- ALL-TIMES -

D2.23.2 Final prototype of integrated system-level verification methodology

Version 1.4

License and Distribution:
This is a public report. It can be re-distributed freely in its original form. The logos used (if any) belong to their respective owners and should be used with prior written permission from the owner.

Release	*Date*	*Author*	*Comment*
0.1	2010-02-24	Peter Gliwa (GLI)	initial document framework including Q/A table
0.2	2010-05-04	R. Heckmann (ABS)	import of D2.23.1 update of XTC description (6.1)
0.3	2010-05-05	Peter Gliwa (GLI)	added section "Obtaining annotation information" added table of content
0.4	2010-05-04	R. Heckmann (ABS)	added questions / answers
0.5	2010-05-04	Björn Lisper (MDH)	added questions / answers
0.6	2010-05-05	Marek Jersak (SYM)	updated introduction and added questions / answers
0.7	2010-05-07	Peter Gliwa (GLI)	added section "Finding reasonable entry points" and extended questions / answers
0.8	2010-05-08	Marek Jersak (SYM)	consistency checks and finalization for review
0.9	2010-05-12	R. Heckmann (ABS)	some updates in code-level section
1.0	2010-05-12	Peter Gliwa (GLI)	a) adapted tool names b) fixed typos, c) added one more approach in section 6.3 d) public license on the front page e) final review and release
1.1	2010-06-28	Björn Lisper (MDH)	revision according to comments from review
1.2	2010-06-29	Peter Gliwa (GLI)	a) review (fixed typos only) b) layout improvement concerning Figure 5 c) added two motivation examples
1.3	2010-07-29 2010-08-17	Marek Jersak (SYM) C. Ferdinand (ABS)	addressed all remaining 2010-05-26 review meeting comments a) removed old future references b) defined use cases named by letters c) focused sections 2 and 3.5 on top-level view, including overall figure (poster) and scenarios d) added section 3.6 on alternatives between tools, clarified role of static analysis added conclusion
1.4	2010-08-17	Peter Gliwa (GLI)	reviewed/extended conclusion

Contents

1 The ALL-TIMES project **5**

2 Introduction **6**

2.1 Motivations for timing analysis 6
2.2 Some Motivating Examples 7
2.2.1 Early phase example 7
2.2.2 Late phase example 7
2.2.3 Scenario: integrated code execution time monitoring 8
2.2.4 Scenario: early verification of code extensions 8
2.2.5 Scenario: relating code-level timing behaviour to system-level behaviour 8
2.2.6 Scenario: locating timing defects 8
2.3 Contribution to ALL-TIMES project goals 9

3 ALL-TIMES Methodology Overview **10**

3.1 Scope 10
3.2 Challenges in early phases 11
3.3 Challenges in late phases 11
3.4 Tool Couplings Addressed 12
3.5 Methodology Overview 12
3.6 Alternatives between tools 14

4 Questions – Answers **17**

5 System-Level Methodology **19**

5.1 System-Level Verification 19
5.2 System-Level Exploration and Optimization 19
5.3 Network Verification 20
5.4 Network Exploration and Optimization 21

6 Integrated System-Level and Code-Level Methodology **22**

6.1 XTC: Exchange format between system-level and code-level tools 22
6.2 Common trace viewer 23
6.3 ECU Verification 23
6.3.1 Scheduling Analysis 25
6.3.2 Tracing and Scheduling Analysis 25
6.3.3 Tracing, Code Path Analysis and Scheduling Analysis 27
6.3.4 Static Code Analysis and Scheduling Analysis 27
6.3.5 Integrated code-level – system-level flow for ECU verification . . . 28
6.4 ECU Exploration and Optimization 31
6.4.1 Influences on End-to-End Timing 31
6.4.2 Execution Time Budgeting using Scheduling Analysis 32
6.4.3 Execution Time Estimation Using Static Analysis 34
6.4.4 Combining ECU budgeting and estimation 34
6.4.5 Early Tracing and Scheduling Analysis 35

7 Code-Level Methodology **36**
7.1 Finding reasonable entry points . . . 36
7.2 Source-Code Analysis . . . 38
7.3 Source-Code Analysis and Static Timing Analysis . . . 39
7.3.1 Early-phase Source-Code Analysis and Static Timing Analysis . . 40
7.3.2 Late-Phase Source-Code Analysis and Static Timing Analysis . . . 41
7.3.3 The "Simple Annotation" Approach for Static Code Analysis . . . 41
7.4 Source-Code Analysis and Tracing . . . 41
7.4.1 Obtaining annotation information by tracing . . . 41
7.4.2 Late-phase Source-code event analysis and tracing . . . 42
7.4.3 Late-phase Source-code path analysis and tracing . . . 43
7.4.4 Early-phase Source-code event analysis and tracing . . . 44
7.4.5 Early-phase Source-code path analysis and tracing . . . 46

8 Conclusion **47**

A Tools and Tool Integrations in ALL-TIMES **48**

B Timing Properties and Constraints **48**
B.1 Timing Properties . . . 48
B.2 Timing Constraints . . . 48

C The modeling of timing in AUTOSAR **50**

D Aerospace and other Non-Automotive Markets **51**

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

This document describes the ALL-TIMES integrated methodology for early-stage and late-stage system-level and code-level timing analysis, optimization and verification. It is an updated, public version of the previous deliverable *D2.23.1*, replacing the initially planned separate deliverables *D2.2.2* and *D2.3.2*. It thus follows the reviewers' recommendation to cover both the system-level and code-level tool integration, as well as the integration between those levels. Hence, the document presents a comprehensive methodology. In particular, it explains the tool couplings from the ALL-TIMES portfolio for a range of typical development tasks. The update includes the experiences obtained during the application of the ALL-TIMES methodology to the case studies, in the form of a *questions and answers table* in section 4 as well as updated descriptions where appropriate (see the revision history table).

2.1 Motivations for timing analysis

In order to analyse or verify correct system timing, various approaches are possible. Which one suits a specific project best, depends on various circumstances that are discussed in the following. Before making a decision in favour of one or the other technique, the motivation should be clarified. This document focuses on the following motivations for the need of system timing analysis.

- **Verifying correct timing.** Certifying software is often mandatory in the aviation industry. Formal methods proving, or supporting the proof of correct timing can help to save certification costs considerably. This is increasingly important for the automotive industry as well. Automotive manufacturers should develop any safety relevant software according to the "state of the art". Otherwise, they could be held liable in case of an accident caused by a software problem. Since the formal approach to prove system timing has been established in several mass production projects in the past few years, this technique is "state of the art". The depth and coverage of the verification depends on the risk analysis. All code ensuring safe function must be proven to be executed in time in any case.

- **Increasing availability.** In automotive, many safety-critical software functions can be turned off, if a problem is detected. Since manufacturers strive for a high availability, it is desirable to verify during development that all health-monitoring functions are executed within their timing constraints.

- **Debugging.** In case of sporadic errors or system crashes, a faulty system timing is often the cause. This often is not obvious, so that correct timing should be explicitly verified/debugged.

- **Budgeting.** In an early design phase, run-time budgets have to be assigned to different software building blocks based on function timing requirements. These budgets have to be broken down into sub-budgets as the software is broken down into smaller building blocks.

- **Estimation.** In parallel with budgeting, feasibility and risk analyses can be performed to check whether the intended partitioning and allocation of software can be implemented at the expected cost.

- **Verifying timing budgets.** If run-time budgets have been assigned in an early design phase, these have to be checked as the project develops. This aspect becomes particularly important if the development is shared between two or more partners or companies.

2.2 Some Motivating Examples

In this section, we give some examples and scenarios motivating the ALL-TIMES technologies and the integrated methodology. The examples are from real life, showing typical situations encountered by project partners where the methodology and the ALL-TIMES toolset/toolchains has the potential to be beneficial.

2.2.1 Early phase example

Consider an automotive OEM who wants to offer advanced driver assistance functions in a future vehicle. Compared to existing cars, this requires multiple innovations, including the integration of vehicle dynamics functions (electronic stability (ESP), active steering, active dampening, engine control, ...) and fusion of different sensor inputs for comprehensive assessment of vehicle surroundings. There is a consensus in the automotive industry, exemplified by AUTOSAR, that such innovations require a flexible and scalable system architecture that focuses on re-usable software building blocks.

In this context, timing analysis is a key enabler for the following tasks:

- Budgeting of available time, be it on the level of individual pieces of software, or end-to-end timing, e.g., from sensor to actuator.
- Estimation of delays due to software execution, arbitration, hand-over or other types of buffering.
- Prediction of the required computation resources to execute a set of software functions.

2.2.2 Late phase example

An automotive tier one supplier provides an ECU together with the platform software to a car manufacturer developing the main functional part of the software. After the first integration where the different parts of the software are glued together, the system shows sporadic crashes which – after a lot of investigation – turn out to be caused by timing problems.

Again, timing analysis is a key enabler for

- tracking down the source of such problems,
- efficiently finding appropriate solutions,
- verifying that similar problems can no longer occur or are at least very unlikely to occur.

2.2.3 Scenario: integrated code execution time monitoring

Most automotive projects include static code checker tools in the build process analysing the software for potential functional issues. MISRA compliance for example is an important topic in this context. Typically, the regular build process is started with a special make-switch activated so performing the analysis does not require much effort.

Static timing analysis can and actually *should* be established in a very similar manner. Developers then immediately see the impact of software changes on the code-level timing.

With the measurement-supported static analysis developed in the ALL-TIMES project, the annotation effort could be reduced to a level that the approach described above becomes applicable even for mass-production projects with their tight schedules.

2.2.4 Scenario: early verification of code extensions

Automotive ECUs typically show a life cycle with several generations. Each generation comes with additional features which require additional computing power – but the processor often remains the same for some of the generations. When starting with the development of a new generation, the question arises if the new features will overload the CPU.

The ALL-TIMES tools can (and in several projects already actually *did*) help to answer this question. For the new features to be developed, execution time estimates are collected from the developers[1]. Based on measurement results of the recent generation captured by Gliwa's T1 and with the help of T1 delay, the system-timing of the next generation can be evaluated or even verified long before the new code segments are available.

Alternatively or in addition, the system timing could be statically analysed using Symtavision's SymTA/S.

With the results of the ALL-TIMES project, the effort for incorporating different tools has significantly decreased.

2.2.5 Scenario: relating code-level timing behaviour to system-level behaviour

Rapita Systems delivered RapiTime to a customer and successfully provided a combination of purely measured timing and coverage values plus the calculated worst-case timing data. The customer was very interested to understand, when longer paths were measured, what was happening at the system level. They wanted to know whether a particular kind of message had been received and to know which subsystems were active. Rapita Systems were unable to provide the customer with this information because the RapiTime tool-chain had eliminated “wall-clock” time information in processing the execution time data. With the combined ALL-TIMES tool-chain, including enhancements to RapiTime, we would have been able to report both the “wall-clock” time of particular events and to show combined system- and code-level traces to answer all their questions.

2.2.6 Scenario: locating timing defects

Rapita Systems delivered RapiTime to a customer with the objective of locating timing defects in one of a number of CPUs in an embedded device. Historically, when timing problems had arisen, the customer had been put to considerable expense to show that the

[1] If the new features already are implemented for other processors or simulation environments, AbsInt's TimingExplorer can support the developers in providing sound numbers.

timing defects did not arise from their code and this effort was felt to be wasted. Using RapiTime, they were able to show not only that their code did not have undesirable timing properties in their standard tests but also that there did not exist pathological worst-case timing behaviours that could be exposed in wider system tests.

However, if we had had access to the ALL-TIMES trace viewer, we would have been able to provide the customer with a solution that quickly and easily showed not only that timing over-runs on their CPU arose not from excessive computation times but that the problems specifically did arise from excessive arrival rates of messages due to defects in one or more of the other CPUs. Furthermore, we could have performed thorough scheduling analysis with SymTA/S to determine the existing tolerance of the system to excessive message rates and possible protection that could be put in place to improve that tolerance.

2.3 Contribution to ALL-TIMES project goals

The definition of WT 2.2 describes the early-stage scope of this document: "Develop a methodology for architecture exploration and performance estimation/optimization based on early timing analysis and 'what-if' analysis."

The definition of WT 2.3 describes the late-stage scope of this document: "Develop a methodology for timing and performance verification using detailed analysis and testing; consider correlations and dependencies to improve analysis accuracy."

Overall, this work contributes to the goals g2 ("Reduce number of manual annotations"), g3 ("Minimise interference by instrumentation"), g5 ("Improve accuracy of system-level timing verification") and g6 ("Enable timing estimation early in the design"), which are defined in the Description of Work for ALL-TIMES.

3 ALL-TIMES Methodology Overview

3.1 Scope

The ALL-TIMES methodology covers both *early exploration* and *late verification*, as well as *code level* and *system level*.

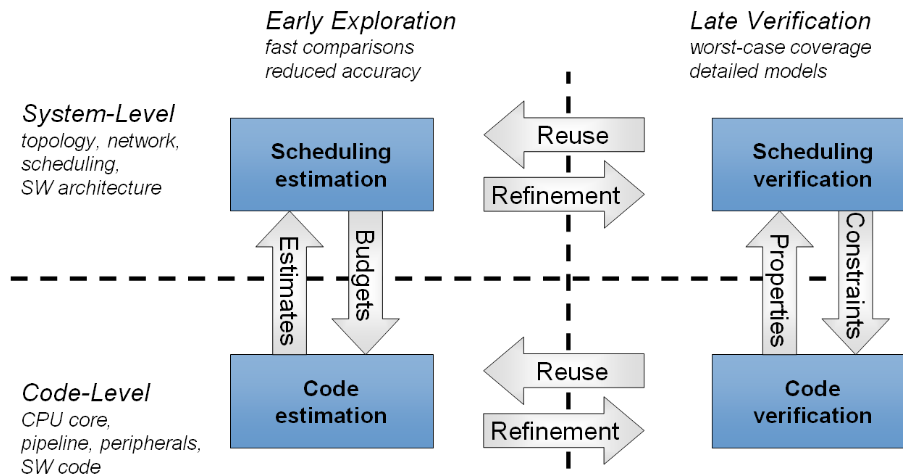

Figure 1: The four quadrants of the ALL-TIMES methodology and their relationships

Figure 1 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 predecessors 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 direction, code-level analysis provides timing properties for consideration on the system level.

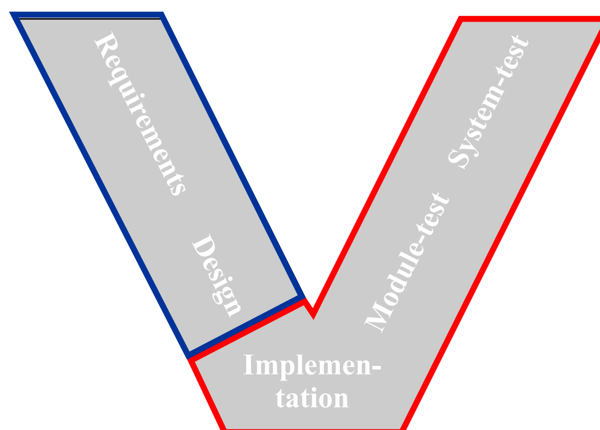

Figure 2: The “early development phase” (blue) and “late development phase” (red) scope of this document

Figure 2 locates the early and late development phases in the V-model. Figure 3 defines

"code level" and "system level" by means of the granularity of the software/components to be analysed. Code analysis is related to a granularity, ranging from finest grained, the phases of a machine cycle, to more coarse grained, a set of runnables grouped to a task or interrupt service routine (ISR). The system level starts at the granularity of runnables and reaches, at the most coarse grained, as far as a complete system built of ECUs which communicate via buses.

