Integration of Code-Level and System-Level Timing Analysis for Early Architecture Exploration and Reliable Timing Verification

C. Ferdinand[1], R. Heckmann[1], D. Kästner[1], K. Richter[2], N. Feiertag[2], M. Jersak[2]

1: AbsInt Angewandte Informatik GmbH, Science Park 1, D-66123 Saarbrücken, Germany

2: Symtavision GmbH, Frankfurter Straße 3b, D-38122 Braunschweig, Germany

e-mail: ferdinand@absint.com, heckmann@absint.com, kaestner@absint.com, richter@symtavision.com, feiertag@symtavision.com, jersak@symtavision.com

Abstract: Developers of safety-critical real-time systems have to ensure that their systems react within given time bounds. Sophisticated tools for timing analysis at the code-level, controller-level and networked system-level are becoming state-of-the-art for efficient timing verification in light of ever increasing system complexity. This trend is exemplified by two tools: AbsInt's timing analyzer aiT, which can determine safe upper bounds for the execution times (WCETs) of non-interrupted tasks, and Symtavision's SymTA/S tool, which computes the worst-case response times (WCRTs) [7, 11, 16]. of an entire system from the task WCETs and information about possible interrupts and their priorities. The two tools thus complement each other in an ideal way. They have recently been coupled to further increase their utility. Starting from a system model, a designer can now seamlessly perform timing budgeting, performance optimization and timing verification, considering both the code of individual functions, as well as function and subsystem integration. The paper explains and exemplifies various use cases and tool flows.

Keywords: Schedulability analysis, timing analysis, worst-case timing verification, architecture exploration

1. Introduction

Today's innovations in many market sectors (automotive, aerospace, rail, consumer, medical, telecommunication, automation, etc.) are, to a great extent, based on electronics and real-time software. The increasing integration complexity of software-based systems leads to performance bottlenecks and turns real-time software design into a complex task. This increases the risk of timing problems that may degrade or compromise the system functionality but are more and more difficult to detect and even harder to resolve. If they are found late in the design, their resolution can be very costly or even delay a project. A second risk results from wrong dimensioning or configuration decisions early in the development, which may cause resource bottlenecks in later phases which then are very difficult to correct.

In this paper, we present technological solutions for both problems in an integrated view. Integration of techniques and tools is key to efficiently support the development. We work out the requirements for each step and the integration. Then, we present recent progress in the field of static code and scheduling analysis techniques. In specific variations, these techniques can be used for both, early exploration and late verification. And their common abstraction levels provide a seamless integration. Finally, we outline a methodology that can be aligned with established automotive and aerospace development processes.

2. Requirements

Ideally, the system is designed such that it provides sufficient computing power and network bandwidth, and at the same time is cost efficient and (depending on the application) provides the necessary safety level. To achieve this, timing must be taken into account in mainly two phases during the development cycle.

Early-stage architecture exploration includes selecting network topology, processors, software-to-controller-mapping and the integration and optimization thereof. This requires predictions of software execution times and system integration effects that arise when several software parts share one hardware component (CPU). These predictions need not be highly accurate. Instead, a quick prediction and the ability to evaluate and compare (many) different alternatives is the key requirement in early stages.

Late-stage verification includes finding the worst-case scenarios and determining how often they can occur. This requires a reliable verification of code execution times, response times, and end-to-end la-

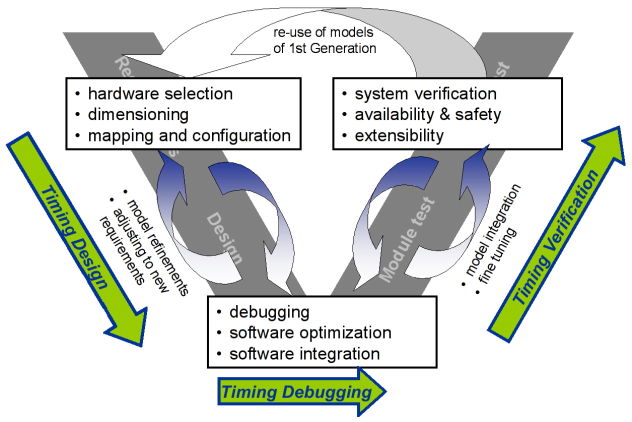

Figure 1: Automotive Development Process (V-Model) and the role of timing analysis

tencies to find the critical corner case scenarios of system execution in the presence of complex processor architectures, multi-tasking operating systems, and network protocols. Here, detecting the worst-case situations and (optionally) the probability of their occurrence is mandatory.

