludes the paper.

2. The Open Timing Analysis Platform

The timing analysis platform should support both WCET analysis research and research on time-predictable systems. The former comprises new techniques for program analysis and the low-level analysis of existing or novel hardware features. The latter includes all constructive approaches, both at the software and hardware level, as well as source and machine code generation or transformation. This is a natural requirement, as the design of systems and the requirement to analyze them are tightly coupled in the field of time-critical systems.

The platform’s main goal is to support the reuse of established techniques at different levels, so that it becomes possible to perform a realistic and reproducible evaluation, while only focussing on the core subject. Consequently, it should be possible to replace any of the following components, while reusing standard implementations for the others:

**Benchmarks** The addition of new benchmarks, such as those collected within the WCET benchmark effort [6], should only require little effort. To this end, compiling the benchmarks for the desired target architecture, providing the input to any particular tool and reusing or evaluating the tool’s results needs to be automated as far as possible.
Portable benchmarks in an lower level intermediate representation, including precise flow facts, should be available, thus lowering the barrier for evaluating low-level analyses.

**Program Analysis** Tools performing program analysis often operate on high-level source code or an intermediate language representation, such as ALF or LLVM bitcode. It should be possible to evaluate tools for high-level WCET analysis by reusing existing target platforms along with suitable low-level analysis tools. Additionally, we would like to reuse analyses not specifically developed within the WCET analysis context, such as polyhedral analyses or bounded model checkers for test case generation.

**Code Generation and Compilation** Research in this category includes new approaches to model-based code generation and analysis-driven compiler optimization. A well-designed, modular compiler infrastructure with support for timing analysis would surely lower the burden for implementing and evaluating compiler improvements for different architectures. Similar to the program analysis case, the integration of existing low-level analysis and measurement tools should only require little effort.

**Target Platform Extensions** Extensions of the target platform may require extensions of the ISA, the hardware, and the WCET analysis tools. To minimize the effort needed in this respect, one would like to build on an extensible compiler framework, and an existing reference architecture together with customizable low-level WCET analysis tools.

**Low-Level Analyses** Low-Level Analyses should be able to reuse flow facts specified along with the benchmark, or obtained using a standard set of high-level analyses. Furthermore, the platform should support the reconstruction of control-flow graphs, and the evaluation and validation by means of measurements.

Considering these requirements, we propose the design illustrated in Figure 2. The first key component is an open and modular compiler framework with a mature, analysis-friendly intermediate representation. The major advantage of a well-defined intermediate representation is that one can effectively separate high-level and low-level WCET analyses. It is still possible to use other intermediate representations such as ALF, by implementing a suitable translation as LLVM backend. We also appreciate the efforts to extend the set of WCET benchmarks available to the public [6], which are central to meaningful evaluations. The second essential component to simplify experiments is an open target platform, with an open, extensible hardware design. This enables the joint development of the hardware and the low-level analysis [10], ideally leading to hardware designs with good worst-case performance. In the next two sections, these two components are discussed in more detail.

### 3. Open Compiler Frameworks for Timing Analysis

Writing production-quality compilers requires huge development efforts. To amortize the initial development costs, compilers are usually long-lived software products, which are able to cope with new developments in language and hardware design, that were not foreseen when the compiler was designed. The compiler produces binaries which are part of the input to WCET analysis, and thus has a strong influence on worst-case run-times and analysis results. Compilers therefore play a crucial role in timing analysis related research, even if the compiler itself is not in the focus of the discussion.

The solution for both the complexity and the long life cycle of a compiler is a modular design. Usually, a front-end translates a programming language to an internal representation. This internal representation is the basis for all further transformations in the compiler, which include optimizations and possibly also transformations to lower-level internal representations. Within this “middle-end”, a modular design enables the replacement of individual passes, which makes it possible for the compiler to adapt to the progress in compiler research. Finally, the back-end translates the internal representation to machine code or assembly language.