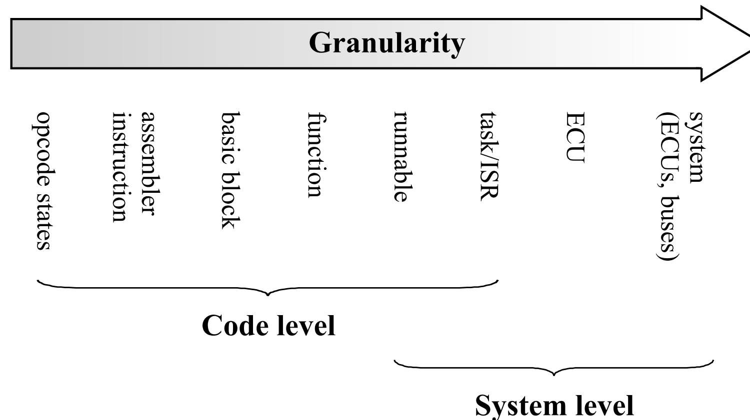

Figure 3: Definition of "code level" and "system level" by means of granularity

3.2 Challenges in early phases

Early development phases are characterized by missing data. In particular, data is at least partially missing on

- detailed configurations of network, ECU, integration
- software run-times

Therefore, the two key questions in early phases are

- where can we get this data from?
- what is an efficient evaluation process?

The ALL-TIMES solution is to

- start with and reuse existing data from previous designs
- perform *virtual* schedule synthesis and integration
- use *virtual* ECU models and code *estimation*

3.3 Challenges in late phases

Late development phases are characterized by huge amounts of data. In particular, code, module and integration tests all produce a large number of logs of various kinds. Therefore, the two key questions in late phases are

- what is the quality of the data with respect to timing?

- what is an efficient evaluation process?

The ALL-TIMES solution is

- focused visualization of specific timing problems in large traces
- combination of worst-case and typical-case analysis
- combination of static system model and measured data

3.4 Tool Couplings Addressed

In ALL-TIMES, the following couplings between partner tools (tool map) have been addressed. The letters (use cases) are referenced throughout this document.

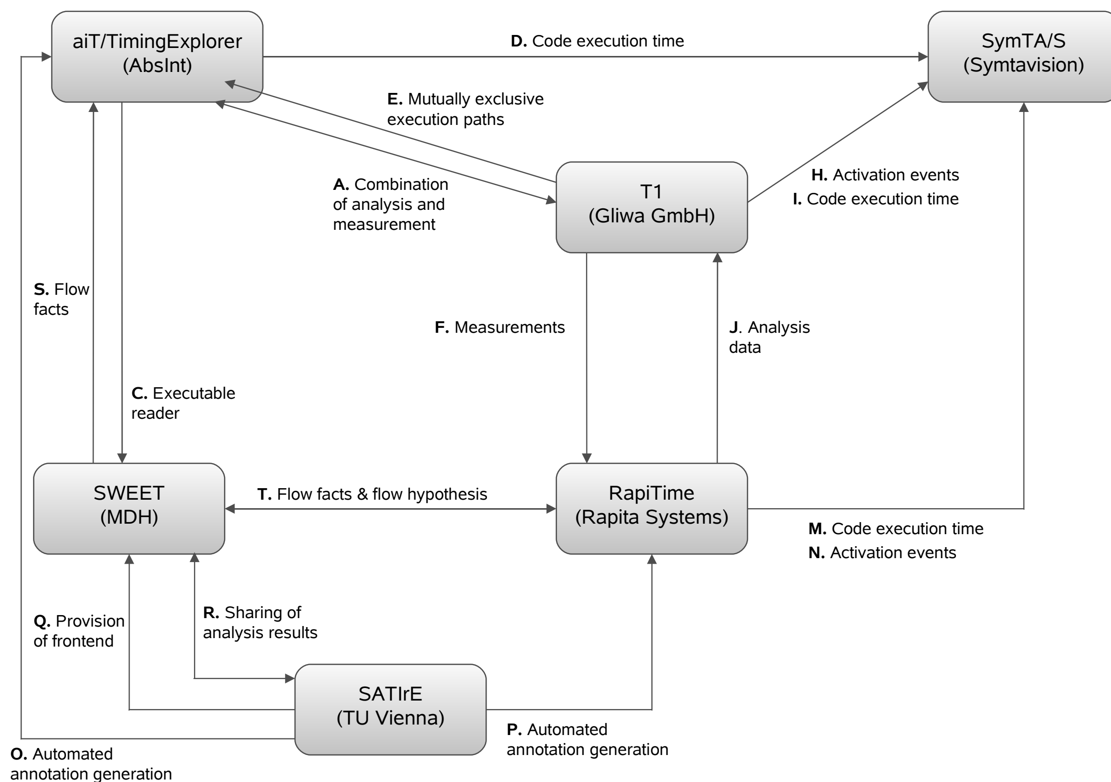

Figure 4: Tool map

3.5 Methodology Overview

Figure 5 shows the relationships of the individual ALL-TIMES tools in the ALL-TIMES methodology. Please note that depending on the specific system and specific timing question, only parts of the ALL-TIMES methodology may be needed. In Section 4, we provide a list of typical questions and “entry pointers” to the appropriate parts of the methodology.

Figure 6 gives a different picture of the tools, and tool connections, which focuses on the possible work flows and tool chains. The roles of the ALL-TIMES tools and their integrations are detailed in the following chapters, starting from the coarse-grain and moving to the fine-grain:

Use Case	Tools	Description
A.	aiT ↔ T1	Combination of analysis and measurement
C.	aiT → SWEET	Executable reader
D.	aiT → SymTA/S	Code execution time
E.	T1 → aiT	Mutually exclusive execution paths
F.	T1 → RapiTime	Measurement input
H.	T1 → SymTA/S	Activation events
I.	T1 → SymTA/S	Code execution time
J.	RapiTime → T1	Analysis data
M.	RapiTime → SymTA/S	Code execution time
N.	RapiTime → SymTA/S	Activation events
O.	SATIrE → aiT	Automated annotation generation
P.	SATIrE → RapiTime	Automated annotation generation
Q.	SATIrE → SWEET	Provision of front-end
R.	SATIrE ↔ SWEET	Sharing of analysis results
S.	SWEET → aiT	Flow facts
T.	SWEET ↔ RapiTime	Flow facts & flow hypotheses

Table 1: Summary of tool couplings

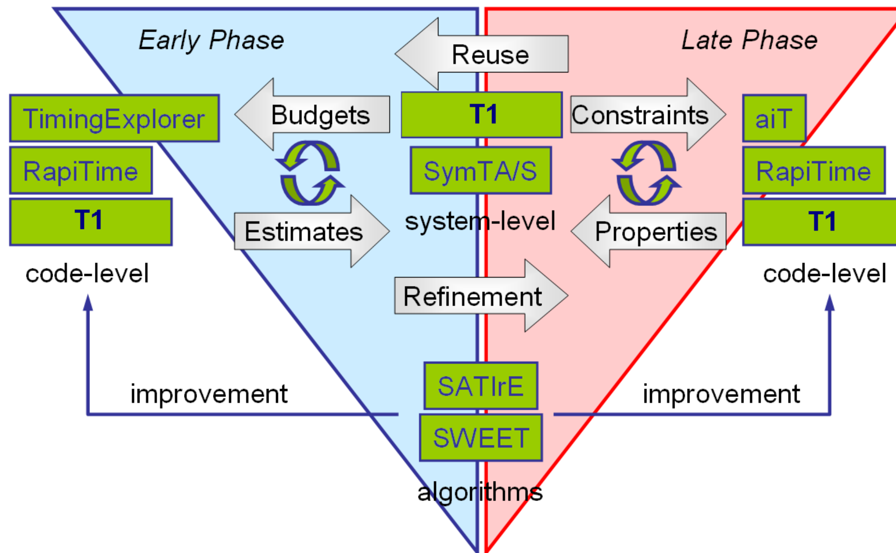

Figure 5: Relationships of the individual ALL-TIMES tools in the ALL-TIMES methodology

- Section 5 addresses system-level timing analysis and there in particular networked systems
- Section 6 addresses the combination of system-level and code-level timing analysis for single ECUs / controllers
- Section 7 addresses code-level timing analysis techniques to improve and accelerate code-level timing analysis

In most cases, we first describe the ALL-TIMES methodology for late-phase timing analysis, and then show how it can be extended to cover early phases as well.

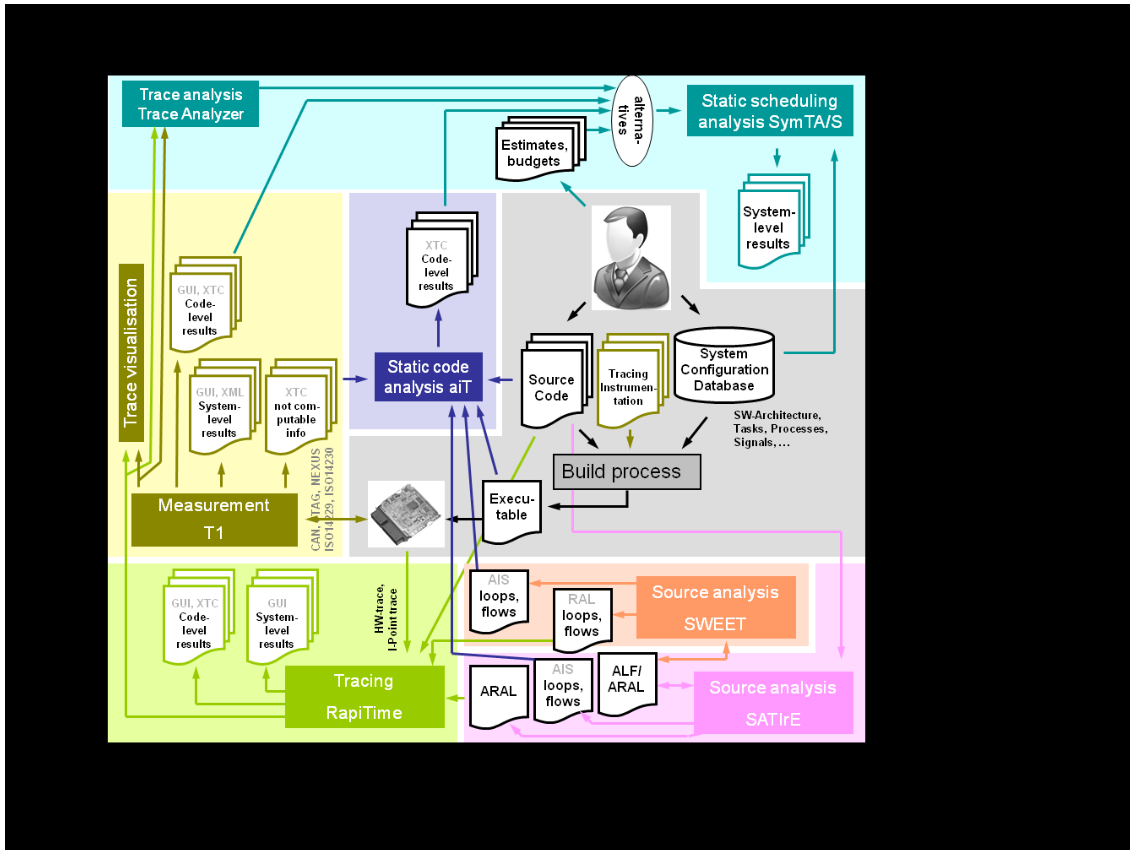

Figure 6: Tools, analysis techniques, tool connections, formats.

3.6 Alternatives between tools

As can be seen in Figure 6, alternative tools/tool-combinations exist. The choice depends primarily on the scope of a project, the development phase and safety requirements. The efficiency of a particular approach is also a factor. A short guideline for selecting the right tools is provided here. For a more profound consideration, consult the 'entry point questions´ in section 4, and the detailed discussions in sections 5, 6 and 7.

- Scope: for a network of ECUs, SymTA/S predicts, optimizes and verifies system-level timing based on static analysis. Symtavison TraceAnalyzer visualizes and analyzes system-level timing based on measurements. The combination TraceAnalyzer – SymTA/S is ideal.
- Scope: for individual ECUs, SymTA/S predicts, optimizes and verifies ECU-level timing based on static analysis. Symtavison TraceAnalyzer and Gliwa T1 visualize, analyze and help debug ECU-level timing based on measurements. The combination TraceAnalyzer – SymTA/S or T1 – SymTA/S is ideal.
- Scope: for individual software functions, aiT predicts and verifies code-level timing based on static analysis. T1 determines code-level timing based on measurements. RapiTime determines code-level timing based on a hybrid static/measurement-based approach. The choice depends on phase and safety considerations (see below.)

- Phase: *in very early development phases*, static approaches are the only alternative, since executable models needed for measurements are not available. Once prototype ECUs or other executable models such as simulators are available, static analysis and measurements are ideally used in combination. Note that the designer has to understand how accurately simulator or prototype timing reflects the timing of the target ECU.
- Phase: *defining and optimizing system configurations.* SymTA/S is suited best, because it does not require executable code and does not require re-instrumentation and re-execution of code to evaluate every change. Instead, SymTA/S provides interfaces to architecture concept tools such as PREEvision by Aquintos. SymTA/S additionally provides sensitivity analysis and design-space exploration. These automate a range of what-if analyses to determine e.g., the execution time reserves for software functions, or the optimal assignment of priorities.
- Phase: *virtual verification.* a combination of measurement and static analysis is ideal. Measurements provide execution times for typical execution scenarios. T1 can be integrated into the build process and automatically provide code execution times for every new software version. Static analysis enables the automatic calculation of worst-case code and end-to-end execution times, independent of test coverage. This avoids having to re-execute a large test-bench to achieve sufficient worst-case coverage, thus speeding up the software development process. SymTA/S can be automatically executed based on the updated measurement data to provide up-to-date information on schedulability, bottlenecks and reserves. It is possible to see the trends (e.g., a software function is using more and more of the available budget with every new version).
- Phase: *verification of the production system.* static approaches (aiT and SymTA/S) are also excellent for verification of the final system (by the same logic as virtual verification), because the techniques are inherently reliable and very fast (see safety and efficiency, below).

- Safety: Measurement-based approaches are easy to apply as a ‘side-effect´ of function-design and -testing. However, coverage and hence their safety depends on the quality of the test-cases which are executed. If test-case coverage is poor, then measurement results are not very reliable. Static approaches on the other hand look at all the possible combinations of all inputs, independent of test-case coverage. This allows to determine both worst-cases as well as statistical distributions of typical cases. Worst-case coverage means that static approaches provide timing guarantees, which are of particular value for hard real-time and safety-critical systems. Statistical coverage means that the probability of timing violations can be determined as well as the typical-case performance optimized, which is of particular value for soft real-time systems and quality-of-service considerations. Static approaches require a model of the system which means some additional up-front effort. In model-based design flows, this is reduced through importing models from system- and software-architecture modeling tools.

- Safety: Safety-critical systems employ elaborate mechanisms that bring the system into a safe state, shoud any error occur. However, if the 'error´ is merely the fact that a safety-monitoring function has not been able to report the error-free state in time (due to some worste-case timing situation), then this is an error state which can be avoided. Static analysis helps detect whether such timing errors are possible or not.

- Efficiency: The initial set-up effort for measurement-based approach is usually lower compared to static approaches. However, static approaches generally execute much faster, since they avoid the overhead for re-executing unchanged code over-and-over again to analyze integration effects.

4 Questions – Answers

This section provides a set of entry points to the ALL-TIMES methodology. It is organized along a range of timing-related questions users may ask. The table below lists these questions, and gives links to the sections in this document that provide answers to these questions. The table serves as a starting point when solving timing problems in embedded projects. Starting with the table becomes part of the methodology.

Question	Where to find answers
How can a reliable schedule be predicted when only part of the code is available (e.g. from a predecessor project)?	Use a combination of code-level timing analysis for existing code with aiT, RapiTime or T1, and timing budgeting / estimation for new code with TimingExplorer. Combine both into a SymTA/S scheduling-analysis model, and virtually verify schedulability: Section 6.4.
How can the deadline of an end-to-end timing (e.g. sensor-ECU1-gateway-ECU2-actuator) be guaranteed?	Use SymTA/S for system-level scheduling analysis / end-to-end analysis based on verified code execution times with aiT, RapiTime or T1: Section 5.1. Add network configuration and network traces to the SymTA/S system model / analysis when analysing distributed system: Section 5.3.
How can functions be allocated in a network of ECUs so that the network configuration meets the optimum with respect to timing?	Set up a system scheduling model in SymTA/S. Allocate those functions which are fixed to the respective ECUs. Manually or automatically explore the allocation of the movable functions and automatically evaluate network load and end-to-end timing for each candidate: Sections 5.2 and 5.4.
How can all deadlines be guaranteed for an existing project where code can be executed?	Use RapiTime to obtain measurement-based WCET numbers: Section 6.3.3. Use aiT to obtain statically analysed WCET numbers. A prerequisite for this is that aiT is available for the target processor: Section 6.3.4.
How can deadlines or timing-constraints be supervised at run-time?	Use T1 and define timing constraints for the timing parameters to be observed. The target code will permanently check the scheduling for any violations and can react by executing a user-defined callback or by capturing a trace of the area around the violation: Section 6.3.2.
How can CPU-load be measured at run-time?	Use T1 and define the observation time frame t_o, on which the CPU-load calculation should be based. The CPU-load is then computed by $C = \frac{t_e}{t_o}$ with t_e being the sum of execution time in t_o: Section 6.3.2.

