- ALL-TIMES -

D4.4.2 Report on main project results

Version 1.2

Release	*Date*	*Author*	*Comment*
0.1	2010-02-24	Peter Gliwa (GLI)	Initial document framework
0.2	2010-04-26	Björn Lisper (MDH)	First full draft version
0.3	2010-05-06	Björn Lisper (MDH)	Changed to be fit as part of the Project Final Report
0.4	2010-05-08	Björn Lisper (MDH)	Changed text for license and distribution
0.5	2010-05-09	Marek Jersak (SYM)	Added SYM tool extensions
0.6	2010-05-09	Björn Lisper (MDH)	Added Q/A table
0.7	2010-05-10	R. Heckmann (ABS)	Some improvements and additions
0.8	2010-05-13	Björn Lisper (MDH)	Added discussion of goals
0.9	2010-05-13	Ian Broster (RPT)	Review and tidy
1.0	2010-05-13	Björn Lisper (MDH)	Finalized report
1.1	2010-06-24	Björn Lisper (MDH)	Revised version
1.2	2010-07-28	Björn Lisper (MDH)	Further revised version

1 The ALL-TIMES project

ALL-TIMES is a medium-scale focused-research project within the European Commission's 7th Framework Programme on Research, Technological Development and Demonstration. It consists of six project partners from both industry and academia. Each project partner is an expert in its particular field and is also a provider of at least one tool for *timing analysis*. More details to the ALL-TIMES project in general can be found on the website www.all-times.org.

2 Introduction

Correct timing of real-time embedded systems is a subject with wide industrial relevance. *Timing analysis* is vital for improving the reliability, performance, and efficiency of embedded systems. It also helps reduce the overall system costs by validating timing requirements, reducing the cost of development, and reducing unit costs in production.

There is a significant body of European research and experience in this area, manifested by a number of tool providers that have commercialised academic results. Today, there is a set of commercially available timing analysis tools, as well as prototype tools developed in academia. However, the tools have operated mostly in isolation: this means that their full potential to aid the embedded software development has not been utilized. There is also a continuous need to transfer new results from academia into industrial use.

The ALL-TIMES project has addressed these issues. The project has combined major European research and development results of timing tools to strengthen the European lead in the area of timing measurement and analysis. It has enabled the interoperability of tools from commercial tool providers as well as academia, and it has developed integrated tool chains using open tool frameworks/interfaces.

3 Partners, Tools, Contact Info, and Overall Project Results

The ALL-TIMES project has taken place Dec. 1st, 2007 – May 31st, 2010. ALL-TIMES has the following six partners, each with a specific tool:

- Mälardalen University (coordinator, academic).
  Tool: SWEET, code-level timing analysis tool
  Contact person: Björn Lisper
  Address: School of Innovation, Design, and Engineering, Mälardalen University, P.O. Box 883, SE-721 23 Västerås, Sweden
  Email: `bjorn.lisper@mdh.se`

- Vienna University of Technology (academic).
  Tool: SATIrE, general purpose code level analysis tool
  Contact person: Jens Knoop
  Address: Institut für Computersprachen, Fakultät für Informatik, Technische Universität Wien, Argentinierstraße 8 / 4 / E185.1, A-1040 Wien, Austria
  Email: `knoop@complang.tuwien.ac.at`

- AbsInt Angewandte Informatik GmbH (SME).
  Tool: aiT, code-level timing analysis tool
  Contact person: Christian Ferdinand
  Address: AbsInt Angewandte Informatik GmbH, Science Park 1, D-66123 Saarbrücken, Germany
  Email: `ferdinand@absint.com`

- Gliwa GmbH (SME).
  Tool: T1, code/system-level timing measurement/analysis tool
  Contact person: Peter Gliwa
  Address: GLIWA GmbH embedded systems, Dollmann Str. 4, D-81541 München,

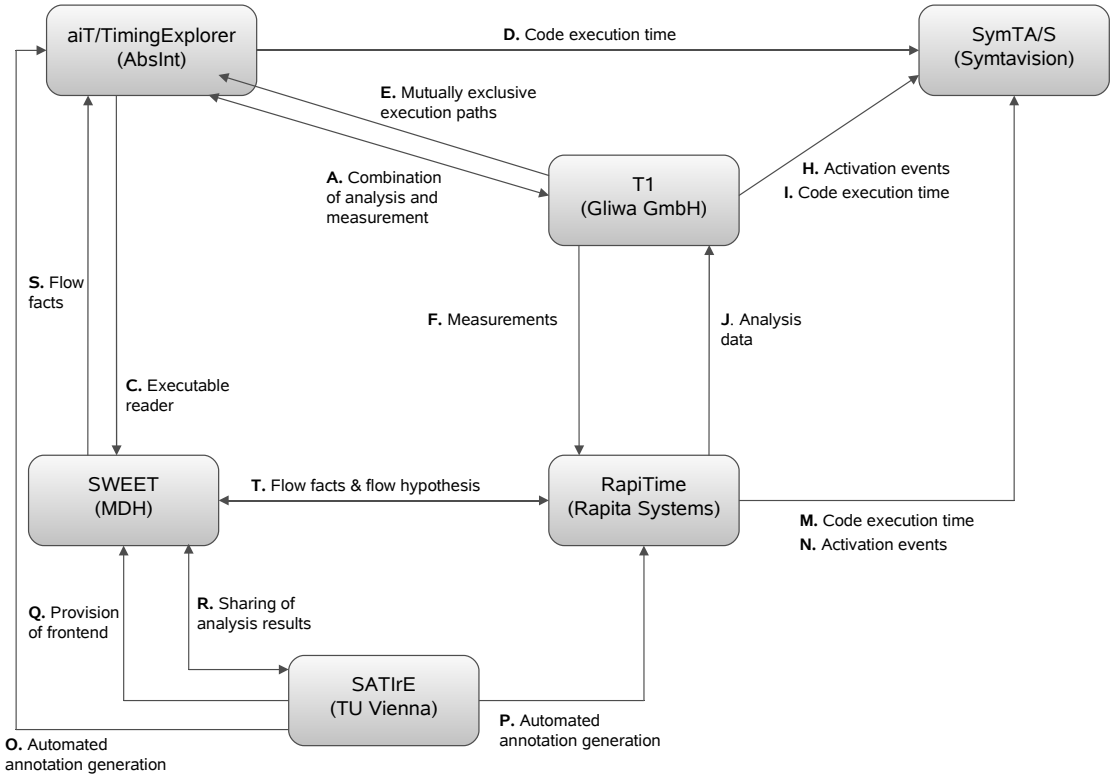

Figure 1: ALL-TIMES integrations between timing analysis tools

Germany
Email: `peter@gliwa.com`

- Symtavision GmbH (SME).
  Tool: SymTA/S, system-level timing analysis tool
  Contact person: Marek Jersak
  Address: Symtavision GmbH, Frankfurter Straße 3b, D-38122 Braunschweig, Germany
  Email: `jersak@symtavision.com`