For an **efficient, integrated flow**, these techniques ideally share a common abstraction level that is efficient (in early stages) and expressive (for later stages) enough but can also be deliberately configured in their specific use for a specific design stage. Furthermore, the accuracy must be suitable for the development stage at hand. And finally, they must provide interfaces between code level and system level as well as between early-stage exploration and late-stage verification.

3. Timing in the System Development Process

The automotive system development process typically follows the V model (Figure 1). In an early system development phase, basic system design decisions are made. Timing analysis is of particular value for

- hardware selection and dimensioning, in particular of processors,
- mapping of tasks to processors (in case of multi-processor or distributed systems),
- configuration of the task scheduler (static, priority-based or time-sliced scheduling, number of tasks etc.).

As many implementation details are not fixed, this phase relies on execution time estimates and budgets. Tool support is needed to quickly determine if the intended budgets will lead to an overall schedulable and real-time capable solution.

During the implementation phase the focus shifts to the actual implementation of the set of software tasks on the target processors. Typically this is an iterative approach where individual software tasks are tested, debugged, and integrated. As more and more software tasks are implemented, the acceptable processor loads are often reached earlier than expected. In that case, timing analysis is needed for code execution time optimization and fine-tuning the scheduler configuration.

In the system verification phase, timing analysis is of particular value for

- code and system timing verification,
- proving availability and functional safety,
- calculating reserves for future extensions.

Development processes in the aerospace industry typically follow a cascade model (Figure 2). This approach is heavily process-oriented, which fits well to the idea of execution-time budgets. The same basic roles of timing analysis during the development process can be identified as in the automotive process.

Like in the automotive process, there is an initial development phase that includes processor selection and dimensioning, the mapping of tasks to processors, and scheduler configuration. This phase benefits from the usage of exploration tools that provide early estimations of the timing behavior.

The aviation industry in particular puts special emphasis on reliability issues. All parts of an aircraft, including software and its realization in hardware, must be certified. This includes proofs that no timing constraints are violated. Therefore the demonstration of upper bounds of worst-case execution and response times is mandatory as part of system verification.

As can be seen, timing analysis is needed for both exploration and verification for both the V model and the cascade approach. The task of comparing different mappings, of budgeting, and of property checking can be found in both flows. Estimation is the key enabler for an efficient application in early-stage exploration. System designers can now focus on their key questions such as processor selection and system configuration, while deferring the decisions on code and operating system details to later stages.

Development standards in both the automotive and aerospace domain (in particular DO-178B, IEC 61508 and the new revisions DO-178C, or ISO 26262) increasingly emphasize ensuring software safety. They require to identify functional

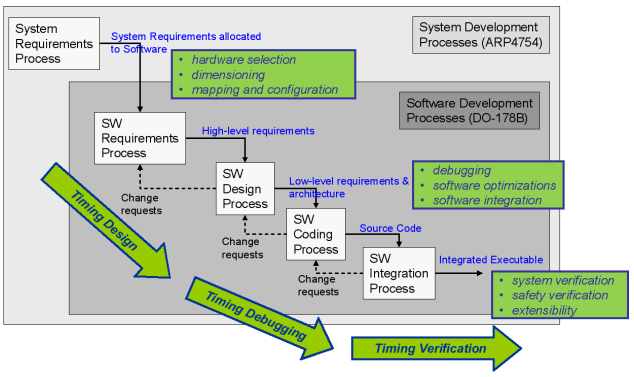

Figure 2: Aerospace Development Process (Cascade) and the role of timing analysis

and non-functional hazards and to demonstrate that the software does not violate the relevant safety goals. Examples for non-functional safety-relevant software characteristics relevant for DO-178B and IEC 26262 WD are runtime errors, execution time and memory consumption. Depending on the criticality level of the software, the absence of safety hazards has to be demonstrated by formal methods or testing with sufficient coverage.

4. Timing Analysis Tools

For both, code execution and system scheduling analysis, static analysis techniques exist. These were originally designed for late-stage verification by providing worst-case analysis for task execution times, response times, and end-to-end latencies. In this paper we focus on two commercially existing tools: AbsInt's aiT Worst-Case Execution Time (WCET) Analyzer [4] and Symtavision's SymTA/S Scheduling Analysis Tool Suite [7]. These are established standards for a growing number of verification tasks, including issues of functional safety.

Both underlying techniques provide an efficient level of abstraction from hardware and from operating system details and are applicable before the final target platform is available. In fact, both have already been used in exploration projects in their original form. However, as mentioned above, early architecture exploration requires less accuracy, does not rely on worst-case analysis, but calls for quick comparisons of "what-if" experiments. Therefore, these technologies have been developed further to better meet the requirements for early-stage exploration.