Question	Where to find answers
How can the CPU load of a given project with processor A be estimated beforehand when ported to processor B?	TimingExplorer can be used to quickly estimate worst-case execution times for various target architectures: Section 6.4.3. This data can be requested by SymTA/S to calculate the overall CPU-load: Section 6.4.4.
How can the effects of reorganising the memory or changing the memory configuration in a given way be evaluated beforehand?	TimingExplorer can be used to quickly estimate worst-case execution times for many different hardware configurations: Section 6.4.3. This data can be requested by SymTA/S to calculate the overall CPU-load: Section 6.4.4.
What kind of corrective actions can be performed, if the actual WCET value exceeds the allocated time budget for a piece of code?	Multiple options exist, including refining analysis accuracy, optimizing code, increasing time budgets or fine-tuning the schedule: Section 6.3.5.
How is it possible to derive execution time budgets for pieces of code in early development phases?	The process starts with function-level timing constraints (typically end-to-end constraints). These have to be broken down into constraints for pieces of code taking a variety of integration effects into account: Sections 6.4.1 and 6.4.2.
In rare cases, the software of an embedded system "crashes". How can it be checked if the cause is a timing issue?	At the system-level, use T1 for triggering on the error, visualizing the timing around the point when the error occurred, see Section 6.3.2. At the code-level RapiTime's "Rewind" feature can then be used to step through the execution to determine the exact path taken through the source code.
How can suitable entry points for the code-analysis be found when a complete system is to be analysed?	The operating system configuration defines tasks and eventually also runnables which serve as good entry points: Section 7.1.
In a project which uses static code analysis already, the annotation effort for specifying loop bounds and targets of computed calls shall be reduced. How can this be achieved?	For high level of automation but possibly unsafe annotations, use trace-based loop analysis. The code analysers aiT or RapiTime can send a request for missing loop bounds and call targets to T1, which can obtain the desired answers by measurements and send a response to the static analyser: Sections 7.4 and 7.3.3. For more complex program flow constraints, and higher degree of safety, use source-level analysis. Source-code analysers like SWEET and SATIrE can be employed to automatically generate annotations in the AIS format readable by the static analysers TimingExplorer and aiT: Section 7.3.

5 System-Level Methodology

The system-level timing analysis methodology distinguishes two system scopes (see Figure 3):

1. networked systems, with emphasis on the partitioning of functions to network nodes and network configuration,
2. single controller systems, with emphasis on software scheduling and software execution.

We highlight the commonalities between the two kinds of systems in Sections 5.1 and 5.2. We then briefly discuss networked systems. However, Symtavision is the only ALL-TIMES partner that addresses networking. Therefore, timing analysis on the network level does not benefit from the integration of ALL-TIMES tools. It is included in the methodology for the sake of completeness. The main focus of this document is on single controller systems, for which the detailed methodology is described in Section 6.

5.1 System-Level Verification

The input to system-level timing verification is a model of the actual system, specifically:

- Software modules
- Communication signals
- Interconnect between software and signals (software architecture)
- CPUs
- Buses
- Interconnect between CPUs and buses (network architecture)
- Mapping of software to CPUs, CPU configuration and scheduling
- Mapping of signals to buses, bus configuration and scheduling
- Timing constraints

These traces augment the static system information with a more detailed view on the observable timing behaviour in specific situations. This can be taken into account for timing analysis, e.g., the construction of a globally worst-case timing scenario based on locally observable timing.

The resulting workflow is shown in Figure 7.

5.2 System-Level Exploration and Optimization

System-level exploration and optimization is an extension to system-level timing verification. Therefore, it uses the same two sources of input data as verification, but the model is based on a *predecessor* system. This existing model is then augmented by new ideas that the new system should implement. Since those ideas are typically available at a higher level of abstraction, a *virtual synthesis* step is necessary to create the level of detail required for timing analysis. The resulting workflow is shown in Figure 8.

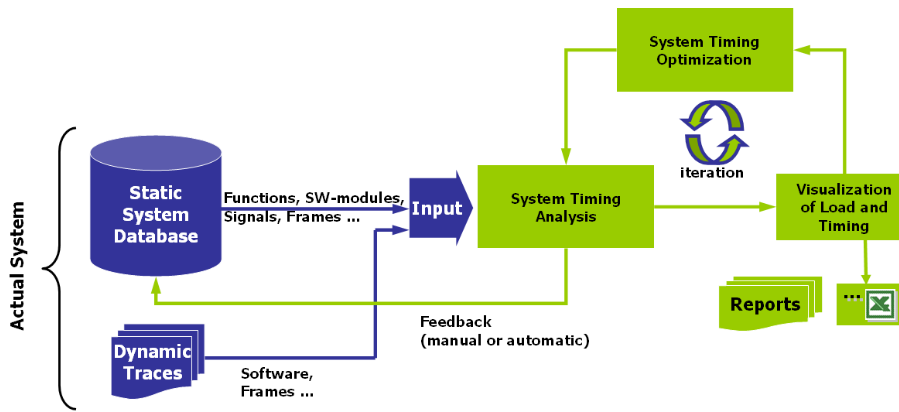

Figure 7: System-level timing verification workflow

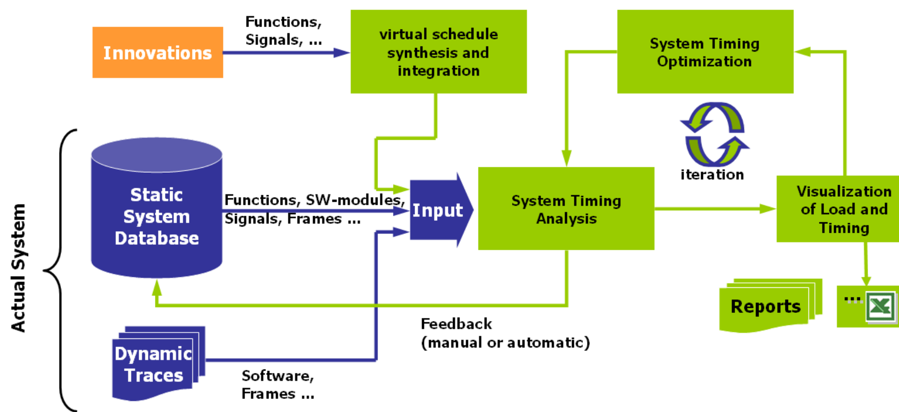

Figure 8: System-level exploration and optimization workflow

5.3 Network Verification

Network timing analysis is concerned with signals sent via network messages. The main goal is to obtain communication times over buses and networks. The ALL-TIMES tools SymTA/S considers the following for network timing analysis:

- arbitration policy of the communication protocol (CAN, FlexRay, ...)
- message and scheduling configuration (sizes, priorities, time-slots, ...)
- periods and jitter for periodic messages
- triggering models for sporadic messages

The main communication delay comes from waiting due to hand-over, synchronization or yielding to higher-priority traffic. SymTA/S analyses this delay. The actual message transmission times are both relatively short and easy to calculate from bus-protocol, bus-speed and message size. SymTA/S does this automatically.

Network timing verification can be fully automated today. The static network configuration is imported into SymTA/S via standard exchange formats, e.g., FIBEX. Additional dynamic information is optionally obtained from network traces using standard tools, e.g., CANanalyzer from Vector Informatik. SymTA/S then provides answers to key questions:

- What is the communication delay on a bus? (worst-case, typical-case, jitter)
- What is the network end-to-end delay? (worst-case, typical-case, jitter)
- Can messages get lost due to a buffer overwrite?

5.4 Network Exploration and Optimization

Once a network verification is in place, network timing exploration and optimization is also straight-forward because virtual network configuration synthesis is relatively easy. Starting point is the existing network configuration. In order to explore the optimization potential of the current configuration, scheduling parameters, in particular the mapping of signals to messages, the assignment of message priorities and offsets (CAN bus), and the assignment of messages slots (Lin bus and FlexRay bus) can be varied. This “what-if” analysis allows to characterize the room for extending an existing system.

Based on these results, it is possible to analyse the feasibility of adding additional network traffic, in order to realize additional vehicle functions. As a constraint, communication design follows the functional partitioning, which is influenced by a variety of factors other than timing. Starting from a network architecture the OEM network and systems responsibles derive the network configuration (which signals are grouped into which frame, frame cycles, etc.) according to an OEM-specific set of rules. When this is done on the virtual level, as supported by SymTA/S, then is is easy to analyse network timing for several networking alternatives. The benefit is a much better understanding of the expected network performance for a specific architecture, and where there are bottlenecks and reserves in the network. The resulting workflow is shown in Figure 9.

Figure 9: Early network exploration and optimization

6 Integrated System-Level and Code-Level Methodology

This section describes the ALL-TIMES methodology for single controller systems. It covers both system-level (integration) aspects as well as code-level (execution time) aspects (see Figure 3). We use AUTOSAR terminology: the smallest software granularity is the *runnable*, runnables are organized into *tasks*, tasks are scheduled by an operating system.

6.1 XTC: Exchange format between system-level and code-level tools

The system-level tool SymTA/S by Symtavision communicates with the code-level tools aiT and TimingExplorer by AbsInt, RapiTime by Rapita, and T1 by Gliwa via the XTC 2.0 interface (Figure 10). XTC 2.0 is also used for the tool coupling between T1 and aiT/TimingExplorer. XTC 2.0 is the ALL-TIMES evolution of XTC 1.0, which was developed in the FP6 INTEREST project as an interface between SymTA/S and aiT. XTC 2.0 is *open* and can be adopted by tools outside ALL-TIMES.

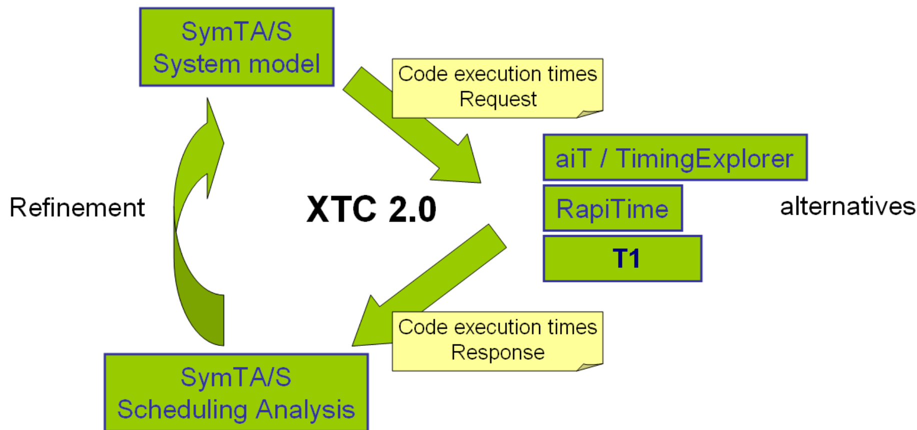

Figure 10: XTC connection between system level and code level

XTC means “eXtensible Timing Cookies”. The main idea behind these “Timing Cookies” is derived from two observations:

1. The envisioned flow between SymTA/S and the code-level tools is essentially cyclic, suggesting a request-response mechanism.
2. Each tool requires a (potentially large) set of data about the system under design, but the intersection of these data sets is small.

Timing Cookies have been introduced to avoid the duplication of the sophisticated user-interfaces available in each tool. The concept of Timing Cookies allows users to keep entering the required information in the appropriate place, and to store the information for the next round of communication between the tools. This is similar to repeatedly visiting a web site that requires certain user information. Such information is typically stored in a cookie and retrieved when the user visits the site again, hence the name “Timing Cookie”.

A Timing Cookie is an XML file consisting of two main sections:

1. One common section that describes the analysis request when the cookie is sent from a system-level to a code-level tool, and additionally holds the response to that request when the cookie is returned from the code-level tool to the system-level tool.

2. A cookie section per communicating tool to hold each tool's local information required for servicing a request and for putting the response in its appropriate context.

As shown in Fig. 10, SymTA/S launches a request to one of the code-level tools for code execution time information. This request is tagged with a unique ID and sent to the code-level tool in a Timing Cookie. If necessary, the code-level tool queries the user for additional required information. The code-level tool answers the request by sending a Timing Cookie with a response back to SymTA/S, and stores the information needed to service the request in its private part of the cookie. This code-level tool specific information is included in subsequent requests, so that the code-level tool can use the information already gathered without the need to ask the user again.

The starting point for the XTC development within ALL-TIMES was XTC 1.0, the result of the FP6 INTEREST project, which provided a simple request and response mechanism. 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, targets of computed calls, 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).

XTC 2.0 has been implemented by the partners AbsInt, Gliwa, Rapita Systems, and Symtavision. All Use Cases[2] between code-level and system-level (D, H, I, M, N) are realized via XTC 2.0.

6.2 Common trace viewer

In order to exchange timing traces between the tools, a common trace format as well as requirements regarding visualisation have been defined by the partners involved (Rapita Systems, Symtavision and Gliwa). The main benefit of this approach is the *integrated* visualization and interpretation of timing data generated by *different* timing analysis tools. Since the ALL-TIMES tools complement each other, the different aspects provided by the tools are presented to the end user in an integrated form (see Figure 11).

Gliwa and Symtavision each extended their existing viewers to support the common trace format. RapiTime can integrate with both, but focuses on the Symtavision viewer since both the Symtavision viewer and RapiTime are Eclipse-based tools written in Java. This facilitates integration.

6.3 ECU Verification

All software running on an ECU has to meet their timing constraints. Key questions that require timing verification include:

- Are all deadlines met? (worst-case, typical-case ...)

[2] See Appendix A

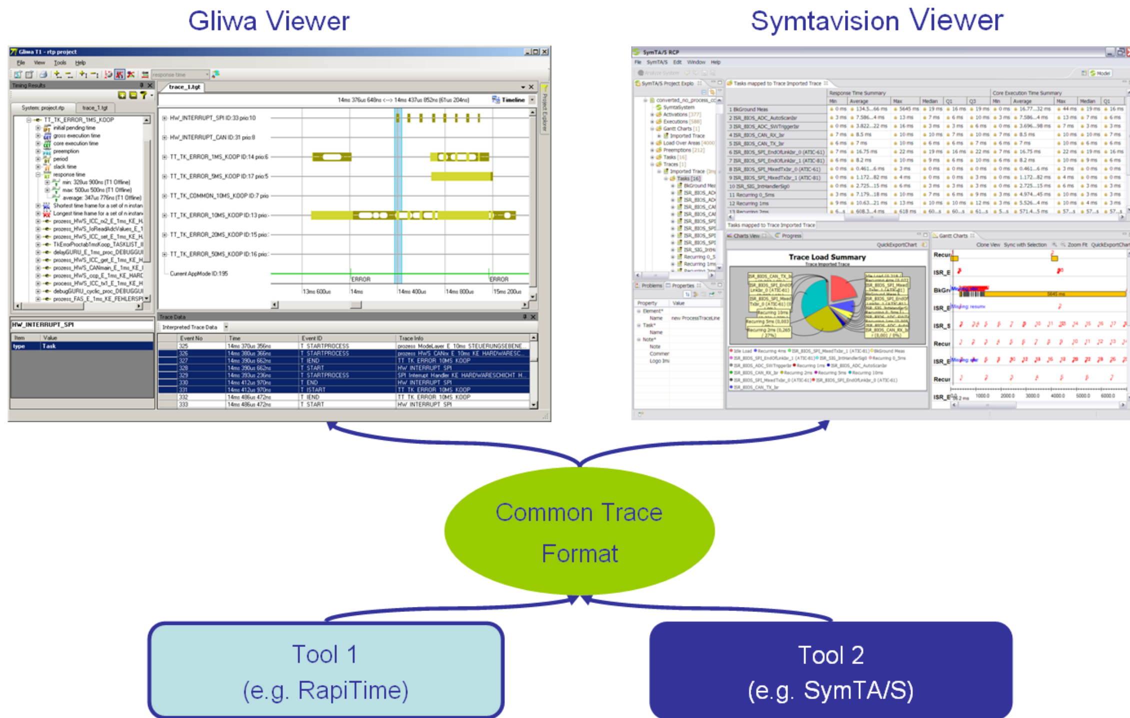

Figure 11: Common trace viewer flow and implementation by Gliwa and Symtavision

- Is there a possible data loss, if yes in which situation?