- Rapita Systems Ltd (SME).
  Tool: RapiTime, code-level timing analysis tool
  Contact person: Nicholas Merriam
  Address: Rapita Systems Ltd., IT Centre, York Science Park, York, YO10 5DG, United Kingdom
  Email: `nmerriam@rapitasystems.com`

Fig. 1 shows the different tools, and the different tool couplings that have been created within the project.

The project has brought the following main technical results:

- integrated *methodologies* for timing analysis,
- prototypes of *integrated tool chains*,

- new/improved *code and timing analysis tools*,
- new *tool couplings* (the so-called "use cases"), including *open tool interfaces*, and
- a *validation* of the integrated tool chains.

In this report we provide a summary of these results. In Section 4 we give a short introduction to timing analysis. Section 5 describes the methodology that has been developed in the project. We give an account for the development of new and improved timing analysis tools in Section 6. In Section 7 we describe the tool couplings (use cases) that have been implemented, and Section 8 decribes the open interface formats supporting these couplings. Section 9 gives an account for the validation of the resulting tool chains. In Section 10, finally, we summarize the main results of the project and draw conclusions about their impact on the timing analysis process.

4 A Short Primer to Timing Analysis

The term "timing analysis" refers to the action of identifying, measuring, displaying, describing or forecasting the time-related behaviour of the software of a (typically embedded) computer system. Timing analysis is a central activity when designing and developing real-time systems. It focuses on either of the following two levels:

- Code level,
- System level.

Code-level analysis assumes an isolated view of a piece of code, whereas system-level analysis takes the complete system into consideration.

The most important timing measures for real-time systems are

- the worst-case execution time (WCET) on the code level
- the worst-case response time (WCRT) on the system level

A variety of techniques have been developed to perform timing analysis. The most important code-level techniques include:

1. *end to end measurement* can be highly automated, and if the right inputs can be identified for the worst case, measurements can derive an exact WCET. However, triggering the inputs to generate a certain situation (worst case) can be difficult. Thus, "end to end" measurement may underestimate the WCET.

2. *static analysis*, which analyses the program without actually running it. It relies on mathematical models of the software and hardware involved. Assuming that the models are correct, the analysis will derive a WCET estimate that is above the true WCET. It may require manual provision of information that can be hard to derive automatically, like upper bounds on loop iterations. Thus, it is harder to automate completely.

3. *measurement-based* (or *hybrid*) analysis, which combines static analysis and measurements, has the advantage of avoiding building hardware models, instead measuring execution times for small parts of the code on the hardware itself. These methods are also rely on adequate testing but to a lesser degree than the purely "end to end" measurement methods.

The most important system-level analysis techniques include:

1. *end to end measurement*: essentially the same approach as on the code-level. However, the focus is not on a single piece of code but instead on timing properties of the full system.

2. *scheduling analysis*: Typically, WCRT's of tasks and functions are calculated based on the WCET's of the different tasks of the system, the scheduling strategy, and the activation frequency of each task. Scheduling analysis works equally well with code execution times obtained by any of the code-level techniques, as well as with estimated code execution times. This allows early-stage exploration and budgeting.

The tools in ALL-TIMES cover all the different timing analysis techniques listed above.

5 Methodology

This section provides a summary of the public deliverable D2.23.2 *Final prototype of integrated timing analysis methodology*[1].

5.1 Overview

The ALL-TIMES methodology helps the user choose the best combination of timing analysis tools and techniques for a given situation. It covers both *early exploration* and *late verification*, as well as *code level* and *system level* analysis.

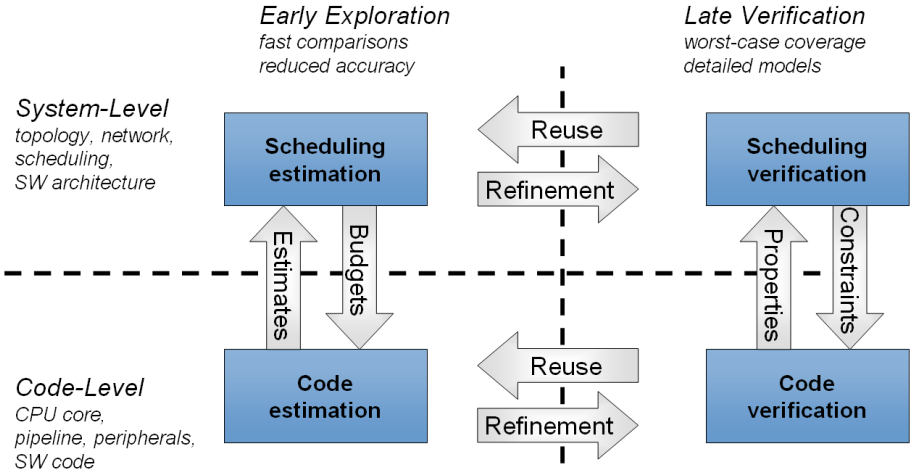

Figure 2: The four quadrants of the ALL-TIMES methodology and their relationships

Figure 2 summarizes the four aspects of the ALL-TIMES methodology and their relationships. As development progresses, early phase models are refined into late phase models. In the opposite direction, late phase models of a predecessor are reused as the basis for early phase models of the successor. In the early phase, system-level models are used to derive budgets for code-level timing. In the opposite direction, code-level analysis provides timing estimates for consideration on the system level. In the late phase, system-level models are used to specify constraints for code-level timing. In the opposite

[1]Available at `www.all-times.org`.

direction, code-level analysis provides timing properties for consideration on the system level.