An example for a modern compiler framework is LLVM [12], which supports multiple input languages and a variety of targets, including well-known architectures such as x86, ARM, SPARC, PowerPC, or MIPS, but also less widespread ones such as Blackfin or XCore. One key concept of LLVM is its intermediate language (LLVM bitcode), which is a low-level but platform-independent language, designed for program analyses and compiler optimizations. In particular, the language simplifies analyses as all instructions are explicitly typed, control flow is explicit, and so is data-flow in static single assignment form. Many optimizations do not rely on the details of a particular instruction set, and can thus be performed on the intermediate language. Therefore, flow fact analyses can be performed on platform independent, but optimized code, eliminating the difficult problem of maintaining the results during optimizations, which modify control flow structures.
Another important characteristic of LLVM is its modular design. There is a clean separation of front-end (converting the source language to the LLVM instruction set), optimizations (on the LLVM instruction set) and the back-end (platform-dependent optimizations and code generation). The independence of the front-end and the simplicity of the LLVM instruction set enables the compilation of experimental languages directly to the intermediate language, avoiding the problems when the C language is not a good fit (lacking e.g. tail calls). Both the intermediate language and the machine code are analyzed and transformed in a series of passes. This allows one to control precisely which optimizations take place during compilation. This is important if one wants to prohibit optimizations altering the structure of control flow graphs.

Adding new analyses and transformations is well supported in LLVM, many can even be realized as plug-ins to an existing LLVM compiler instance. This lowers the burden of experimenting with new optimizations or new computer architectures considerably. Still, the maintenance of extensions to third-party software is a delicate task. In this respect, it is helpful to work with software that is open-source, has an open development process with public mailing lists and source code repositories as well as a well-defined release procedure.4

Timing Analysis Support in LLVM We are interested in a small and maintainable set of extensions to the LLVM compiler, which are essential for WCET analysis support. Most importantly, as illustrated in Figure 2, the compiler needs to maintain a mapping between the optimized intermediate representation and the machine code representation. LLVM already keeps a partial map between machine code basic blocks and basic blocks in the LLVM intermediate representation (IR). Unfortunately, the mapping between blocks in the IR and the machine code is currently not always one-to-one. Some instructions in the IR need to be realized by several machine code basic blocks for certain instruction sets. Although identifying the correct machine code basic block is certainly feasible, removing the offending instruction in the intermediate representation is preferable. More problematic are transformations which turn several basic blocks in the IR into one basic block at the machine code level. The most prominent example is if-conversion, which is difficult to implement as a high-level optimization, as the IR does not support all forms of predication. This transformation needs to be supported, as flow facts referring to if-converted basic blocks may no longer be valid. However, as the if-conversion pass is explicitly represented in the LLVM back-end, it is possible to either exclude this pass, or replace it by a custom pass which supports the modification of flow facts.

In addition to flow information, we also would like to preserve a detailed mapping for value information. This is more difficult, as there is not always a direct correspondence (e.g., a 64-bit value represented as two 32-bit values). At the moment, LLVM already keeps high-level information on memory operands in the machine code representation, which can be used to derive address ranges for the analysis of memory accesses.

For high-level WCET analyses, we think that LLVM bitcode is a suitable representation, with similar properties as ALF. As we want to reuse analyses that process ALF as input, we wrote a corresponding LLVM back-end. In addition, it is possible to reuse all analyses developed to run on LLVM bitcode. Many “simple” loop bounds, for example, are already known to the loop analysis pass integrated in LLVM mainline.5

To be fair, not all information present in a high-level programming language (or a higher level assembler such as Java Bytecode) are directly available in LLVM. For experimental domain specific programming languages, or code generators targeting hard real-time systems it thus might be useful to add additional information to the assembler. To this end, the LLVM IR has explicit support for custom meta data, which is not limited to debugging information. Currently there is no explicit support for transforming meta data during optimization passes, however. Consequently the transformation of source-code level flow facts will require additional implementation efforts.