Several alternative solutions are provided by ALL-TIMES. The selection mainly depends on the required level of confidence and the effort to deploy a specific combination. The alternatives are ordered here from lower to higher confidence.

- **T1 solely**: when software and hardware is already available, this is the quickest and cheapest approach to verify system timing. Visualisation of run-time scenarios and seamless measurements of execution times on the target allow timing debugging on the one hand and supervision of timing constraints on the other. However, measurements in general require good test coverage in order to provide reliable results.

- **T1 + SymTA/S:** T1 provides code-level + full system-level measurement, visualization, and debugging. The quality and confidence of results depends on the coverage of the test vectors that were traced and measured.

  SymTA/S adds system-level scheduling analysis to tracing. Worst-case scheduling scenarios are constructed based on measurements obtained with T1 (use cases H, I). Specifically, measured runnable and task execution times as well as activating events are recombined in the worst possible combination, even if this combination was not observed during any single trace. This approach increases coverage and hence confidence in the results.

- **RapiTime + SymTA/S** or **T1/hardware tracer + TimeWeaver + SymTA/S:** these alternative combinations add code-level path analysis to measurement and scheduling analysis (use cases M, N or alternatively uses cases A, E, H, I).

Specifically, measured basic-block execution times are added along feasible paths through the code, even if some paths were not observed during any single trace. This approach further increases coverage and hence confidence in the results.

- **aiT + SymTA/S:** this combination provides full static code-level and system-level worst-case analysis (use case D), based on code-path analysis together with a detailed processor model (abstract interpretation, no measurements), combined with scheduling analysis. This approach provides very high confidence.
- **aiT + SymTA/S + SATIrE/SWEET:** here, the previous combination is further extended with a selection of source-level analyses by SATIrE and SWEET. These analyses reduce the need for error-prone manual annotations, and can thus yield even higher confidence. This approach provides the highest confidence in the coverage of all worst-case situations.

6.3.1 Scheduling Analysis

System-level timing analysis based on scheduling analysis can identify the critical corner cases for which function timing can potentially be at risk. These critical corner cases result from the resource sharing strategies employed in ECUs, and the interactions between tasks and runnables. The critical best-case and worst-case scenarios form the basis for reasoning about the data-flow latencies reaching from sensors to actuators. If the corner cases stay within the predefined time bounds, then evidence for correct implementation can be established. To perform such system-level timing analysis for verification purposes requires values for the time consuming entities within the system. In the following this is explained for the different tool combinations available in ALL-TIMES.

6.3.2 Tracing and Scheduling Analysis

Tracing allows to record the order code got executed in. The ALL-TIMES trace tools (T1 and RapiTime) record in detail *when* the code got executed. In particular, this allows to visualize the execution around the time when an error occurred. The granularity of the trace depends on the tracing method used.

RapiTime uses instruction traces produced by logic analysers, emulators and on-chip tracing mechanisms. In the following, these tools are referred to as “hardware trace tools”. They show the finest granularity down to assembly instruction level. Most also allow a function level view simplifying debugging. Some come with OS-awareness and display runnables, tasks and interrupts. For example, for OSEK operating systems there is the ORTI[3] standard bringing OS-awareness to RapiTime in an OSEK OS based project.

In comparison, T1 focuses mainly on the system view, but allows going into detail (down to C-Code instruction level) as required. T1 does not offer as many details as hardware trace tools but as an advantage connects to real “in-the-field” hardware more easily and thus allow tracing under actual operational conditions.

Independent of the tracing tool used, it is necessary to raise a trigger when an error occurs in order to get a sketch of the area around the problem. The T1 example in Figure 12 (taken from one of the case studies in the project) demonstrates how tracing and visualization help to find the cause of problems that without this technique would be very hard to find. The trigger (green vertical bar with green triangle) is positioned at a point in

[3]ORTI = **O**SEK **r**un-**t**ime **i**nterface

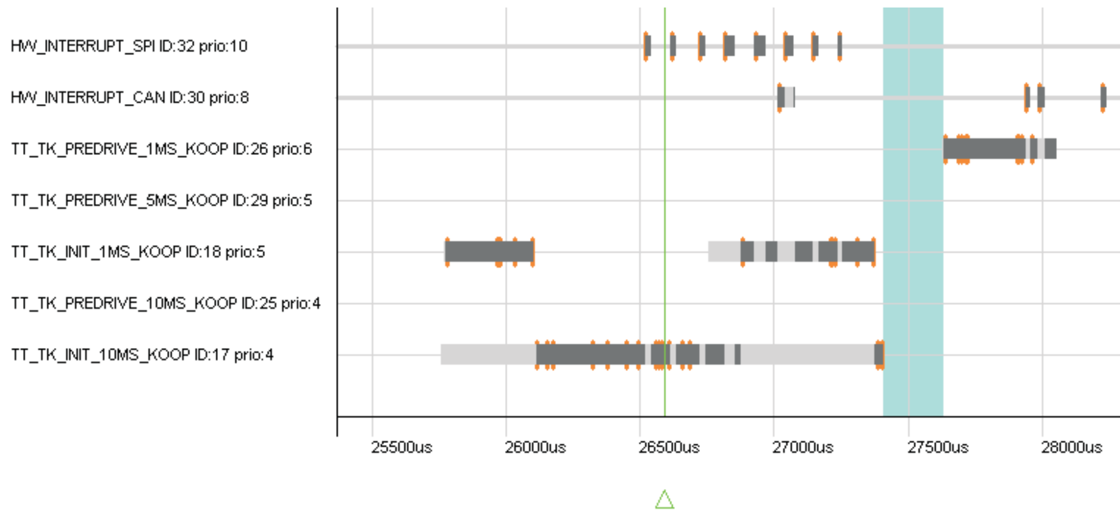

Figure 12: Example for a T1 trace

task TT_TK_INIT_10MS_KOOP where an application mode switch is initiated. This takes place as soon as no task is pending or running[4] any longer and then starts with an initial phase with interrupts disabled (blue section). In an optimised version of the software, the application mode switch sometimes led to SPI commutation problems. The reason for this can be derived from the trace shown in Figure 12. If task TT_TK_INIT_10MS_KOOP terminates earlier as in the trace, the last activation of task TT_TK_INIT_1MS_KOOP at $26.750ms$ is ignored and the blue section starts at that time instead, hitting the group of SPI interrupts (top line) and corrupting SPI communication.

We summarize, that tracing and measurement is particularly effective when a system crashes with unknown cause. In that case, the ALL-TIMES methodology checks for system timing problems by

1. making the problem reproducible,
2. making the system traceable,
3. instrument the code and/or configure the trace environment in a way that the area around the problem is triggered,
4. trace and interpret the results.

Once the measurement of data is established in a project, the next step in the ALL-TIMES methodology is static system-level scheduling analysis. As explained in Section 6.3, worst-case scheduling scenarios are constructed based on measurements from different traces. The recombination of the observed code and event timing into the worst possible combination is shown in Figure 13. The worst-case response time[5] is compared for the *10ms task* (lower part of Figure 13) against a typically measured response time (upper part of Figure 13). As can be seen the difference can be significant.

With the tool-couplings provided by the ALL-TIMES tools, the effort to combine measurement and scheduling analysis is reduced significantly. Figure 15 shows the complete tool flow.

[4]New task activations after an application mode switch request are ignored.

[5]for a definition of worst-case response time, see Appendix B

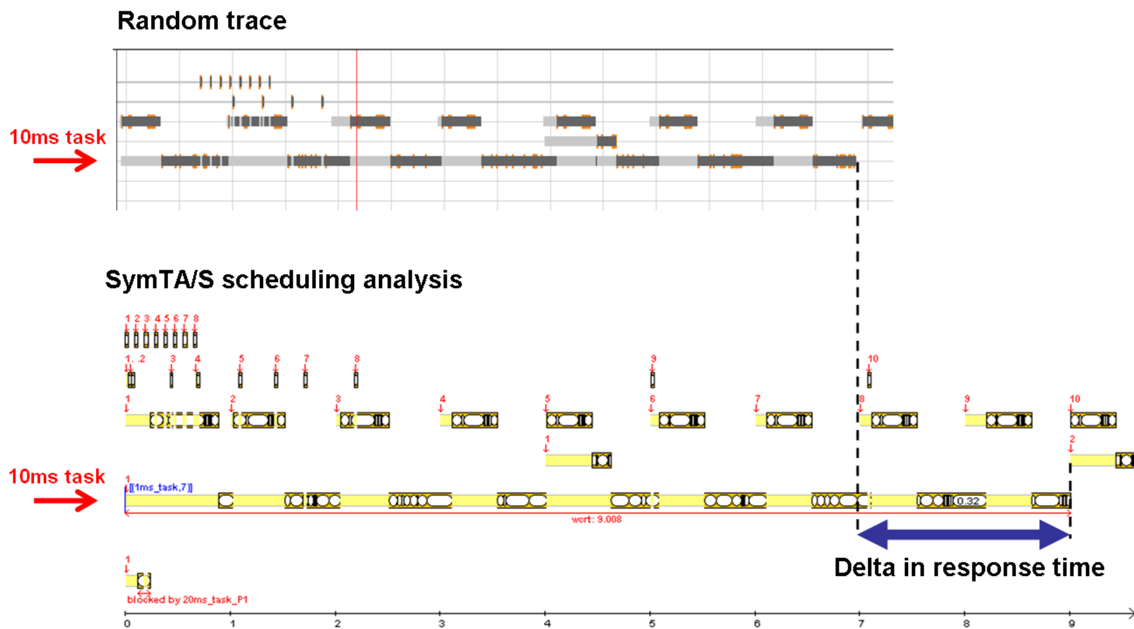

Figure 13: Comparison of measurement and scheduling-analysis

6.3.3 Tracing, Code Path Analysis and Scheduling Analysis

RapiTime allows the measurement and synthesis of the core execution time[6] of a runnable or task. By exploring all possible combinations of observed sub-paths, RapiTime can not only measure the behaviour of complete paths but also synthesis paths that could lead to longer than measured core execution times. The longest time achievable by any such synthesized path is the RapiTime estimate of the worst-case execution time (WCET). The synthesized path shows exactly how this time would be achieved. The combination of T1/hardware tracer + TimeWeaver provides essentially the same functionality.

In the same way as explained in Section 6.3.2, RapiTime or T1/hardware tracer + TimeWeaver can be combined with SymTA/S. The benefit of this combination is, that synthesizing a worst-case path through each piece of code further increases coverage and hence confidence in the results, compared to relying solely on measured execution times. The integrated ALL-TIMES methodology for RapiTime, SymTA/S and optionally SATiRE/SWEET is shown in Figure 15.

6.3.4 Static Code Analysis and Scheduling Analysis

Static code analysis provides safe upper bounds for the core execution times (WCETs)[7] of runnables and tasks. AbsInt offers the code-level WCET analysis tool aiT that determines safe upper bounds for execution times of sequential pieces of code. aiT is applied to complete executables for a fixed target architecture that is modelled exactly. As for RapiTime, SATIrE/SWEET can optionally be used to automatically derive annotations by a source-level analysis and pass the results to aiT.

When integrated into the build process, static code analysis can provide the WCETs of tasks, interrupts and runnables. The calculated WCETs are passed to SymTA/S using the

[6]for a definition of core execution time, see Appendix B
[7]for a definition of core execution time, see Appendix B

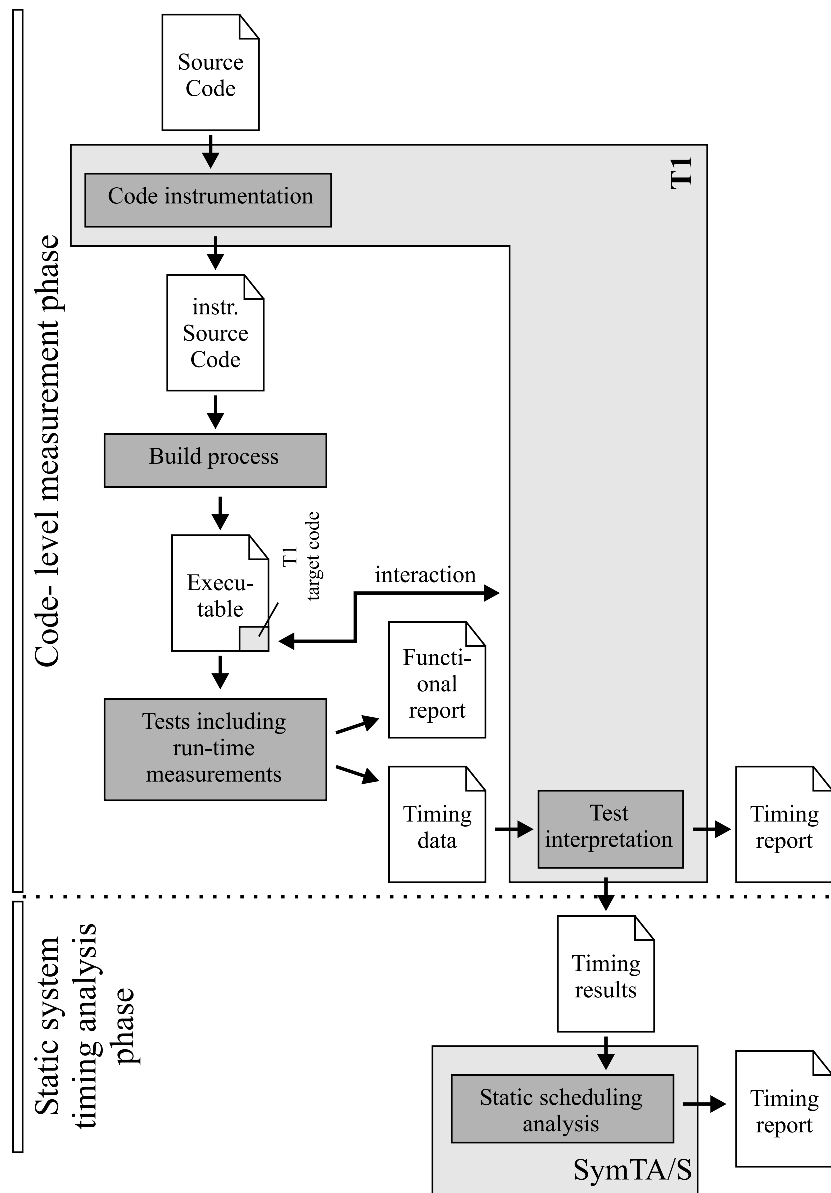

Figure 14: Measurement- and Scheduling-analysis-based system timing verification with T1 and SymTA/S

XTC mechanism (Section 6.1). Figure 16 shows the integrated ALL-TIMES methodology for aiT and SymTA/S.

6.3.5 Integrated code-level – system-level flow for ECU verification

In the verification stage of the ALL-TIMES methodology, the WCETs obtained by any of the code-level techniques are compared against the runnable's or task's budget. If the WCET value is smaller than the budget, then the verification is complete for that runnable or task, and we move on to the next piece of software, noting the "headroom" observed for this runnable or task. The headroom is the amount that the budget exceeds the WCET and is spare computation time.