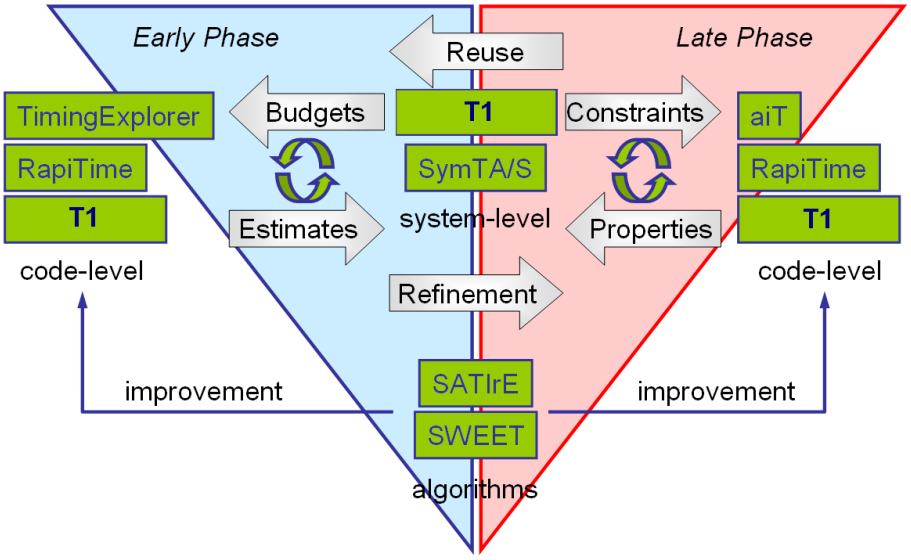

Figure 3: Relationships of the individual ALL-TIMES tools in the ALL-TIMES methodology

Figure 3 shows the relationships of the individual ALL-TIMES tools in the ALL-TIMES methodology. SymTA/S is the tool that is used for system-level analysis, both in the early exploration and late verification phase. TimingExplorer[2], RapiTime, and T1 can be used for code-level analysis in early exploration, passing the results to SymTA/S. Similarly, aiT, RapiTime, and T1 can be used for a more precise code-level analysis in the late verification phase. In both phases, SATIrE and SWEET can be used to increase the precision of timing estimates as well as improving the level of automation by performing source-level code analyses and passing on the results to aiT/TimingExplorer or RapiTime.

Figure 3 shows how the tools fit into the different design phases. Figure 4 gives a different picture of the tools, and tool connections, which focusses on the possible workflows and tool chains.

The system-level analysis methodology also includes timing analysis of networked systems, which is supported by SymTA/S. ALL-TIMES, however, only addresses single controller systems.

5.2 Integrated System-Level and Code-Level Methodology

The code-level analysis tools T1, RapiTime, and aiT/TimingExplorer can all, in different ways, estimate the WCET's of tasks. Given these estimates, SymTA/S can perform a schedulability analysis on system level. Important support for the methodology is provided by the automation of this information exchange through a common exchange format called "eXtensible Timing Cookies" (XTC 2.0). The format supports an iterative workflow between code- and system level tools, with a request/response pattern of communication.

Another important part of the methodology is tracing. By making the system traceable, different problems can be pinpointed. When the area around the problem is identified, the

[2]TimingExplorer is a version of aiT adapted for early stage analysis. See Section 6.