A final important consideration is the mapping between the final executable to the machine code representation in the compiler, when using an external assemblers and/or linker. One advantage of LLVM in this respect is that linking and global optimizations can be done on the bitcode level, which avoids the need for most global optimizations on the machine code. Another problem often encountered in timing analysis is the analysis of libraries only available in machine code form (e.g., for implementing integer division in software). If it is possible to reconstruct the basic control flow, platform-dependent assembler code should also be represented using LLVM bitcode. This way, existing analysis tools operating

4http://llvm.org/docs/HowToReleaseLLVM.html
5Analyzing the Mälardalen benchmarks (see Section 5), the relatively simple scalar evolution analysis integrated in LLVM identified 113 constant loop bounds (55.7% of all loops) and 32 symbolic loop bounds, identifying 71.4% of all loop bounds and all loop bounds in 45% of the programs in the benchmark suite
on the intermediate representation can be used even if some parts of the program are only available in binary form.

4. Target Platforms

FPGA technology has enlarged the target audience for computer architecture research. Soft cores, which are synthesized by the system designer, enable easy customization of processor cores. Usual customization options include the configuration of cache sizes and replacement policies or the addition of custom functional cores. Depending on the licensing policy, even changing the instruction set or the pipeline organization may be allowed.

While these features simplify the development of application-specific hardware platforms, they also provide opportunities for WCET analysis. Usual processors that are manufactured as ASICs necessarily remain a black box with regard to their precise behavior. If the observed behavior does not match the documentation, the actual behavior must be reverse-engineered in order to model it accurately. Also, vendors only provide a limited number of configurations for COTS processors. Configurations that are unlikely to attract a significant number of customers are unlikely to be manufactured at all.

In contrast, an open hardware platform provides three major benefits for WCET analysis research:

- The actual hardware specification can be simulated; its behavior can be observed in detail and validated against the hardware model of the WCET analysis tool.
- Problematic features (e.g., branch prediction) can be omitted if they result in unacceptably high overestimations in the WCET analysis.
- Features that are potentially beneficial for WCET analysis can be evaluated with regard to both effectiveness and hardware cost, even if they are unlikely to be attractive for markets that emphasize average-case performance.

In our opinion, an architecture with strong support from the WCET community is more likely to leave the realm of academia and enter the industrial market than fragmented approaches from individual research groups. As an open hardware platform simplifies collaboration and provides the possibility to integrate the ideas from different research groups, we believe that only such a platform will be able to receive this strong support.

While it would have been possible to integrate a software simulator into our framework, we chose a platform that runs on an FPGA for two reasons: First, we wanted to show that our framework is capable of spanning the full range of WCET centered research, from high-level analysis down to hardware development. Second, the timing results of software simulators are often not as precise as one would want them to be. For example, the FAQ for SimpleScalar states that “It is very difficult to produce the same exact execution each time a program executes on the SimpleScalar simulators.”

There are currently several approaches to create time-predictable computer architectures from scratch [2, 15]. However, these projects still need a considerable amount of work before being usable as research platforms for other research groups. In contrast, architectures that were not explicitly designed for WCET analysability, such as the LEON3 processor, seem to be mature enough to serve as a starting point for exploring time-predictable computer architectures. As LEON3 is well supported and there also exists a considerable amount of infrastructure built around it, we think that it is a good choice for a hardware platform for WCET analysis research. However, if time-predictable computer architectures gain momentum, they may become equally good or even better choices.

The decision for LEON3 in favor of other soft core processors as target platform was a rather pragmatic one, as it is supported by both LLVM and aiT and already has been ported to the FPGA boards we had at hand. However, while its pipeline is relatively simple, LEON3 provides support for scratch-pad memory and the possibility to configure the size, associativity and replacement strategy of its caches. This make LEON3 an interesting target for research on the memory subsystem. As the proposed framework simplifies the integration of new architectures, switching to a different target platform can be done with minimal effort. For popular existing architectures, it is possible to reuse existing compiler

---

6Compare Fisher, Faraboschi and Young [5]: “In fact, ’cycle-accurate’ has become so diluted it is sometimes used to refer to simulators that attempt to report cycle counts, regardless of the level of accuracy. […] Measuring the accuracy of a simulation is at best considered an academic discipline and at worst never performed.”