Otherwise, if the WCET value exceeds the budget, corrective action is required:

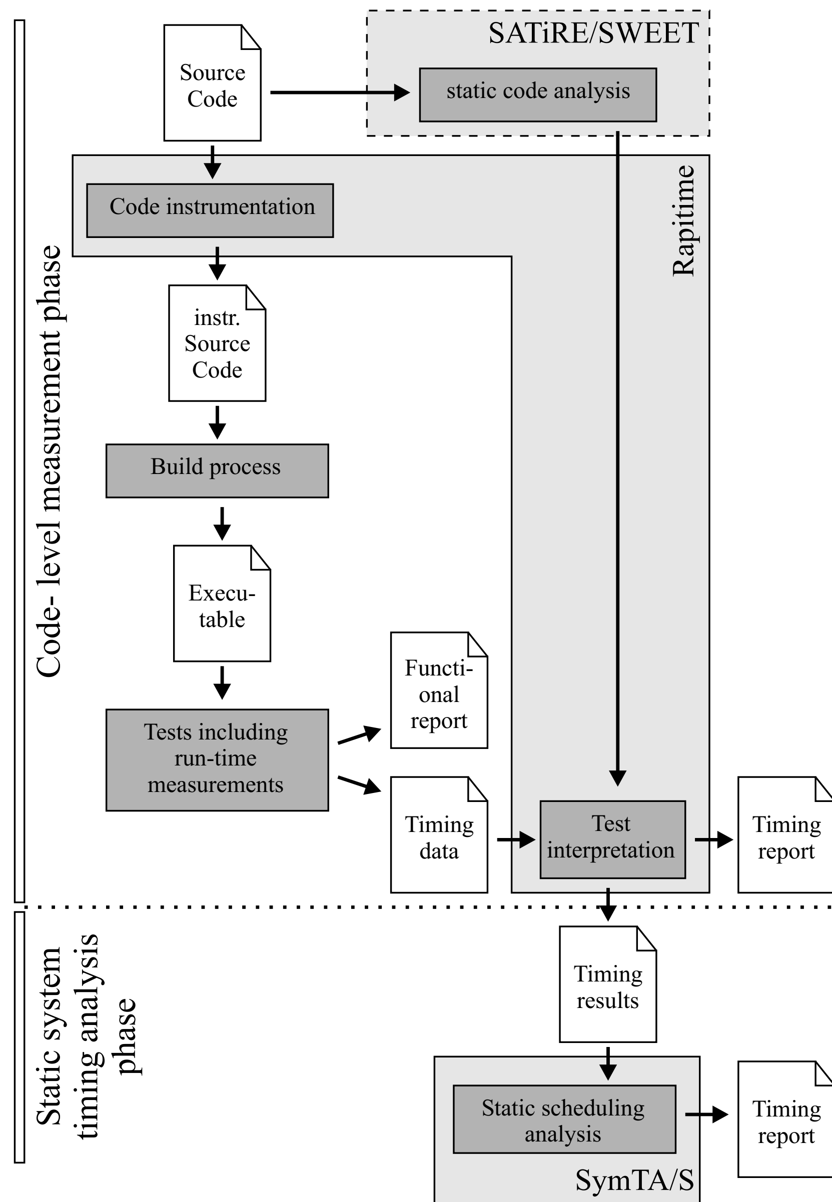

Figure 15: Measurement-based system timing verification with RapiTime and optionally SATiRE/SWEET

- If the WCET was obtained using aiT or RapiTime, then both tools offer techniques to reduce over-estimation, such as context expansion or loop count constraining. Assuming they are correctly applied, these features bring the estimated WCET down toward the true WCET, which may be less than the budget. Using SWEET and SATIrE, some of these features can be applied automatically, rather than requiring careful manual analysis of source code or even object code.

- A second approach is to actually reduce WCETs by optimising the software. Both aiT and RapiTime report identified worst-case paths. Therefore, it is very easy to identify which parts of the software consume most time on the worst-case path. This information ensures that potentially expensive optimisation efforts are only deployed where they will have maximum benefit for the WCET of the software.

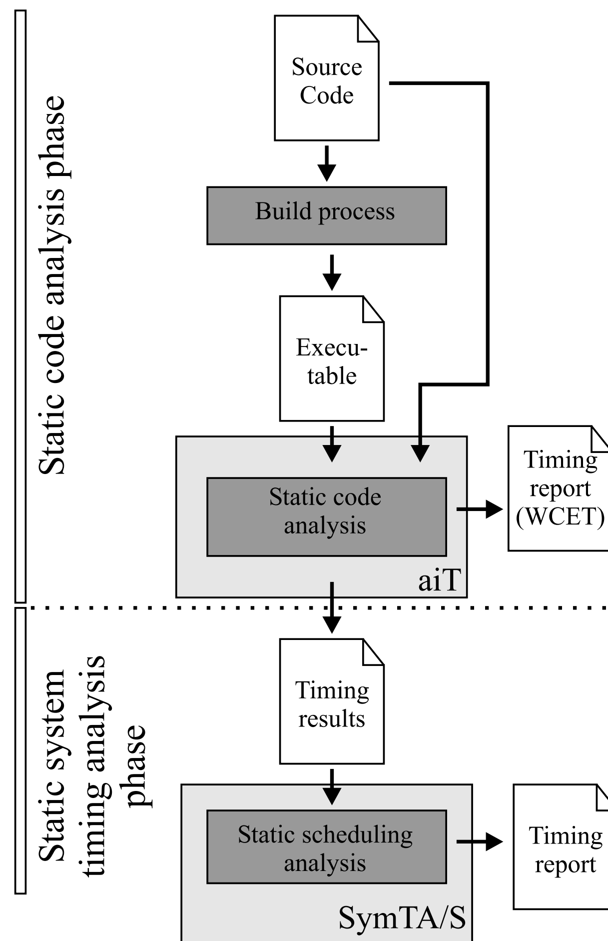

Figure 16: System timing verification based on static code analysis

- A complementary approach is to increase the budget. This is promising in particular if WCET analysis of other runnables or tasks has shown significant headroom in their budgets, meaning that the budgets can be re-balanced. SymTA/S sensitivity analysis and exploration allow such budget experiments to be performed, to determine whether or not the overall system remains schedulable.

It is sometimes suggested that adjusting the schedule and optimising the source code are activities that do not belong at the verification stage. However, the reality is that it is often cheapest to slightly modify the application or schedule and verify the correctness of the modifications, than to do a more accurate prediction in the first place (if this is feasible at all). So refinement, rescheduling and optimisation are inevitable during verification.

6.4 ECU Exploration and Optimization

When developing a real-time system controller, a key question is the optimal choice of CPU, and its configuration (cache, memory ...). As indicated in Section 5.2 the ALL-TIMES methodology combines virtual ECU configuration and scheduling to derive software execution time budgets, with execution time estimation to determine the feasibility, bottlenecks and reserves when implementing software according to those budgets.

Many of the techniques from the ALL-TIMES ECU verification methodology (see Section 6.3) can be reused also in early design phases, using predecessor systems and prototypes. The major difference in the early phase ALL-TIMES methodology is a combination of SymTA/S for virtual configuration and budgeting and TimingExplorer for estimation. If the new system is only a relatively small evolution of an existing, fully implemented system, then it is also possible to insert artificial delays into the code and estimate the feasibility of system extension through measurements using either T1 or RapiTime (see Section 6.4.5).

6.4.1 Influences on End-to-End Timing

In order to appreciate the ALL-TIMES early-stage methodology, it is necessary to understand the various influences on timing. Figure 17 shows the major timing effects contributing to end-to-end delay on an ECU along a signal path.

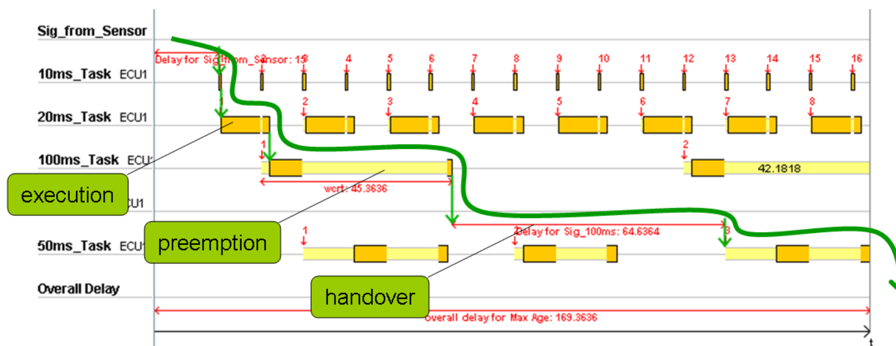

Figure 17: Different timing-effects contributing to end-to-end delay on an ECU

In the following, these main timing parameters are explained, together with an assessment of their availability and quality in an early design phase.

Runnable execution rates : runnable execution rates are obtained from controller design, even automatically. For example in the FP7 INTEREST project, execution rates were automatically extracted from ASCET-MD into a SymTA/S system model.

Data flow : data flow between runnables is obtained from controller design in the same step as runnable execution rates.

Preemption and blocking : preemption and blocking times depend on integration of multiple runnables and tasks. Therefore, prediction requires virtual integration.

Data handover between runnables, especially in multirate systems : as with preemption and blocking, the time needed for data handover (when the data is waiting in memory) can only be answered when considering the integration of runnables and tasks. Therefore, prediction again requires virtual integration.

Runnable execution times : runnable execution times to be estimated. This is difficult since they heavily depend on parameters which are not known in early design stages: the detailed implementation of an algorithm, optimizations performed by the compiler, or the allocation of software to specific memory locations (which impacts cache performance). On the upside, the specifics of candidate processors (in particular, pipeline and cache/memory architecture) are typically well-documented.

In summary, the function architecture (runnable execution rates and data flow) can be described in detail early in the design, and often automatically extracted from model-based design tools. On the other hand, runnable execution times and timing effects stemming form integration are difficult to describe in detail in early design phases, because the code is not yet written and integrated on the final target.

The ALL-TIMES methodology focuses on the dimensioning and configuration of an ECU such that it performs reliably under *worst-case* conditions. In the automotive domain, very often the concern is raised that 'worst-case dimensioning' yields a system which is too expensive. This is a misconception. A worst-case analysis is performed for specific scenarios, for which the system definitely has to work.

1. A safety critical system must be guaranteed to reach a safe state before a specific deadline in case of a fault.
2. Object recognition should work reliably for a minimum number of objects.
3. Some deadline overruns in engine control are acceptable but those leading to ECU reset must be avoided.

Of course, the system may perform better in the average case, but such a question is of lesser importance for dimensioning and configuration. The corner-cases are critical. The ALL-TIMES methodology thus focuses on the combination of static analyses: SymTA/S on the system-level, and TimingExplorer on the code level. Both tools can be augmented to yield additional estimates for the distribution of timing. A second advantage of this combination (in particular compared to simulation) is the independence from an executable system prototype or from a good test-suite.

6.4.2 Execution Time Budgeting using Scheduling Analysis

The starting point is a function architecture which has to be implemented on an ECU. An initial schedule can be generated based on the function periods and data-flow. Standard function modelling tools, e.g., ASCET-MD provide this synthesis capability. The initial system configuration is then imported into SymTA/S. As far as available from previous systems, execution times for existing runnables are imported. At this point, it is possible to perform exploration for a *reference* CPU already known from a previous design, to see the impact of additional runnables. At this stage, the following degrees of freedom can be explored:

1. change the execution time for the new runnables
2. change the ordering of runnables and allocation of runnables to tasks

3. in case of a multiprocessor controller or multicore CPUs, change the allocation of runnables to CPUs / CPU cores

The changing of execution time needs automation support, since the number of possible values is very large and manual exploration would be inefficient. SymTA/S uses evolutionary optimisation and sensitivity analysis for this purpose. The other changes can be performed manually, because the reordering or remapping of runnables usually is severely constrained. An engineer will therefore consider such an option very carefully. SymTA/S makes it easy to execute each change, once the decision to explore it has been made.

In a second stage, the impact of changing the ECU can be explored. SymTA/S supports an automatic speed-up / slow-down exploration of the CPU speed, which linearly reduces respectively increases the execution time of each runnable. The qualitative effect of exploration at this level is shown in Figure 18.

Figure 18: Exploration of different CPU speeds, configurations and executions times with different end-to-end timing results

Once the appropriate configuration has been determined, the resulting execution time budgets for each task and runnable are derived. Starting from the overall budget for the path (see Figure 17) we derive the latest points in time when a handover of data between different tasks or runnables on the path has to occur. This gives us the maximum permissible response time for each task or runnable on this path. This is shown as the first step in Figure 19. From this and the worst-case preemption scenario, the maximum permissible execution time of each task is derived. This yields the execution time budget. It is shown as the second step in Figure 19.

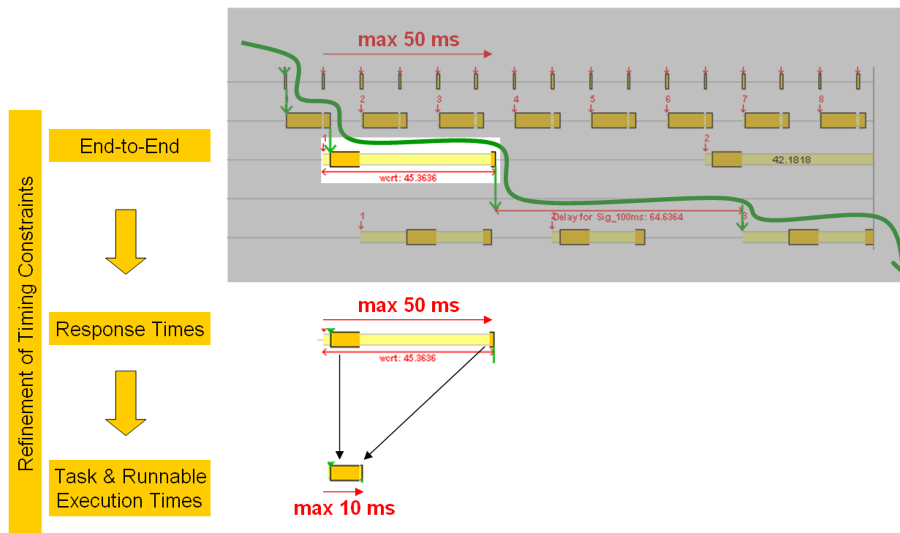

Figure 19: Deriving runnable budgets from virtual schedules

One aspect that is often overlooked but becomes obvious from Figure 17 is the relatively small portion of the path delay that is spent on executing tasks along the path. Delays due to preemption or due to handover have a much larger influence. This is a key reason why the budgeting approach described works, and why the detailed execution time of code can be addressed later.

6.4.3 Execution Time Estimation Using Static Analysis

AbsInt has developed the TimingExplorer as a complement to aiT to facilitate the computation of WCET estimates for a wide range of target architectures without requiring exact models for them.

Execution time estimation using TimingExplorer can be started in early design phases once (representative) source code of (representative) parts of the application is available. This code can come from previous releases of a product or can be generated from a model within a rapid prototyping development environment.

As a first step, source-code analyses are performed (see Section 7.2) with SATIrE and SWEET. These analyses produce additional input for the actual timing analysis with TimingExplorer (see Section 7.3). Since the goal is to assess how the timing behaviour of the code differs on different hardware, TimingExplorer (like aiT) operates on binary executables. Hence, the available source code has to be compiled and linked using representative standard compilers for each of the potential target processors. The resulting fully linked executables are the main input of TimingExplorer, but TimingExplorer also reads the annotations generated by the source-code analysers and the source code itself to be able to refer to source code in its user interactions. TimingExplorer also reads a hardware description detailing the target architecture (memory layout, cache properties, etc.). By running TimingExplorer several times with different hardware descriptions and comparing the resulting execution times, the most appropriate configuration can be chosen.

Together with the source-code analysers, TimingExplorer produces estimations for the non-preempted worst-case execution times of runnables and tasks. These are returned to SymTA/S for an integrated methodology (see next section).

Source-code analysis is very natural to use in early design phases. In addition to the usual benefits of having a more automated analysis, replacing potentially unsafe manual annotations and reducing the effort of producing these, a source-code analysis does not have to be repeated as long as the source code is not changed. Thus, the analysis results can be reused also in the case that the design exploration considers target architectures with different instruction sets, which is not possible for analyses on binary level. Furthermore, source-code analysis can be applied even before the code is compiled for a certain target. This enables very early pinpointing of potential problems. For instance, a source-level loop bounds analysis can be used to identify loops that may have problems with high iteration bounds, or even might not terminate.

6.4.4 Combining ECU budgeting and estimation

The ALL-TIMES methodology for early-stage ECU exploration is a combination of the tool flows described in Sections 6.4.1, 6.4.2 and 6.4.3. It is shown in Figure 20.