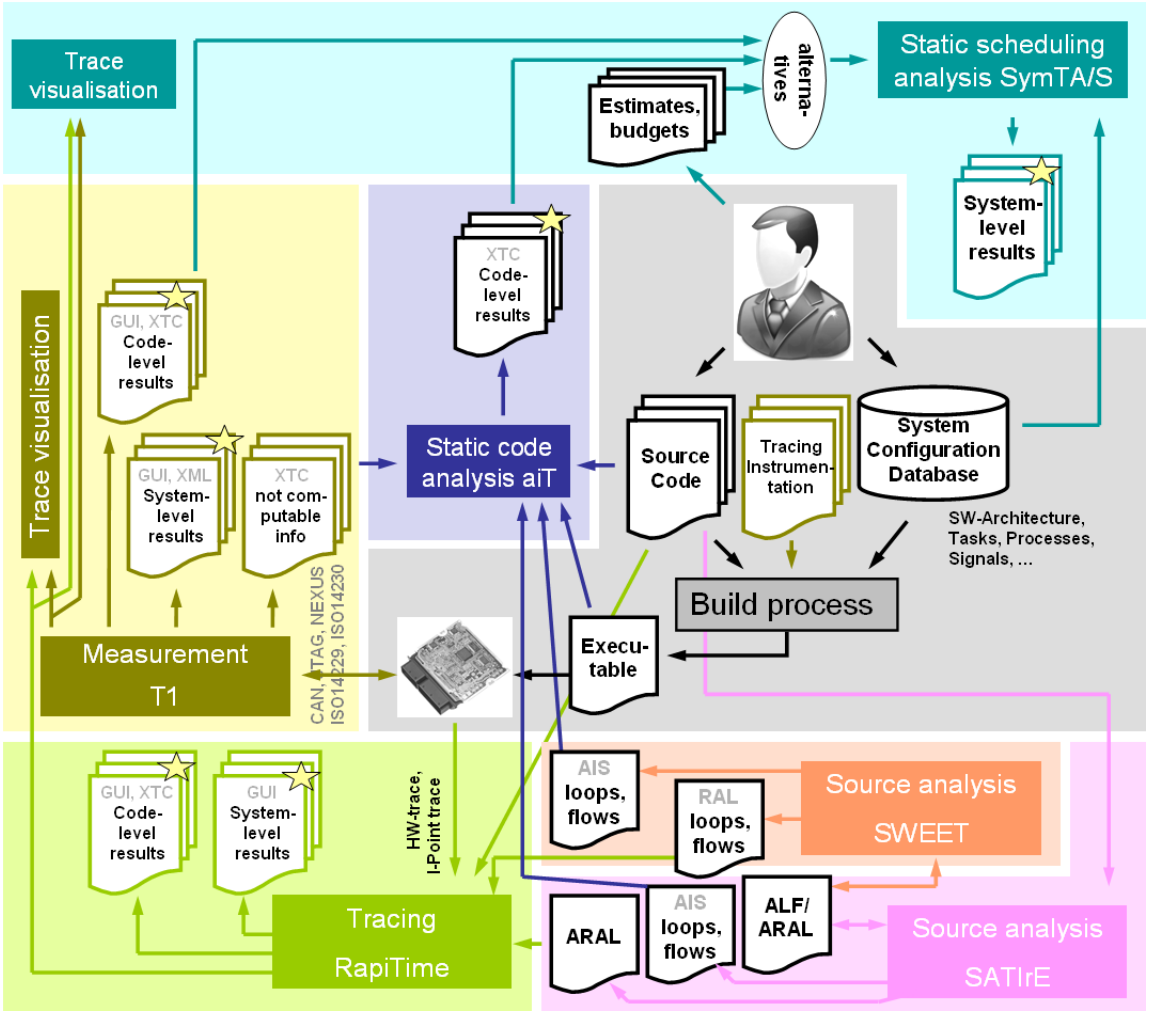

Figure 4: Tools, analysis techniques, tool connections, formats.

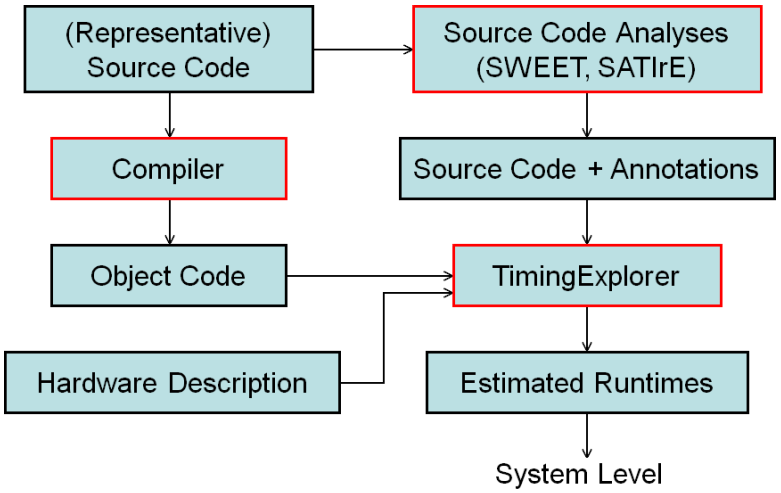

Figure 5: Workflow for source-code analysis and static timing analysis

code can be instrumented to allow a finer inspection. The ALL-TIMES toolset provides support for this process in various ways, including a common standard for trace viewers with two implementations. The standard includes a common trace format, which further facilitates the exchange of trace information between the tools.

The workflows for early exploration and late verification are similar. For late verification, some of the code level tools are used to obtain timing estimates such as WCET, and SymTA/S then peforms an analysis on system level using these estimates. For early design space exploration and time budgeting, hypothetical WCET estimates for different hardware configurations are generated and subsequently used by SymTA/S to verify whether timing constraints will be met. The static analysis tool TimingExplorer is designed for this purpose, but also RapiTime or T1 can be used if, for instance, there are simulators available for the intended hardware platforms.

5.3 Code-Level Methodology

Code level analysis is done to estimate WCET's, to be used by system level tools, but it can also be used for other purposes such as identifying parts of the code with performance problems (timing debugging). For computing WCET's, any of T1 (end-to-end measurements), RapiTime (hybrid analysis), or aiT/TimingExplorer (static analysis) can be used. T1 and RapiTime require an instrumentation of the code to make it traceable: this is not needed for aiT/TimingExplorer.

RapiTime and aiT/TimingExplorer estimate the WCET using program flow constraints (loop iteration bounds, infeasible paths, function pointer sets). The tools have limited capability to derive these constraints, so a user must sometimes provide some of these manually, through annotations. The need for this can be reduced by applying a source-code analysis that automatically derives some of the missing constraints. The tools SATIrE and SWEET can be used for this purpose: they both compute such constraints from C source code, and are able to export them in the annotation formats used by aiT/TimingExplorer and RapiTime. SATIrE computes function pointer sets, and SWEET computes loop iteration bounds as well as a variety of infeasible path constraints. Fig. 5 shows a typical workflow for code level analysis (static analysis, early stage, using source-code analysis).

The workflows for late stage analysis are quite similar.

T1 can also estimate loop iteration bounds and function pointer sets from traces, and export this information to aiT/TimingExplorer through the XTC 2.0 format. This provides an alternative to source-level analysis in situations where the source-level analysis is hard to apply.

5.4 Tradeoffs between Automation and Confidence

The code-level tool combinations enabled by the ALL-TIMES tool connections provide different tradeoffs between automation and confidence in the result. Which combination to select depends on the requirements: for late stage timing verification the requirements on confidence are usually high, in particular for safety-critical systems, whereas for early stage design space exploration analysis time and level of automation are more important.

In general, a static analysis tool like aiT gives the highest confidence in the result, and given enough information it can compute WCET estimates that are guaranteed not to underestimate the actual WCET. On the other hand, a considerable amount of information may have to be provided by the user, and analysis times can be long. Tools that are based on hybrid methods, like RapiTime, provide somewhat less conficence since the accuracy of the estimate now is dependent on the quality of the test vectors. On the other hand, these methods avoid the sometimes intricate modelling of low-level hardware timing properties. WCET analysis that is purely based on end-to-end measurements and high-water-marking provides the lowest confidence by also the highest level of automation.

For static and hybrid WCET analysis methods, a very significant time may have to be spent on writing manual annotations to provide program flow constraints that the tool fails to derive automatically. ALL-TIMES has enabled new combinations of tools, where such constraints can be derived by one tool and used by another. This can be used both to improve the level of automation, and to increase the confidence in the result. Again, there is a tradeoff between these objectives.