7http://www.simplescalar.com/faq.html#Q2
back-ends and low-level WCET analyses. For new architectures, only the compiler back-end and the low-level WCET analysis need to be retargeted, while existing high-level analyses and transformations can be reused.

5. Tool Integration and Evaluation

To demonstrate the benefits of an open timing analysis platform, we integrated two existing tools into the platform prototype for the LEON3 processor. The overall architecture is illustrated in Figure 3. The basic tool chain used for the LEON3 processor consists of the \texttt{llvm-gcc} C front-end, the LLVM compiler (including its SPARC back-end), the LEON3 linker, and the LEON3 processor running on a FPGA board.

5.1. Tool Integration

To support the integration of timing analysis tools, we developed two small components built upon the LLVM infrastructure. The \texttt{ALFBackend} pass creates an ALF specification for the module representing the program. It is sufficient to consider a single module, as LLVM is capable of linking modules at the IR level. We encode the names of LLVM basic blocks into generated ALF labels, therefore it is trivial to establish a correspondence between these two representations. The goal of the \texttt{ExtractMappings} component is to output all the information available to link together the machine code and the bitcode representation. This includes the mapping between machine code and bitcode basic blocks and a mapping between memory address operands and the corresponding memory locations in the intermediate representation.

The first tool we integrated is SWEET. It takes the ALF representation of the program as input, and produces output suitable to be used with aiT. The mapping file is used to patch the output of SWEET, replacing program locations in the IR with addresses of machine code basic blocks. For the WCET analysis itself we use the commercial tool aiT [4]. We provide it with the precise single-path flow facts from SWEET, and so no manual intervention was needed in order to calculate the WCET in our experiments.

5.2. Prototype

To evaluate the prototype described above, we validated the low-level timing analysis of a simple, cache-less LEON3 design. The goal of this evaluation is not to assess the quality of one particular analysis. Instead, we want to provide evidence that using the platform together with other third-party tools, the validation requires very little effort.
Table 1: Processing steps involved in the evaluation

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Involved Tools</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.c</td>
<td>.bc</td>
<td>llvm-gcc, llvm</td>
<td>Compile the benchmarks to LLVM bitcode, run LLVM optimizer on bitcode</td>
</tr>
<tr>
<td>.bc</td>
<td>.alf</td>
<td>llvm + ALFBackend</td>
<td>Generate ALF file, using equivalent label names for basic blocks</td>
</tr>
<tr>
<td>.bc</td>
<td>.s.map</td>
<td>llvm + ExtractMappings</td>
<td>Generate the assembler file for the benchmark; generate mappings between basic blocks in bitcode and machine code</td>
</tr>
<tr>
<td>.s.map</td>
<td>.elf.map</td>
<td>gcc + scripts</td>
<td>Use the LEON3 gcc to generate the binary; extract address of machine code basic blocks from binary</td>
</tr>
<tr>
<td>.alf.map</td>
<td>.ais</td>
<td>SWEET + scripts</td>
<td>Generate flow facts by abstract execution of the program</td>
</tr>
<tr>
<td>.elf</td>
<td>Measurements</td>
<td>LEON3 debug interface</td>
<td>Run a script interacting with the LEON3 debug interface to load the binary, and perform accurate timing measurements using the processors instruction trace buffer facilities</td>
</tr>
<tr>
<td>.elf.ais</td>
<td>WCET bound</td>
<td>aiT</td>
<td>Provide the binary, the machine settings file and the annotations to aiT, which then fully automatically calculates a safe WCET bound</td>
</tr>
</tbody>
</table>