The starting point is a SymTA/S model of the existing ECU configuration and ECU timing information. The ECU timing information has been obtained from a predecessor

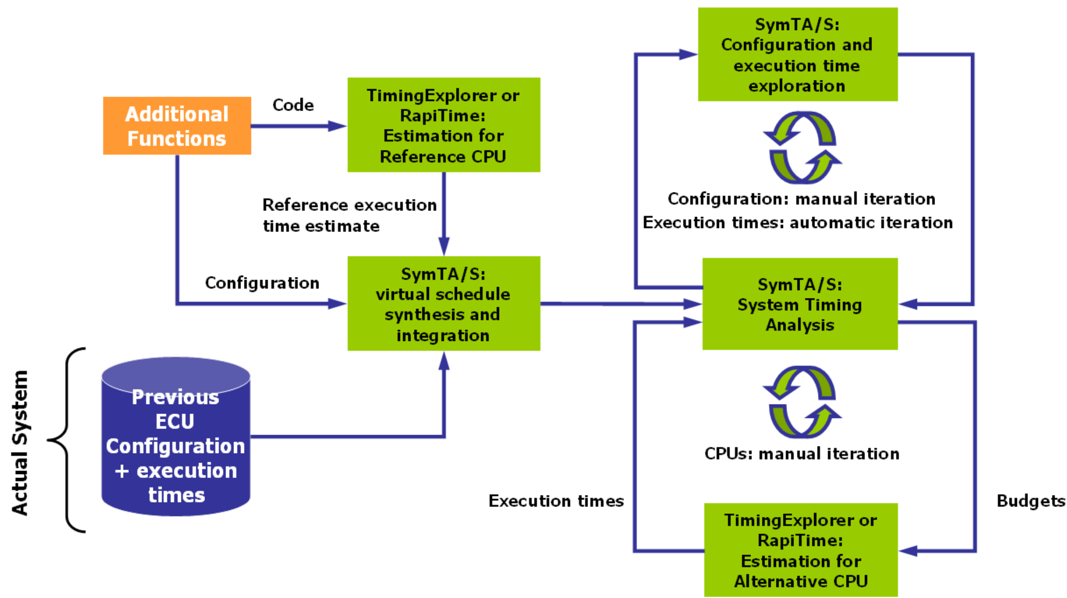

Figure 20: combination of system-level and code-level for ECU exploration and optimization

system using the combination of tools described in Section 5.1. This initial model is augmented by the structural information of the new functions. An initial estimation of the execution time of these functions is done for the existing ECU, either using an ECU model (aiT) or the actual ECU (T1 or RapiTime).

The actual exploration is an iterative process. We distinguish between two iteration loops: one (upper in Figure 20) where the CPU is *not changed*, the second (lower in Figure 20) where the CPU is *changed*).

Each single iteration consists of two steps:

1. make a change to the existing system model
2. analyse the timing of the changed system

The next iteration then takes the timing analysis results obtained so far as a starting point for further changes. The exploration of CPU speeds and code execution times is automated in SymTA/S. The exploration of CPU configuration and CPU type is manual in SymTA/S and TimingExplorer.

Once the best ECU configuration has been selected, ECU implementation can start. SymTA/S sensitivity analysis shows the bottlenecks and reserves of the selected configuration. As implementation progresses, code execution times can be continuously checked against budgets. If there is a deviation in particular in a “risky” part of the design, then special effort has to be expended on this area. When necessary, the original budgets can be shifted from the remaining reserves (see Section 6.3.5). This methodology guarantees that bottlenecks are detected and can be solved as early as possible in the development.

6.4.5 Early Tracing and Scheduling Analysis

If a version of the software is available that can link and be executed, even if not on the final target hardware but on a simulator or alternative hardware, RapiTime can be used to

analyse measured and synthetic worst-case execution time (WCET) values as described in Section 6.3.3. In combination with SymTA/S, the rescheduling and optimisation options, which are relatively limited at the verification stage, are much more likely to be available at earlier phases of development.

In addition, RapiTime can also be used when there is no full, executable version of the software. Incomplete parts of the software can be replaced by "stub" functions, to which the RapiTime user can attribute any chosen WCET values. This allows a RapiTime WCET analysis to be done with a combination of actual and estimated WCET values. As the software is completed, stubs can be replaced with actual code, improving WCET estimate. But the user has at least a very good chance to detect timing problems before the software is finished.

Where estimates are required for incomplete software, the estimation process can be improved by performing a RapiTime analysis on any earlier versions of the software on any earlier versions of hardware. Such analysis profiles the worst-case path, helping the user to determine approximately what proportions of the worst-case path computation are data copying, integer arithmetic, floating point operations, input/output and so on. Knowing such characteristics of the worst-case path assists in estimating the performance with a new compiler or with new hardware, for example. If some new hardware doubles floating point performance but leaves data copying unchanged and if the worst-case path involves 50% floating point and 50% data copying then we can estimate that the net improvement will be about 33%.

In addition to other traditional sources of estimates, the WCET estimates might come from using RapiTime on unit tests for software that is not yet complete enough for integration testing.

7 Code-Level Methodology

In the area of code-level analysis, we distinguish between analysis of source code (Section 7.2), static timing analysis on the binary level (Section 7.3), and timing analysis via tracing (Section 7.4). Source-code analysis is also static. It is described separately since its results serve as input to both static timing analysis and timing analysis via tracing.

7.1 Finding reasonable entry points

Most code-level analyses are executed on code which is part of an embedded *system* with all its system-level characteristics (preemptions, mutually exclusiveness etc.). Since the pure code-level analysis tools cannot handle these, it is simply not possible to (code-level) analyse a whole system by starting a code-level analysis at the reset vector or the main function. The question arises, where to start the code-level analysis or in other words: what are reasonable entry points for the code-level analysis.

In most cases, a system uses an embedded operating system which organises the code to be scheduled in tasks and interrupts. These typically work as a kind of container for smaller grained scheduling objects. Depending on the operating system used, these are functions, processes or runnables. In the following, the smaller grained scheduling objects will be referred to as "runnables". The runnables of a task get executed one after the other when the task is started. Thus all tasks and interrupts as well as the runnables serve as meaningful starting points for code-level analysis. A more detailed analysis can also consider the next level, the functions and an even more detailed analysis might start

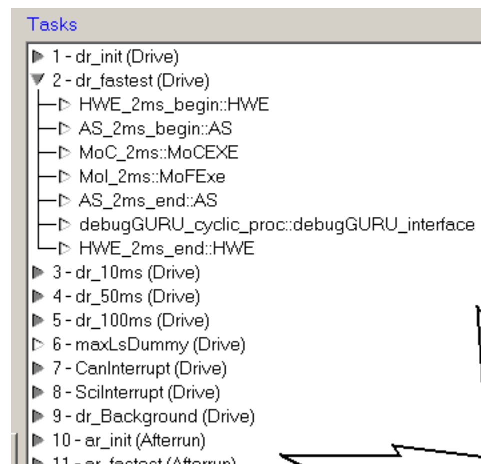

Figure 21: The VECU OS-configuration as an example for the link between tasks and runnables

at the border of basic blocks. In fact, all code-level items shown in Figure 3 down to the level of basic blocks are reasonable entry points for code-level analyses.

The operating system configuration defines tasks and – for some operating systems – which runnables belong to each task. See Figure 21 for an example of an operating system of this kind.

For other operating systems, the implementations of the tasks form the link between tasks and runnables. The following code example taken from an automotive application demonstrates such a link.

```
TASK(osTask5ms)
{
    CanRxBasicCANTask();
    CanRxFullCANTask();
    CanWakeUpTask();
    CanErrorTask();
    klProcessCanTermCommands();
    azApplAzg();
    cnCheckCanMessages();
    TerminateTask();
}
```

The methodology for finding meaningful entry points for code-level analysis derived from these thoughts is this:

1. the top level entry points for code-level analysis are tasks. These can be taken from the operating system configuration.

2. the second level entry points for code-level analysis are runnables (if applicable for the operating system used). These too can be taken from the operating system configuration.

3. In case the operating system does not support runnables, the definition of the tasks may provide simple function calls to functions which can be treated as runnables.

7.2 Source-Code Analysis

The tools for timing analysis, which operate on the binary level by static analysis (Section 7.3) or tracing (Section 7.4), can profit from information obtained by source-code analysis since certain kinds of analysis information are easier to derive from source code than from binary code. In particular, analyses on the source-code level have access to all the program variables introduced by the programmer, as well as to expressions that refer to structure members, array elements etc. Identifying and naming objects can therefore be much simpler on the source level than on the binary level where many objects are identified only by hardware register numbers or memory address computations.

Two other examples where a source-level analysis can avoid problems that appear on the binary level are switch statements, and 32-bit arithmetic compiled down to a 16 bit instruction set. On the source level, analysis of such statements will typically pose no problems at all. However, the compiled binary can be very hard to analyse. For instance, some embedded compilers that optimise for space will compile switch statements into intricate jump table structures. If 32-bit arithmetic is compiled down to 16-bit operations, then typically the original arithmetical operations will be obfuscated into a sequence of 16-bit operations, making it very hard to identify, e.g., loop parameters. For both these exanples, it can be very hard to derive any sensible information at all on the binary level.

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. 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.

Within the ALL-TIMES project, SWEET and SATIrE are used for source-code analysis. We now mention the source level analyses implemented within ALL-TIMES, and we exemplify the usage of the different analyses.

A basic ingredient of SATIrE's source-code analysis is a points-to analysis that among other things determines the possible values of function pointers. Given a C program, the analysis computes for every expression that refers to a memory location or function a set of possible objects or functions that may be referenced by that expression. The computed set of possible targets will always include all the targets that can actually be referred to, but may include additional targets that actual executions of the program will never reference.

Constraining the possible values of function pointers is very important to obtain a tight WCET analysis, since such constraints bound the sets of possible functions that might be called in different program points. Obviously, if the analysis can exclude functions with long execution times then tighter WCET estimates will result.

The interval analysis integrated in SATIrE derives information on possible values of integer variables in C programs in the form of lower bounds and upper bounds, i.e., intervals, which may be open on either side (lower bound $-\infty$ or upper bound ∞). The analysis is a flow-sensitive data-flow analysis implemented with SATIrE and PAG. It is integrated with SATIrE's points-to analysis so that integer assignments or reads through pointers can be resolved to sets of possibly referenced variables. At the end, each program point is associated with a mapping from variables to intervals.

As the analysis evaluates branch and loop conditions to derive more precise information, it sometimes finds conditions that are always true or always false (in some context). In

these cases, one of the outgoing paths is unreachable and is marked by the analyser as such. The results of an interval analysis can also be used for many other purposes, like for instance restricting the values of numerical pointers. This information is interesting in systems with different memories that have different access times. If a memory access can be surely predicted always to go to a fast memory, then it will be possible to calculate a tighter WCET estimate.

When calculating worst-case execution times, analysis of loop bounds plays an important role. A calculated loop bound should be as close as possible to the real loop bound, but never below it. SATIrE includes a component that computes loop bounds for loops based on iteration variables. It uses results of the interval analysis and structural information about the program to build a set of inequalities. Solving these inequalities yields constraints on the numbers of loop iterations.

SWEET does not itself read C or C++ source code, but uses SATIrE's C/C++ front ends for this task. SATIrE has been extended with a back end that produces ALF code from its internal representation of C programs; ALF is the intermediate language of SWEET. All analyses performed by SWEET are in fact based on the ALF representation of the program.

Loop bounds and other flow constraints are SWEET's speciality. SWEET can perform flow analysis of both source-level and binary level code through translations into its input format ALF, although for the reasons mentioned above, the precision is expected to be higher for source-level analysis. SWEET's flow analysis can produce a number of program flow constraints ranging from loop bounds to intricate infeasible path constraints. The flow analysis algorithm of SWEET is designed to give high precision at the cost of potentially long analysis times. This can be contrasted with the loop bounds analysis of SATIrE, which on average can be expected to be faster but also less precise.

7.3 Source-Code Analysis and Static Timing Analysis

Static timing analysis typically is used for calculating the WCET of tasks, interrupts or functions. It can be performed using the AbsInt tools TimingExplorer in early phases (Section 7.3.1) and aiT in late phases (Section 7.3.2).

To successfully analyse a task, these tools need information about the number of loop iterations, recursion bounds, targets of computed calls and branches. The precision of the analysis can be improved if infeasible execution paths, i.e., paths that are never taken, are recognized. Partially such information can be automatically discovered when analysing the binary. For instance, the tools contain a loop-bound analysis on binary level that can discover bounds for the number of iterations of simple loops. However information that cannot be discovered automatically has to be supplied to the tool in the form of annotations. To reduce the need of manual annotation and to increase the degree of automation, SWEET and SATIrE can be employed to obtain the necessary information by source-code analysis.

The high-level information found by SWEET and SATIrE is passed to the timing analysers via source-code annotations, using the respective annotation formats of the tools. Below, we show some annotations in aiT's "AIS" format that have been generated by SWEET:

```
FLOW SUM ("edgebi.c_38_3") + ("edgebi.c_50_3") >= 16 ;
FLOW SUM ("atc_recupprotocolimpl.c_65_5")
```

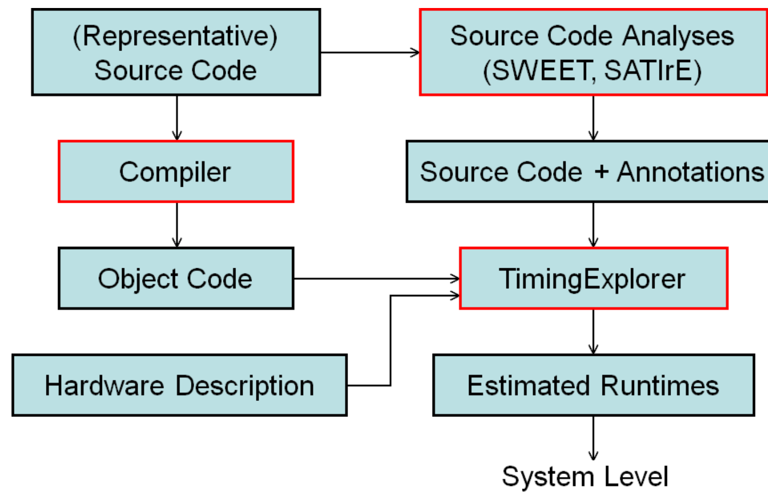

Figure 22: Workflow for source-code analysis and static timing analysis

```
         <= 1("ATC_RECUPPROTOCOLIMPL_IMPL_calc") ;
FLOW SUM ("atc_recupprotocolimpl.c_65_5") + ("atc_recupprotocolimpl.c_68_3")
         <= 2("ATC_RECUPPROTOCOLIMPL_IMPL_calc") ;
```

These annotations express constraints on how many times different program points can be executed together. An additional advantage of generating source-level annotations is that they can be inspected by the programmer, who can relate them to the source code.

7.3.1 Early-phase Source-Code Analysis and Static Timing Analysis

The workflow for the combination of source-code analysis and static timing analysis in early design phases is shown in Figure 22. The process can be started once (representative) source code of (representative) parts of the application is available. This code can come from previous releases of a product or can be generated from a model within a rapid prototyping development environment.

As a first step, source-code analyses are performed using SWEET and SATIrE. These analyses do not depend on the target architecture. They can be performed before the actual architecture exploration. The results of source-code analysis are used to generate annotations that serve as input for TimingExplorer. If trace data is available, then it is also possible to generate program flow constraints from the traces as described in Section 7.4.1. The accuracy of these constraints will depend on the quality of the test vectors, but for early phase timing analysis, the reduced confidence may still be sufficient.

It is the job of AbsInt's TimingExplorer to analyse the timing behaviour of the code and to produce an estimate for its run-time on the architectures of interest. Since its goal is to assess how the timing behaviour of the code differs on different hardware, the tool operates on binary executables. Hence, the available source code has to be compiled and linked using representative standard compilers for each of the cores considered as potential target processors. The resulting fully linked executables are the main input of TimingExplorer, but TimingExplorer also reads the annotations generated by the source-code analysers and the source code itself to be able to refer to source code in its user interactions. TimingExplorer further reads a hardware description detailing the target architecture (memory layout, cache properties, etc.). By running TimingExplorer several

times with different hardware descriptions and comparing the resulting execution times, the most appropriate configuration can be chosen. The computed WCET results also serve as input to system-level analysis.