The highest level of confidence is given by a static program flow analysis of the executable. This is done by the tool aiT, and SWEET can also do it using a translation into ALF through use case C. But static program flow analysis on this low level of code is hard, and the level of automation may be low.

Next level of confidence is provided by a static program flow analysis of the source code. SATIrE and SWEET can perform source-level analysis, and pass the results to aiT and RapiTime. This analysis is not completely safe, since the compiler might apply optimizing transformations that alter the program flow. But usually the discrepancy is low, and in addition optimizing transformations (like loop unrolling) tend to lower the actual bounds, leaving the bounds derived at source level safe. Coding standards for safety-critical systems sometimes prescribe that compiler optimizations are turned off: this provides an increased confidence in the results of source-level program flow analysis. The level of automation for source-level analysis is typically higher than for analysis of executables, since there is more information in the code.

Program flow analysis based on tracing provides the lowest level of confidence. Here, the quality of the result depends strongly on the appropriateness of the test vectors. On the other hand, this kind of analysis can often be completely automated. T1 can do this kind of analysis, and pass the results to aiT.

A caveat with source-level analysis is that for best results, the source code must be available for all the significant parts of the system. This is not always the case. If some

important parts of the source code is missing, then the analysis must make assumptions about its properties. If these assumptions are conservative there will be high confidence in the results, but possibly large overestimations or even failure to deliver results. Less conservative assumptions may be unsafe, leading to possibly underestimated program flow constraints.

6 Tools

In this section, we briefly summarize the development of new or improved tools that has taken place within the ALL-TIMES project.

6.1 System-Level Tools

SymTA/S has been extended with support for the new XTC 2.0 format, as well as the ATF format for exchanging trace information. These interfaces cover all the use cases where SymTA/S is involved.

SymTA/S has been further extended with a Distribution-Analysis prototype, in order to capture both worst-case schedules as well as typical cases, their distribution and the probability of worst cases.

Common Trace Viewers Symtavision has developed a prototype tool 'Symtavision Trace-Analyzer' based on the ALL-TIMES agreed trace viewer requirements. Symtavision has also added support for multicore ECUs.

Gliwa has extended their existing trace viewer to support the common trace format (ATF) that has been developed within the project, as well as finer trace resolution on the code level.

6.2 Code-Level Tools

TimingExplorer The TimingExplorer tool has been developed within the project. It is a version of aiT for static WCET analysis, which is tuned to suit design space exploration in early design phases. Therefore, it sacrifices absolute safety for analysis speed, making a more approximate WCET analysis. This is useful in design space exploration, where many different hardware configuration alternatives must be explored within reasonable time. TimingExplorer is fully compatible with aiT as regards interfaces, and can thus be hooked up in tool chains in exactly the same way as aiT. Both TimingExplorer and aiT have been provided with interfaces for the XTC 2.0 format.

SWEET A new version of SWEET has been implemented, which uses ALF as input format for its program flow analysis. SWEET is a full-fledged WCET analysis tool, but only the program flow analysis is of concern within the ALL-TIMES project and it is this part that has been re-implemented. It can now generate a number of new program flow constraints. It has been enhanced with a new format for input value annotations, to allow a more precise value-sensitive analysis. Its Flow Fact format for representing program flow constraints has been extended to be more expressive, and to facilitate the translation into the corresponding annotation formats used by aiT/TimingExplorer and RapiTime. SWEET has also been extended with backends that generate flow facts in the ARAL, aiS, and RapiTime annotation formats: these backends implement use cases R, S, and T.

SATIrE is a source program analysis/transformation tool built on top of the Rose compiler framework[3]. It has been extended in various ways within the project to support source level analysis. Several aspects of pointer analysis have been implemented: a post-analysis phase for pointsTo information, extracting function-call/function-pointer specific information (fcall-points-to), an analysis for FCALL-ASL calculation (abstract fcall source code locations), and a pointsTo data generator for ASL based connections. A flow-sensitive interval analysis has also been implemented. The "melmac" tool for translating Rose's internal format into ALF has been implemented: together with Rose's C frontend this provides a C frontend for SWEET implementing use case Q. SATIrE has also been extended to export function pointer sets to the aiS and RapiTime annotation formats, thereby implementing use cases O and P.

RapiTime has been extended with new interfaces for the XTC 2.0 format, allowing easy and error-free information exchange with other tools. RapiTime has now support for the ATF format for exchanging trace information; ATF is so rich in information that RapiTime can automatically construct timing reports from ATF files without requiring source code. RapiTime can also export ATF directly. The ATF support in RapiTime has enhanced RapiTime "Rewind" v2.3, allowing it to relate wall-clock times to events in the trace. RapiTime's annotation format has been extended to use the Abstract Source Location (ASL) format to specify program locations. This has benefits for both user interaction and for importing annotations from SWEET. RapiTime has now an ARAL reader. This allows RapiTime to support results from SATIrE analysis.

7 Tool Couplings (Use Cases)

The following tool couplings (use cases) have been implemented within the project. See also Fig. 1:

A. Combination of analysis and measurement (aiT ↔ T1) T1 can export measurement-based timing information,as well as loop iteration bounds, to aiT through the XTC 2.0 format. The connection also enables comparisons between measured end-to-end execution times from T1, and estimates done by the static analysis of aiT.

C. Executable reader (aiT → SWEET) A translator for binary formats, stored in aiT's internal representation, into ALF has been implemented for the NEC V850 and PowerPC instruction sets. This connection enables SWEET to perform program flow analysis on binaries, using aiT as a "frontend".

D. Code execution time (aiT → SymTA/S) Through XTC 2.0, aiT exports WCET estimates for uninterrupted tasks to SymTA/S. This connection existed already before ALL-TIMES, through the earlier version of XTC, but has been maintained into XTC 2.0.

E. Mutually exclusive execution paths (T1 → aiT) Through this connection T1 can export information about unreached (and therefore possibly unreachable) program points, as well as function-pointer destinations, to aiT. The format is XTC 2.0.