4.1 Code Level

"Code level" refers to the "un-preempted" execution times of sequential code pieces such as OS tasks and processes. Of particular interest for dimensioning and verification are the worst-case execution times (WCET). WCET estimations for such code pieces can be obtained from the AbsInt tools *TimingExplorer* [5] (specialized to early-stage architecture exploration) and *aiT* [4] (specialized to late-stage system validation). Both tools employ static analyses that produce results valid for all executions with all possible inputs. In contrast, hardware simulation, emulation, or direct measurement with logic analyzers can only determine execution times for selected inputs. Achieving sufficient coverage using such methods can be quite expensive, and they still cannot be used to infer the execution times for all possible inputs in general. (A survey of methods for WCET analysis and of WCET tools is given in [17].)

Since the timing behavior of the code differs on different hardware, aiT and TimingExplorer operate on binary executables, which are the main input of the tools. They also read source code to be able to refer to it in their user interactions. In addition, they read annotations that might be manually written or automatically generated by code generators or external source-code analyzers, and a hardware description detailing the target architecture (memory layout, cache properties, etc.).

The difference between the two tools is that aiT is targeted towards validation of real-time systems and therefore aims at high precision and closely models the underlying hardware, while TimingExplorer is intended to be used in early design phases for exploring alternative system configurations during the search for a suitable processor configuration (core, memory, peripherals, etc.) for a project.

aiT is sound in the sense that it computes a safe over approximation of the actual WCET. Depending on the target processor this can lead to high analysis times (typically up to several minutes for a task). Such analysis times are acceptable in the verification phase, but are undesirable for configuration exploration. Moreover as opposed to verification for which soundness is of utmost importance, for dimensioning hardware an over approximation is not always necessary. As long as TimingExplorer reflects the task's timing behavior on the architecture and allows comparison of the timing behavior on different configurations, slight under approximations are acceptable. Therefore some precision is traded against ease of use, speed and reduced resource needs leading to the possibility to quickly explore the timing behavior of one's software on a wide selection of potential hardware configurations (without the need to acquire all this hardware).

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. The available source code is compiled and linked using representative standard compilers for each of the cores considered as potential target processors. Each resulting executable is then analyzed with the TimingExplorer for the corresponding core. For each core, the user has the possibility to specify memory layout and cache properties. The result of the analysis is an estimation of the WCET (worst-case execution time) for each of the analyzed tasks given the processor configuration.

Since TimingExplorer is based on static analysis, it does not require that the explored architectures are available or even exist at all. There are instances of TimingExplorer for a variety of parameterizable cores that represent typical architectures of interest. To be applicable in early design phases, the cache and memory mapping is completely parameterizable so that the user can experiment with different configurations. It is possible to set the cache size, line size, replacement policy and associativity independently for the instruction and data cache. Furthermore, one can choose how unknown cache accesses are to be treated – as cache hits or cache misses. With respect to the memory map, one can specify properties for memory areas, such as the time it takes to read and write data to the area, whether write accesses are allowed or the area is read-only, if accesses are cached, etc. To see the effect of these settings on the WCET, one simply has to rerun the analysis after modifying the settings. The questions developers will be able to answer are like "what would happen if I take a core like the ABCxxxx and add a small cache and a scratch pad memory in which I allocate the C-stack or a larger cache" or "what would be the overall effect of having an additional wait cycle if I choose a less expensive Flash module", ...

On the other hand, the aiT tool is intended to be used in the late development stage during the validation of the timing behavior of the developed system. It computes a safe upper bound for the worst-case execution time (WCET) of a task, assuming no interference from the outside. Effects of interrupts, IO and timer (co-)processors are not reflected in the predicted runtime and have to be considered separately within system-level timing analysis.

aiT and TimingExplorer employ static program analysis by abstract interpretation [3] to get information on the values in the processor registers and memory cells, on cache contents, and on the behavior of the processor pipeline [9, 13]. This information is used to obtain WCET bounds for basic blocks. On the other hand, integer-linear programming is employed for finding a possible worst-case path and an upper bound for the overall WCET [10, 14].

The results of aiT and TimingExplorer are reported as annotations in call graphs and control-flow graphs, and as report files in text format and XML format. The overall WCET bounds/estimations for sequential code pieces can also be communicated to the system-level analyzer SymTA/S, which computes worst-case response times from the sequential WCETs, taking into account interrupts and task preemptions. Details on the integration between aiT/TimingExplorer and SymTA/S are presented in Section 5.