7.3.2 Late-Phase Source-Code Analysis and Static Timing Analysis

The methodology for late-phase source-code and static timing analysis is similar to that for early-phase analysis presented above. The main difference is that architecture exploration with TimingExplorer is replaced by a timing validation with the aiT tool. In contrast to TimingExplorer, the aiT tool can produce a provable upper bound of the real WCET (and needs much more resources in terms of analysis space and time for this).

A precondition for the correctness of aiT's results is the correctness of the annotations given to it. Since a compiler may rearrange the code, e.g., by unwinding loops, the results of source-code analysis cannot be simply taken over, but require a manual validation if aiT's results are to be used in a rigorous validation process.

7.3.3 The "Simple Annotation" Approach for Static Code Analysis

Between architecture exploration in early phases and timing validation in late phases, static timing analysis can also be used to monitor the temporal behaviour during the evolution of a system. There is an approach with little annotation effort and still high practical value for this.

If a static code analyser calculating WCETs is integrated into a build process and thus provides WCETs for selected functions with each build, it becomes very simple to compare execution times of different versions of the software. The absolute values of the calculated WCETs are not relevant for this approach; they only serve as a benchmark during the development and the evolution of the project. The closer the calculated WCET is to the real WCET, the better the benchmarks, of course. With fewer annotations, the overestimation increases. But if a function's calculated WCET results e.g. in around 220 % of the intended WCET for several versions of the software and suddenly reaches 480 % for a new version, the process described gives a valuable hint that there was a change in the code that very likely has a great influence on the execution time. This works even with few annotations and once integrated into the build process serves as a very good "alarm system" for sudden increases in time consumption.

7.4 Source-Code Analysis and Tracing

7.4.1 Obtaining annotation information by tracing

When installing static code analysis tools in existing projects, the analysis typically runs into a number of "dead ends" where some loop bounds or call targets of indirect function calls cannot be statically computed. A source-level analysis typically helps to find some missing constraints, but often cannot find them all. Generally speaking, there are two ways to solve this problem:

- manually provide the missing information, i.e. add hand coded annotations
- execute the code and observe, i.e. trace or measure loop bounds and/or call targets

Manual annotation is very time-consuming and especially in larger projects (like for automotive ECUs), it often is very difficult to get hold of the experts being able to state

the details required for the annotation. It may be almost impossible in case the relevant code is embedded in a library delivered by another company like e.g. an operating system vendor. The main advantage of manual annotation is the independence of hardware and any other tool.

Observing the code while it executes and simply measure loop bounds and call targets is a very efficient approach once the measurement infrastructure is applied. The XTC extension developed within ALL-TIMES allows fully automated communication between the static code analysis tool requesting information and the measurement tool providing information. Based on the reply, the static analysis tool can annotate the code accordingly and continue the analysis at the point it previously ran into a dead end. Any dead ends that follow can be solved by another request/response, the next iteration, and so forth.

There is one thing that is very important to keep in mind when using this approach. Since the annotations are based on measurements, they are as good as the measurements are. And the quality of the measurements is linked to the quality of the test cases which were executed during the measurement. A small set of test vectors e.g. might lead to the execution of only a subset of application features and this might in turn result in only a subset of possible call targets being executed during the measurement. As a result, the static analysis might miss the *real* WCET path in the software.

Measurement based static analysis is a very powerful technique and very quickly leads to WCET results. In order to get reliable results, the following must be done:

- implement powerful tests which maximise code coverage in order to capture all call targets and maximise the data variety in order to capture the real upper loop bounds
- review the results, i.e. the automatically generated annotations
- keep in mind that the analysis no longer can *guarantee* to provide the worst case

AbsInt's aiT and Gliwa's T1 support XTC and the ALL-TIMES industrial case-study "VECU" can be completely[8] analysed in about two hours time on a standard PC.

7.4.2 Late-phase Source-code event analysis and tracing

Tracing system-level related events like start/end of the tasks, interrupts and runnables allows to reconstruct a run-time situation and understand what happened. A trace shows which tasks, interrupts and runnables executed at what time, took how long and consumed which amount of run-time. If all tasks and interrupts are traced, it becomes possible to calculate the core-execution times of each occurrence of a task and interrupt simply by subtracting the core-execution times of all preemptions for the occurrence in question. In this case, tracing allows to measure all timing properties listed in appendix B on page 48.

When extended by used defined events, traces can also incorporate the relation from application-functionality and/or -data to the timing of the application.

The effort to integrate source-code event analysis and tracing into application is relatively low (approx. 1 day up to a week) and should be done in a way that changes to the system configuration – e.g. adding a new task – automatically adapts the instrumentation.

Tracing itself consumes system resources: run-time, RAM and flash-memory. Checking if the remaining resources are sufficient should be done prior to an integration knowing that concerning the run-time this can only be a guess. However in over seven years, Gliwa

[8] Only two loop bounds remain to be annotated manually – out of 120 without any measurements.

GmbH never came across a system which did not offer enough resources for instrumentation.

For projects that have safety requirements up to IEC61508 SIL 3 or ISO/CD 26262 ASIL D, tracing solely can be sufficient to prove correct code-level and system-level timing. The user benefits for late-phase source-code event analysis and tracing are these:

- understand the system and track down timing problems → timing debugging
- measure WCETs of tasks, interrupts, runnables, functions, basic-blocks
- measure minimum, maximum, average and distribution of various timing properties like IPT, GET, CET, RT, DT, PER, ST, JIT (see appendix B on page 48.
- low integration and maintaining effort
- easy to use after the integration and thus suitable not only for timing experts or integration engineers but also for function developers e.g. for code-level optimisation

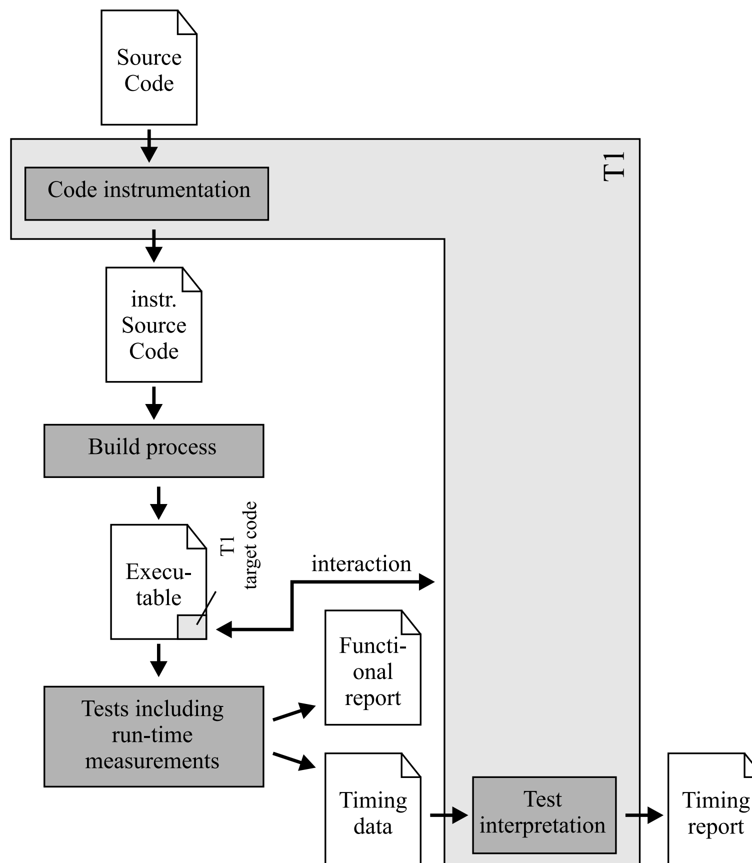

Figure 23: Code analysis with T1

7.4.3 Late-phase Source-code path analysis and tracing

Although the results have a different significance, the deployment of RapiTime at the code level is the same as at the system level described in Section 6.3.3.

When concentrating on the code level, RapiTime's ability to report which parts of the software, down to the granularity of basic blocks, contribute most to the synthetic worst-case path and to the longest measured (high water mark) path allow code analysis effort to be minimised. By concentrating verification efforts on the parts of the code that make the largest contributions, the return on investment is maximised.

For example, imagine our task is to verify that a polling, supervisor interrupt completes with at most 1000 machine cycles of computation. A RapiTime analysis of the software might show that most of the time on the worst-case path is consumed by two common functions. With the remaining, parts of the software that contribute less to the worst-case timing characteristics, the safest and most conservative analysis can be done with potentially large overestimation of the worst-case execution time but without the need for expensive, detailed user involvement. With the two most critical functions, the user can carefully refine the analysis to ensure that safety is maintained whilst overestimation is minimised with a less conservative analysis.

As mentioned in Section 6.3, the analysis can be assisted by source-code analyses. These can be used to increase both precision and confidence in the result, or alternatively to reduce the amount of manual annotations, in a way similar to static timing analysis as described in Section 7.3. However, trace-based timing analysis can benefit from source-code analysis also in other ways. For example in situations where code instrumentation must be sparse due to hardware limitations. Then, the instrumentation points must be selected such that they capture the variations of timing to a maximal degree. This implies that parts of the code that exhibit small timing variations can be more sparsely instrumented. In particular this is interesting for loops: loops that always iterate the same mumber of times are prime candidates to exclude from instrumentation since they are likely to have small timing variations. A static loop bounds analysis, which in addition to upper bounds also can find *lower* bounds, can be used for this purpose: if these bounds coincide, then the loop has a constant number of iterations. The loop bounds analysis of SWEET has this ability, and thus can be used for this purpose.

7.4.4 Early-phase Source-code event analysis and tracing

Assuming that a system already incorporates source-code event analysis and tracing as described in Section 7.4.2, some early phase approaches become possible. In the following, a real customer project is described where tracing in the early phase was very successfully used.

The ALL-TIMES AFS case-study makes use of the third generation of BMW ECUs. When development for this generation started, the second generation was present and used a more expensive processor that included internal flash memory. The third generation was mainly developed to a) reduce costs and b) incorporate new features, both contradicting requirements. The plan was to use a cheaper processor which does not include internal flash so that all the internal code present in the second generation had to be moved to the much slower external flash. At this point, there was the strong request to find a way to prove at this early phase of the third generation that the final system can handle the slower memory as well as the additional functionality.

The following solution was provided, implemented and in the end it predicted the final CPU load with only 2% deviation.

- move all code to the (already existing) external flash

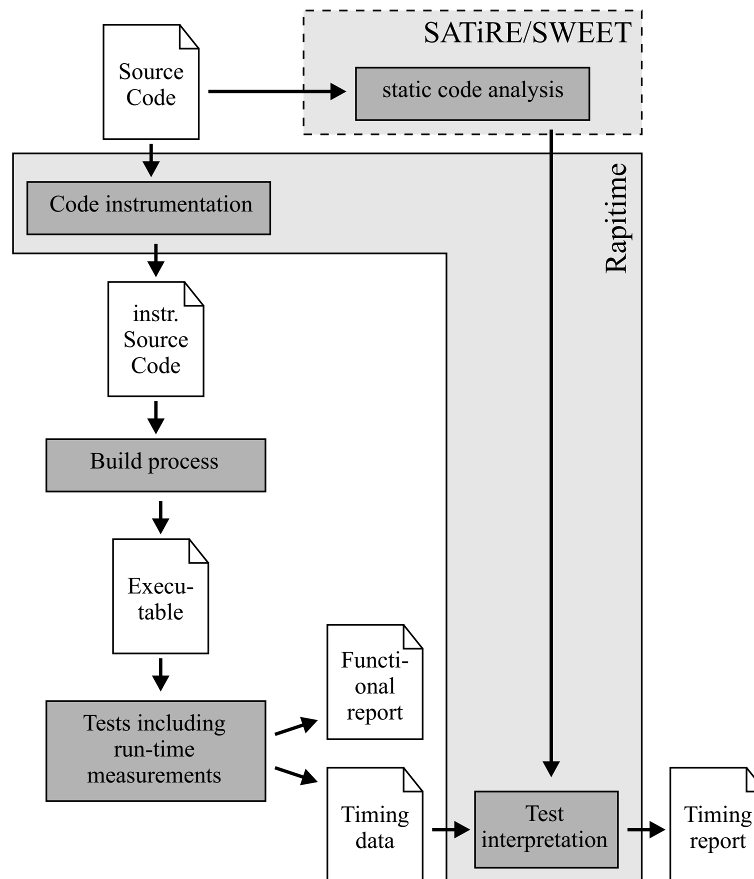

Figure 24: Code analysis with RapiTime and optionally SATiRE/SWEET

- interview function developers in order to find functions which have a comparable complexity with the ones to be developed
- measure the WCETs of these functions
- design the new timing layout, i.e. define where in the schedule the new functionality shall be executed
- using delayGURU, place run-time stubs at these points

In this way, it was possible to perform run-time measurements in an environment very close to the system to be developed at an early stage. Tests were performed at the HIL[9] and then also in the car.

The user benefits of such an approach are

- saving time and costs since implementing run-time stubs with delayGURU is quickly done
- having the possibility to run a future scenario in a realistic environment in an early phase
- and with the new improvements through ALL-TIMES:

[9]HIL = **H**ardware **in** the **l**oop test environment

- passing the results gained to static scheduling analysis (SymTA/S)
- combine/compare the results with results gained through static code analysis (aiT or RapiTime plus optionally SWEET/SATiRE)

7.4.5 Early-phase Source-code path analysis and tracing

Although the results have a different significance, the deployment of RapiTime at the code level is the same as at the system level described in section 6.4.5.

RapiTime analysis, improved with code level results from SATIrE and SWEET, on early versions of software, possibly running on previous generation hardware and/or a simulator, highlights which parts of the software contribute most to the worst-case path. When parts of the software are not even available, stubs can be introduced and their worst-case execution time overridden in the RapiTime analysis using estimates of the expected behaviour of the final code.

Such analysis allows timing defects to detected *as early as possible* in the development process. This greatly reduces the cost of making changes to the software and its configuration. Furthermore, any code optimisation that are required can be guided by RapiTime timing reports. This ensures we focus first on the parts of the code that contribute most to the worst-case execution time and therefore the reactive performance and safety of the software. Rapita have a number of experiences of being able to achieve optimisations with very high benefit to cost ratios using this approach on real, industrial software.

8 Conclusion

Timing analysis has been the subject of academic study for a number of years, and some commercial tools have emerged. However, the tools have operated mostly in isolation, thus limiting their applicability and not utilising their full potential.

Besides the creation of *integrated tool chains*, enabling different timing analysis tools to work together in a highly automated fashion, ALL-TIMES has developed a *methodology*. The ALL-TIMES methodology helps the user choose the best combination of timing analysis tools and techniques for a given situation.

New methods and tools for *early timing estimation* have been developed, which can direct the design work to provide a timing-correct system in shorter time.

Methods and tools for *late timing verification* have been refined, which can reduce verification costs significantly.

The methodology covers the optimised use of static analysis of source code, static timing analysis on the binary level, timing analysis via tracing and measurement, and static scheduling analysis. Although the methodology handles a great bandwidth of techniques, it offers convenient starting-points for various situations, see sections 2.2 and 4. This simplifies finding the right approach for a specific timing problem significantly.

The ALL-TIMES methodology was inspired by the experience of the SME partners working on many timing-critical systems in many customer projects mainly in the automotive and aerospace domains. The methodology has been successfully validated with various industrial case studies.