[3] `www.rosecompiler.org`

F. Measurement input (T1 → RapiTime) Traces are exported from T1 to RapiTime through the ATF common trace format. RapiTime can then perform a WCET analysis using the traces, and generate a detailed WCET report that can be viewed in the RapiTime report viewer.

H. Activation events (T1 → SymTA/S) Traces containing activation events are exported from T1 to SymTA/S using the ATF common trace format. SymTA/S can then perform a schedulability analysis based on this information, and the trace viewer of SymTA/S can visualize the trace data.

I. Code execution time (T1 → SymTA/S) BCET's/WCET's extracted from traces are exported from T1 to SymTA/S through the XTC 2.0 format. This use case is closely related to use case H.

J. Analysis data (RapiTime → T1) This use case allows the export of traces from RapiTime to T1 using the ATF common trace format. T1 users can then reuse the RapiTime instrumentation.

M. Code execution time (RapiTime → SymTA/S) RapiTime exports WCET estimates to SymTA/S, using the XTC 2.0 format. This use case is similar to use case D.

N. Activation events (RapiTime → SymTA/S) RapiTime provides traces containing runtime events such as scheduler events, task/thread activation events, etc., using the ATF common trace format. SymTA/S imports the trace data to show its visual representation.

O. Automated annotation generation (SATIrE → aiT) SATIrE exports function pointer sets to aiT/TimingExplorer using the AIS annotation format. This reduces the need for manual annotations.

P. Automated annotation generation (SATIrE → RapiTime) Similarly to use case O, SATIrE exports function pointer sets to RapiTime using RapiTime's annotation format and ASL (abstract source code locations).

Q. Provision of frontend (SATIrE → SWEET) The melmac tool converts Rose's internal format (which SATIrE is built upon) into ALF code, which can be analyzed by SWEET. Through the C frontend that belongs to the Rose framework, this makes SATIrE act as a C frontend to SWEET enabling SWEET to perform source-level program flow analysis. Melmac also generates "map files" that relate code locations in the C code and the corresponding locations in the ALF code: SWEET needs this information when generating annotations for aiT and RapiTime.

R. Sharing of analysis results (SATIrE ↔ SWEET) The ARAL format has been designed to exchange program analysis information. API's have been created that can be used to read and write analysis results in ARAL. SWEET uses one of these API's to generate program flow constraints in ARAL, and SATIrE exports function pointer sets in this format.

S. Flow facts (SWEET → aiT) "Flow facts" (program flow constraints) generated by SWEET are exported to aiT/TimingExplorer in the AIS format. SWEET can generate a variety of flow facts, ranging from simple loop iteration bounds to elaborate infeasible path constraints. The provision of these can reduce the need for manual annotations.

T. Flow facts & flow hypotheses (SWEET ↔ RapiTime) Similarly to use case S, SWEET can also export flow facts to RapiTime in the RapiTime format, using ASL to specify program locations. An experimental version of SWEET that can read traces from RapiTime and generate flow facts from these (so-called "flow hypotheses") has also been implemented.

8 Interface Formats

In this section we briefly decribe the open interface formats that have been developed within the project. Specifications for these formats are available at www.all-times.org.

8.1 System-Level Interface Formats

XTC 2.0 XTC stands for "eXtensible Timing Cookies", and is an XML-based format used to exchange different kinds of timing analysis data. An earlier version (1.0) was developed in the FP6 INTEREST project as an interface between SymTA/S and aiT. This interface has been extended in different ways within ALL-TIMES, and it is now also supported by T1 and RapiTime. XTC supports a request/reply style of communication, where different tools successively add information to the cookie. With XTC 1.0, only information about worst-case execution times and maximum stack usage could be exchanged. The new XTC 2.0 developed since then also includes iteration bounds of loops, response times, activation patterns aligned with the upcoming AUTOSAR 4.0 and TIMMO event model descriptions, and scheduling overheads (activation and termination overhead, context-switch overhead, and context-switch cache penalties). Data are now annotated with their source (e.g. static analysis, tracing, simulation, or configuration).

ATF This is a common, XML-based trace format that has been developed within ALL-TIMES. It allows the exchange of trace information, including metadata, among the tools concerned. It is supported by aiT, SymTA/S, T1, and RapiTime. Similar to XTC, it has a "cookie" function allowing different tools to add information.

8.2 Code Level Interface Formats

ALF ALF has its origins in the ARTIST2 FP6 Network of Excellence, and stands for "Artist2 Language for Flow analysis". It has been further refined, and implemented, within ALL-TIMES. ALF is a code format, basically a programming language, but it is not intended to be written by humans. Rather, the idea is to translate programs in other code formats into it in order to analyze them. ALF is thus designed to be (1) easy to analyze, and (2) so it can faithfully represent code on different levels, ranging from source level to binary level. Therefore it includes both high-level constructs, such as function calls, as well as low-level constructs such as jumps with computed targets. The new version of SWEET, which is developed within ALL-TIMES, performs a program flow

analysis of ALF, and SATIrE has been extended to generate ALF code from C sources. There is also a translator from aiT's internal representation of binaries into ALF.

ARAL This is a format to exchange program flow analysis information, which has been developed within ALL-TIMES. It allows to express contexts such as call strings, and it has a language for expressing analysis results. It is used by SATIrE and SWEET.

ASL ASL stands for "Abstract Source Locations", and it is a format to express locations in source code. This information is needed to express source code analysis results. ASL is now used for this purpose in RapiTime's annotation format, and both SWEET and SATIrE export annotations in this format using ASL.

9 Validation

The two principal project objectives have been:

- O1: Integration of timing analysis tools, and
- O2: Increase of productivity of embedded systems development projects by 25% of the design time pertaining to timing issues.