The idea of the validation is to take a set of benchmarks with given input data, such that every execution results in the same execution trace. If there is no indeterministic behavior, the SWEET tool is able to generate very precise flow facts. For example, the exact execution frequency for each basic block will be derived by the tool. This flow facts are used as an input for the low-level timing analysis tool. This way, we expect that the analyzed WCET is close to the maximum observed execution time. If the WCET is below one measured execution time, the validation fails. A large gap between these two metrics indicates that the exact timing behavior of the target architecture is difficult to analyze, or requires more flow information than usually available.

The starting point for the evaluation are those Mälardalen benchmarks\(^8\) which do not use floating point arithmetic (19 out of 26), and the LEON3 design running on an Altera DE2-70 evaluation board. Additionally, we created a machine settings file for aiT, which specifies the timing characteristics of the platform. For each benchmark we analyze the main function, with only one possible path executed at runtime.

The steps involved in the automated analysis are illustrated in Table 1. All of the described tasks are fully automated, except the analysis with aiT, which runs on a different machine and requires some simple GUI interactions for setting up the analysis. Running the benchmarks automatically, we obtained reasonable results for all but three benchmarks. The problematic benchmarks apparently triggered bugs in the third-party tools used, which we did not investigate further. The difference between measured execution time and observed execution time, relative to the observed execution time was collected for each benchmark. This ratio ranged between 3% and 21%, which seems reasonable for the target platform analyzed.

6. Conclusion

To evaluate new techniques in WCET-analysis research or to assess the strengths and weaknesses of new approaches for building time-predictable real-time computer systems, one needs a considerable infrastructure, consisting of a compiler, a target hardware platform, and tool support for timing analysis for the available hardware/software system. As it is very labor and cost intensive to establish such a framework on a per-case basis, i.e., whenever one wants to evaluate some new ideas in any of these areas, we propose an open, freely available timing analysis platform that could serve as a common basis for testing new technologies with low efforts.

In order to allow for an easy adaption, configuration, and modification of components to fulfil the user’s needs, the platform is based on open-source hardware and software. It is built around a core consisting of the LLVM compiler framework for code generation and analysis and the LEON3 processor hardware platform that can be configured and modified to assess processor platforms of manifold configurations. Further we have integrated two representative tools, the academic tool SWEET for program-flow analysis and AbsInt’s tool aiT for static WCET analysis to demonstrate the ease of adapting the environment. We foresee that research issues like the following could be explored with reasonable efforts by making

\(^8\)http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
use of this platform:

- research on methods for control-flow analysis and flow fact generation
- exploration of new techniques for WCET analysis, like new WCET computation methods or parameterized WCET analysis
- code generation strategies or compiler optimizations to obtain good worst-case performance, small execution-time jitter, or code that can be analyzed for its WCET with low efforts
- investigation into back-annotation methods for providing the programmer with detailed information about the timing and other relevant feedback about the analyzed code
- implementation and evaluation of new hardware features that improve the time-predictability of code execution, like new cache replacement strategies, scratch-pad architectures, prefetching mechanisms, alternative instruction sets or CPU architectures
- investigation of programming paradigms and programming languages that could be compiled to the LLVM intermediate representation

This list is by far not complete, and still we feel that it provides a good picture about the opportunities that an open timing analysis platform could offer. The benefit such a research platform can offer to the research community will strongly depend on its acceptance and use. The more people use the platform and add components the greater the value will become for its future users.

**Availability**

We are currently preparing a public release of the platform prototype. A link to the source code repository will appear in the post-proceedings of the workshop.

**Acknowledgements**

The authors would like to thank the group at MDH for providing the SWEET tool, AbsInt for providing aiT and the anonymous reviewers for their valuable comments. The research leading to these results has received funding from the European Community’s Seventh Framework Programme [FP7/2007-2013] under grant agreement number 214373 (Artist Design).

**References**