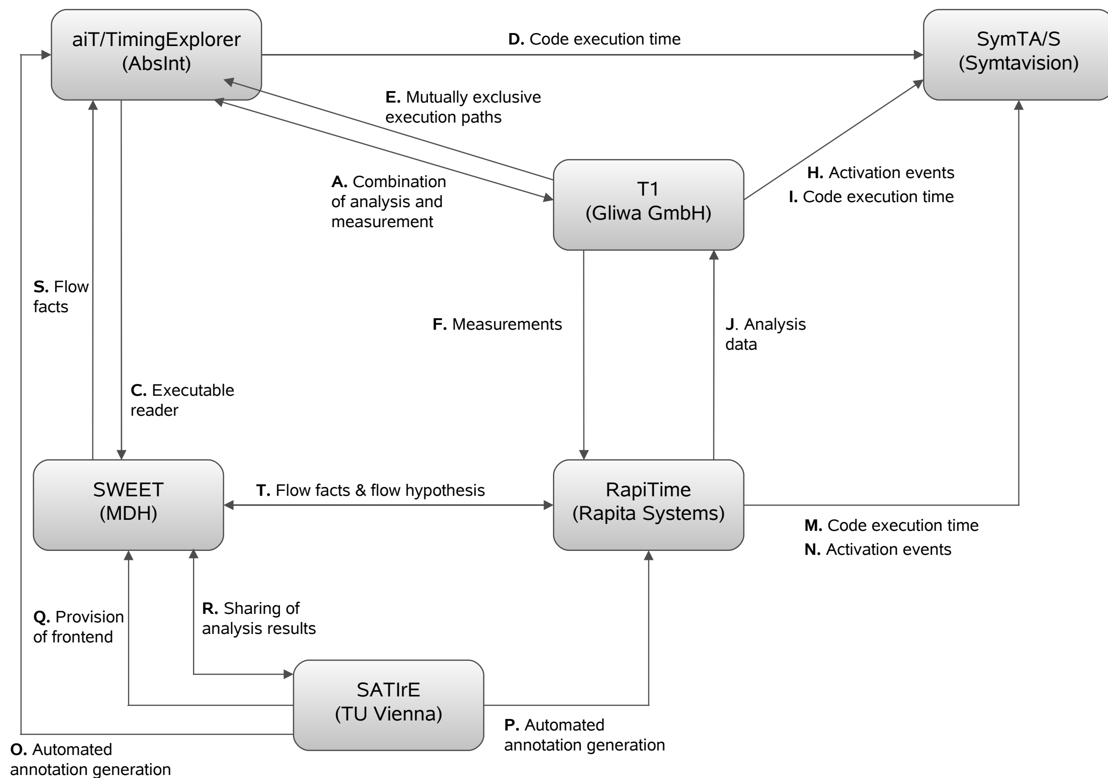

Figure 25: ALL-TIMES integrations between timing analysis tools

A Tools and Tool Integrations in ALL-TIMES

Within ALL-TIMES, a number of tool integrations have been realised through the interfaces that have been defined and implemented in the project. These tool integrations are denoted “use cases”, and they all provide increased functionality by enabling novel combinations of timing analysis tools. Fig. 25 shows the major tools in ALL-TIMES, and the different data-flows which result from the use cases implemented within the project.

B Timing Properties and Constraints

B.1 Timing Properties

Timing properties are timing related measures. Table 2 summarizes the most important timing properties. All of these except for the core execution time require a system view because they depend on preemptions etc. Figure 26 visualises the timing properties with a simple timing situation where a task A gets preempted by a task B. All shown timing properties are properties of task A.

B.2 Timing Constraints

For each timing property, it is possible to also specify a timing constraint. Timing analysis then verifies, if a timing property meets the corresponding constraint. The definition of

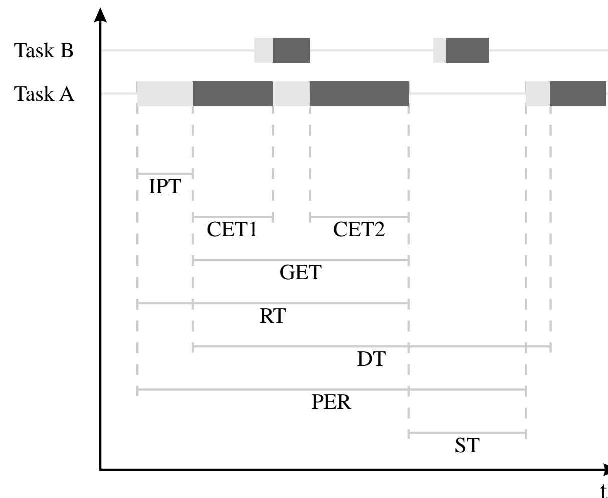

Figure 26: Timing properties visualised in a trace (all related to TASK A)

ID	Abr.	Name EN	Name DE	Description
1	IPT	initial pending time	Initialwartezeit	from activation to start
2	CET	core execution time	Nettolaufzeit	execution time not including any preemptions
3	GET	gross execution time	Bruttolaufzeit	execution time including all preemptions
4	RT	response time	Antwortzeit	from activation to termination
5	DT	delta time	Deltazeit	from start to start ("measured period")
6	PER	period	Periode	from activation to activation (period not as measured but as configured)
7	ST	slack time	Restzeit	"remaining" run-time: from termination to activation (tasks) or start (interrupts)
8	JIT	jitter $\left(1 - \frac{DT}{PER}\right)$	Jitter	deviation of delta time from period

Table 2: Timing results

timing constraints should specify time ranges and not time points. This is illustrated in the following examples:

- "The deadline for the response time of task A is 6.5 ms" is a valid constraint because it defines the time range $0 < t < 6.5\,\text{ms}$.
- "All periodic tasks must terminate once within their period" is a valid constraint because it defines the time range $0 < t < period$. The requirement that the tasks terminate within this time implies that the tasks actually run.

An example for a poor definition would be: "The temperature control algorithm must be executed every 10 ms". It is not possible to run any code *precisely* every 10 ms. The definition lacks the required precision and should read something like "The temperature control algorithm must be executed every 10 ms ± 1 ms".

C The modeling of timing in AUTOSAR

The AUTOSAR partnership – an alliance of OEM manufacturers and Tier-1 automotive suppliers with many associates – has established a number of de-facto open industry standards for automotive E/E architectures with concepts borrowed from earlier successful standards such as OSEK/VDX. AUTOSAR's goal is the definition of a software architecture with standardized APIs and configuration files for application and basic software, which allows exchanging parts of the system's software in ways that programmers know from Java or C++. Key goals are modularity, scalability, transferability and re-usability of software among projects, variants, suppliers, customers. This shall enable new, optimized ways of mapping software to nodes in a flexible network architecture. This is indicated in Figure 27.

Obviously, system timing properties have a strong impact on key steps in the newly envisioned AUTOSAR methodology. For instance, adding a software component to an existing ECU potentially introduces non-functional timing and performance interference with the original ECU software due to scheduling, arbitration, blocking, buffering etc., eventually generating hard-to-find timing problems, including transient overload, buffer under- and over-flows, and missed deadlines that can finally make the new or the old functions or the entire ECU fail. Therefore, AUTOSAR is currently working on system-level timing extensions for the next release AUTOSAR R4.0. The key focus is on scheduling (ECU and network) and end-to-end timing-chains. The extensions target two key challenges. First, there is a model mismatch between the AUTOSAR software components and the task / runnable model used for scheduling (see Figure 28). And secondly, the logical end-to-end timing dependencies of the software structure view do not yet reflect the complex interaction mechanisms of the implementation that realizes the actual communication (see Figure 29).

The current activities of AUTOSAR are focusing on model extensions, while methods are of major concern in the EU-funded ITEA2 project "TIMMO". TIMMO's goal is defining a timing augmented description language (TADL) for software component models such as AUTOSAR, along with an appropriate methodology. Requirements for that methodology are derived from scenarios provided mostly by the OEM and Tier-1 partners (the intended "users" of TIMMO) that are primarily concerned with their development and business processes, so the TIMMO methodology is defined around roles, work products and work tasks. The timing expertise is brought into TIMMO by few technology providers such as Symtavision, ETAS, and TTTech. Here, there is a natural link between TIMMO and ALL-TIMES because ALL-TIMES focuses on tool chains to enable timing analysis, which have to be applicable to the roles and processes developed in TIMMO and AUTOSAR.

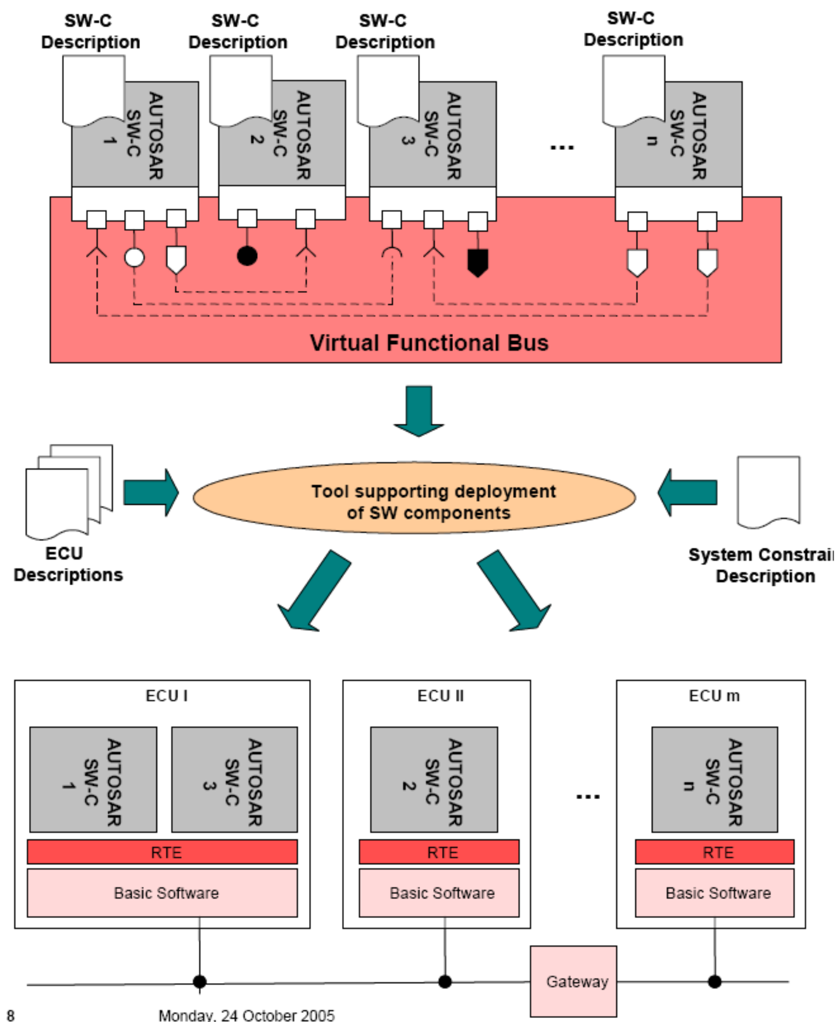

Figure 27: Flexible Software Mapping in AUTOSAR

D Aerospace and other Non-Automotive Markets

The ALL-TIMES methodology focuses on automotive requirements. However, the basic methodology also applies to other domains simply because the general situation is very similar. Specifically, we want to highlight aerospace/aviation, because the similarities to automotive are very obvious.

software integration: Automotive has AUTOSAR with its software components as the key components or building blocks, while avionics has Integrated Modular Avionics (IMA) with its "protected partitions". The similarity has been subject to many talks and comparisons already.

operating system standards: Automotive has OSEK and the OSEK-like AUTOSAR-OS (priority based) and avionics has partitioned ARINC 653 (time-multiplexing combined with priority scheduling)

timing models: AUTOSAR will have a timing model, and the ARINC 653 OS mecha-

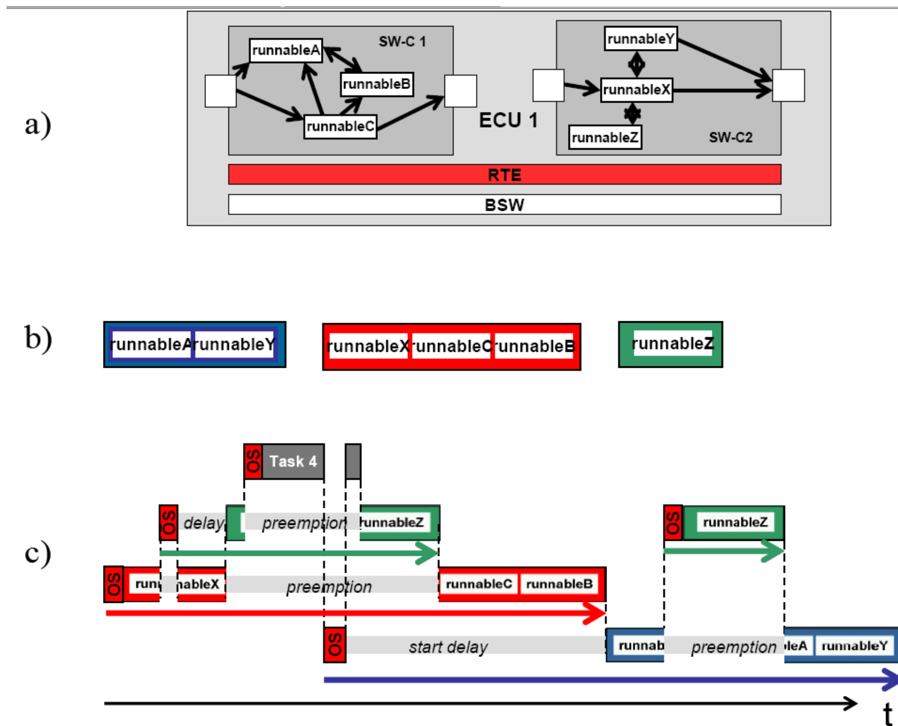

Figure 28: Software Components Structure vs. Task Structure

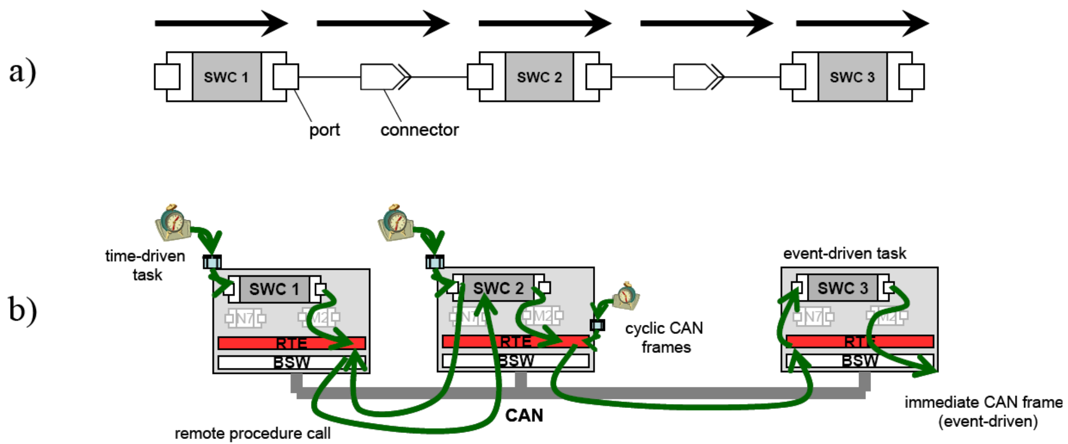

Figure 29: View and Implementation Mechanisms of “End-to-End Timing Chains”

nisms actually rely on predefined time control.

standardized communication protocols: Automotive has CAN (priority based) and FlexRay (time-multiplexing), avionics has AFDX (Round-Robin) and TTP (time-multiplexing)

supply-chains: Both markets have a small number of OEMs (car or aircraft manufacturers), a larger number of Tier-1 suppliers and a variety of Tier-2, software, and tool suppliers. IP-protection and responsibilities are common issues.

Without going into the details, we can summarize that the development processes share very similar basic principles. It is a multi-supplier, component-based approach that facilitates integration through standardized interfaces. Timing parameters play a role in both standards, especially when it comes to integration.

Of course, the mechanisms are different. AUTOSAR OS uses priority-based scheduling that leads to tight timing-dependencies among all involved software-components. The partitioning idea of ARINC 653 partially eliminates this dependency by explicitly multiplexing the processor time using a static, fully predictable time division scheme. However, inside each partition, scheduling is still dynamic. Therefore, the basic principles (and challenges) of scheduling remain. On the one hand, the overall time schedule (called major frame) shall be optimized to reduce the overall idle time. This is an early optimization task. On the other hand, each partition (the pendant to automotive software components) must receive enough processor time to complete their tasks in time. This is a later-stage verification that must be planned for in the beginning. Here is a natural interface between components (the ARINC partitions) and the IMA integration that is identical to the integration of AUTOSAR software-components using the OSEK-mechanisms.

Hence, the fundamental methodology that was presented in detail in the preceding sections should also be applicable to avionics and aerospace. It is beyond the scope of the ALL-TIMES project to validate an aerospace methodology, since all validators come from the automotive domain. However, in the parallel INTERESTED projects, several validators are from the aerospace domain, and AbsInt and Symtavision are testing some of the methodology ideas there.