These objectives can be refined into seven more technical project goals, which contribute to the project objectives according to the following table:

Goal	Description	O1	O2
g1	Enable source-code/binary code analysis integration	High	-
g2	Reduce number of manual annotations	-	High
g3	Minimise interference by instrumentation	-	High
g4	Open tool integration	High	-
g5	Improve accuracy of system-level timing verification	Low	High
g6	Enable timing estimation early in the design	Low	High
g7	Demonstrate results in industrial context	Low	High

The validation in ALL-TIMES has been carried out using two case studies involving real code for automotive systems: AFS (Active Front Steering) from BMW, and VECU (VEhicle Control Unit for fuel cell) from Daimler. The validation was done using two typical scenarios for using the tool chains developed within ALL-TIMES: an early stage design space exploration scenario, and a late stage verification scenario. The toolflows used are shown in Figures 6, and 7.

Originally, the intention was to use the VECU case study code to validate the early stage tool chain, and the AFS code for the late stage chain. However, it turned out that the source code for AFS, for NDA reasons, was stripped of most of its interesting contents. This made it unsuitable for validating the source-level analyses (SWEET, SATIrE, and associated use cases). It was therefore decided to validate also the late stage chain using VECU. In addition, some tool connections not involving source-level analysis were validated using AFS.

Validation w.r.t. O1 has been done on a per use-case basis, where a use case has been considered validated if used in any of the early/late stage scenarios. All the finally implemented use cases have been validated in this way except use case C (binary reader), use

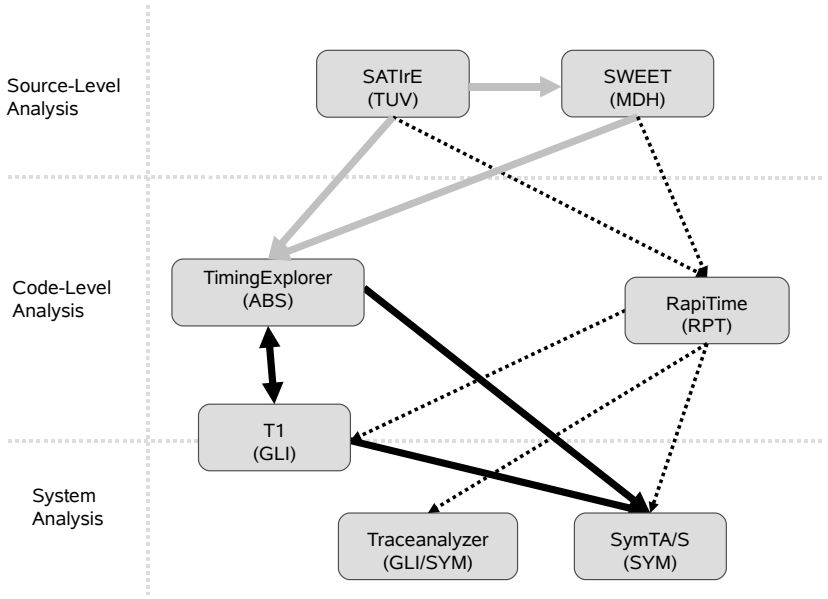

Figure 6: Tool chain for early stage development.

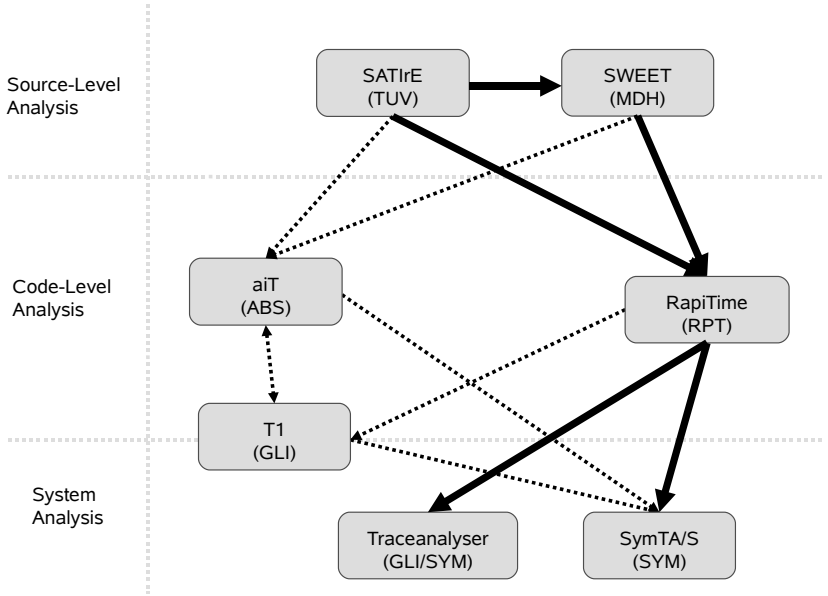

Figure 7: Tool chain for late stage development.

Workflow	Before	After
Set up system model	103.5 min	2 min
Set up aiT configuration for current hardware	72 min	8 min
Provide annotations for loop bounds and function pointers	3164 min	191.5 min
Alter aiT configuration to new hardware	5 min	5 min
Perform WCET analysis by aiT	48 min	48 min
Pass WCETs to SymTA/S	16 min	1 min 20 s
Scheduling analysis by SymTA/S	5 min	5 min
TOTAL	56.80 h	4.34 h

Table 1: Early stage design scenario

Workflow	Before	After
Find function pointers and select loops for instrumentation	3 h	0.5 h
Instrument and analyze for WCETs	3 h	3 h
Check for WCET pessimism	4 h	2 h
Check for WCET optimism	4 h	2 h
Regenerate RapiTime reports	0.5 h	0 h
Build system model	1.5 h	0.26 h
Scheduling analysis by SymTA/S	0.08 h	0.08 h
TOTAL	16.08 h	8.84 h

Table 2: Late stage design scenario

case O (automated annotation generation), and use case R (sharing of analysis results). Use cases C, and R did not fit well into the tool chain scenarios, and in the end none of them turned out to be essential for achieving the goals of the project. Use case O was not O1 validated mainly due to lack of time, but it is very similar to use case P and results can be expected to be very similar as well.

Validation w.r.t. O2 has been done in the following way. For each use case, its productivity increase has been estimated from the estimated times to effectuate the tool coupling, with and without the added support. The total productivity increase was then estimated from the early and late design scenarios mentioned above, involving the use cases. For each scenario a typical workflow was set up, and for each step in the workflow the required time without and with the ALL-TIMES technologies (before/after) was estimated. See Tables 1 and 2.

By summing the estimated times for the individual steps, an estimate for the total workflow was obtained. In this way, the improvements in time for performing a timing analysis could be estimated for each scenario by comparing the times needed before and after ALL-TIMES. The results indicate a tenfold speedup for the early design scenario, and a speedup by a factor of two for the late verification scenario. The estimates obtained have been complemented by interviews with experts at Daimler and BMW, which mainly confirm that our estimates are realistic.

In both scenarios, the main improvements come from the more automatic derivation of program flow constraints enabled by the tool couplings. In the early design space exploration scenario, the program flow constraints are derived by tracing. It has been assumed that test vectors provide enough coverage to yield a sufficient confidence in the results. The late stage verification scenario uses source-level analysis for the same purpose. Here, is is assumed that source code is available for all the significant parts of the system. For both scenarios we have assumed that, at the start of the work-flow all the tools have been installed and appropriately integrated ready for any project but with no prior knowledge of the project in question.

The design scenarios cover the design activities that pertain to timing analysis. Also other design activities, like re-coding time-critical parts of the code that constitute bottlenecks, may be devoted to timing issues. However, the large improvements for the design scenarios provide strong evidence that the goal of a 25% productivity improvement of design time pertaining to timing issues is met.

Has the project reached its more detailed goals g1–g7? Yes, we consider them achieved, for the following reasons:

g1 *Enable source-code/binary code analysis integration*: achieved, through the source-level analyses and their connections to timing analysis tools (use cases O, P, Q, R, S, T). Use cases O and R were not validated: however, use case O is very similar to use case P, and use case R was not strictly necessary to use in order to validate the source-level analyses. We thus consider the integration achieved.

g2 *Reduce number of manual annotations*: achieved, through the source-level, and trace-based program flow analyses (use cases above, and use cases A, E). In particular, the trace-based analyses turned out to be efficient for this purpose in the validation.

g3 *Minimise interference by instrumentation*: achieved. The source-level analysis tools can find program parts with static program flow, like loops that always iterate the same number of times, which are likely not to require instrumentation (use cases P, T). Also the new version of T1 allows on-line instrumentation, a side-product of use cases A and E so that it is no longer necessary to modify C code when, e.g., measuring the execution times of functions. These techniques are particularly valuable when there is no or limited hardware support for tracing, which means that the instrumentation points must be sparse to avoid large probe effects and overflowing buffers.

g4 *Open tool integration*: achieved, through the various open interfaces developed in the project (XTC 2.0, ATF, ALF, ASL, ARAL). In particular XTC and ATF enable a large number of tool integrations, both on system- and code level. Furthermore, the existing open annotation formats for aiT and RapiTime have been used to interface the source-code analyses performed by SWEET and SATIrE with these tools.

g5 *Improve accuracy of system-level timing verification*: achieved, through the combination of measurement-based and static timing analysis techniques. The measurement-based techniques are often optimistic (missing corner-cases), while the analysis techniques are often pessimistic (conservative algorithms). The combination allows to

- reduce analysis pessimism by measuring where analysis lacks the data to find accurate bounds.

- increase the confidence in measurement by verifying key data with static analysis.

Additionally, interrupt- and task- switching costs as well as measurement overhead are now considered – this improves the accuracy of system-level timing considerably especially in more complex systems.

g6 *Enable timing estimation early in the design*: achieved, through the tool TimingExplorer and the source-level analyses. TimingExplorer allows fast design space exploration of different hardware alternatives, and the source-level analyses can be done already in early design stages.

g7 *Demonstrate results in industrial context*: achieved, through the validation using the industrial case studies. VECU was used as the main case study, and some use cases were additionally validated using AFS. As mentioned above not all use cases were validated: 3 out of 16 were not. However, we consider the 13 validated use cases, and the scenario validation w.r.t. O2, to be fully sufficient to demonstrate all the important results of ALL-TIMES.

We can thus conclude that the project has fully reached its objectives, as well as the seven detailed goals.

10 Conclusions

Overall the calculated and estimated productivity increases, as well as the expert feedback, show that the project goals have been achieved, and that the project results have the potential to substantially improve the timing analysis process for time-critical software.

One particular success is the trace-based analyses of program flow properties, like loop bounds and function pointers, since they allow a high degree of automation for code-analysis. This is manifested in very high speed-ups for the corresponding use cases. A conclusion is that these techniques are a good choice if the precision obtained by this flow information is sufficient, and if unsafe estimates can be tolerated.

A second success is the high speed-ups resulting from the automation of information exchange through the XTC and ATF formats. This is also a very clear improvement, enabling a much faster interaction between code- and system-level analysis tools. In addition, this exchange of information enables new opportunities to visualise different kinds of timing information. This further enhances the productivity of the engineer.

A third success is the automation of system modelling and collection of timing data, and the resulting speedup, in particular for roundtrip-engineering between the code- and the system-level. At the same time the now possible choice of code-level techniques, including mixing different techniques, provides system designers with an unprecedented tool-box, that is easily adapted to specific needs (required level of confidence, design phase ...).

The source-level analysis shows less impressive numbers. One reason is that the source-level tools both are academic prototypes, whereas the other prototype tools are derived from commercial industrial-strength tools. A second reason is the fact, that some files containing analysis-critical code could not be parsed, since they were available as object code only.

However, the results also clearly show that source-level analysis improves the quality of timing analysis whenever it is applicable. One reason is the increased confidence in

the analysis results, especially compared with trace-based analysis. For safety-critical systems safe timing estimates may be needed, and then static analysis may be the only option. Another situation is early analysis before binaries are available: then, trace-based methods are not applicable. However, the methods have to be applied at the place in the value chain where source code is available. In particular, automotive suppliers should thus learn to use source-level techniques.

In conclusion, we believe that ALL-TIMES provides sufficient motivation for the automotive industry to pursue timing analysis much more regularly and systematically in the future. OEMs should be motivating their suppliers to use the available techniques for all code that is in their responsibility. In particular the ALL-TIMES industrial partners will be disseminating this message full scale.