Integration in the Development Process:

Worst-case execution time analysis can be integrated seamlessly into the development process and is not only applicable at the validation stage but also at the development stage. One advantage of static analysis methods is that no testing on physical hardware is required. Thus the analyses can be called just like a compiler from a workstation computer after the linking stage of the project. As an example, aiT and TimingExplorer feature batch versions which facilitate the integration in a general automated build process. This enables developers to instantly assess the effects of program changes on the worst-case execution time.

Certification and Tool Qualification:

Making sure that an application is working properly means addressing many different aspects.

In the *validation stage* the goal is to verify that the worst-case execution time bounds of the application are not exceeded. To be amenable for certification according to DO-178B, the analysis tools themselves have to be qualified. The qualification process can be automated to a large degree by *Qualification Support Kits*. The qualification kits for aiT and TimingExplorer consist of a report package and a test package. The report package lists all functional requirements and contains a verification test plan describing one or more test cases to check each functional requirement. The test package contains an extensible set of test cases and a scripting system to automatically execute all test cases and evaluate the results. The generated reports can be submitted to the certification authority as part of the DO-178B certification package.

4.2 System Level

On the system-level, a different approach has to be applied. Usually the code is not fully implemented. Therefore, the system model is typically a mix of a detailed model based on a *predecessor* system, and code execution time estimates for the parts which have not been implemented yet. The analysis focus is not reliable worst-case timing, but quick what-if analysis of different software mappings and system configurations.

The system-level timing analysis methodology distinguishes two system scopes:

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.

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

- Software modules
- Communication signals and semantics
- CPUs
- Buses
- Mapping of software to CPUs, CPU configuration and scheduling
- Mapping of signals to buses or memories, bus configuration and scheduling
- Timing constraints

System Dimensioning and Configuration:

When developing a real-time system controller, a key question is the optimal choice of CPU / CPU configuration (cache, memory ...). Such decisions have to be based on software execution time budgets, with execution time estimation to determine the feasibility, bottlenecks and reserves when implementing software according to those budgets.

As explained, code-level and system-level verification techniques can be reused also in early design phases, using predecessor systems and prototypes [12]. The major difference in the early phase is a combination of SymTA/S for virtual configuration and budgeting and TimingExplorer for estimation.

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 dataflow. Standard function modeling tools provide this synthesis capability. The initial system configuration is then imported into SymTA/S. In the automotive domain, this increasingly happens through the standardized AUTOSAR format. As far as available from previous systems, code execution times 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 code. At this stage, the following degrees of freedom can be explored:

1. change the execution time for the new code
2. change the scheduler configuration
3. in case of a multiprocessor controller or multicore CPUs, change the allocation of code 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 optimization and sensitivity analysis for this purpose. The other changes can be performed manually, because the reordering or remapping of code usually is severely constrained by other design considerations. 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.

System Timing Verification:

The system model described above is augmented with a more detailed view on the observable timing behaviour in specific situations. Each piece of code is characterized locally using either WCET analysis or measurement (depending on the required level of safety). This locally observable timing is then combined into a global timing verification using scheduling analysis [7, 8].

When static code analysis is used on the code level, this analysis provides safe upper bounds for worst-case code execution times (WCETs). In the integrated tool flow described in this paper, WCETs of tasks, interrupts and runnables are calculated in aiT and passed to SymTA/S using the XTC mechanism (Section 5.1). This approach has been developed during several research projects, most recently ALL-TIMES [6].

Alternatively, tracing allows to measure the execution time of code. This is only as good as the test cases by which the system has been exercised. However, with a good test-suite, a lot of information can be obtained about the system, including statistical data and visualizing the execution around the time when an error occurred.

Independent of the code-analysis technique, code execution times obtained by any of the code-level techniques are compared against the code execution time budget defined in the initial design phases. If the obtained value is smaller than the budget, then the verification is complete for that code, 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 execution time and is spare computation time.

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

- If the execution time was obtained using WCET analysis, then aiT offers 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.
- A second approach is to actually reduce execution times by optimising the software. Both aiT and trace tools such as Gliwa T1 report identified worst-case paths. Therefore, it is easy to identify which parts of the software consume most time. This information ensures that potentially expensive optimisation efforts are only deployed where they will have maximum benefit for reducing the execution time of the software.
- A complementary approach is to increase the budget. This is promising in particular if analysis of other code 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.

The mentioned technologies for code- and system-level exploration (SymTA/S Architecture Explorer and aiT TimingExplorer) are well suited to support these processes in the design phase. System verification is performed during system integration. Increasingly, this includes the verification of the timing properties of the system, which is supported by the combination of SymTA/S System Verifier and aiT WCET analyzer.

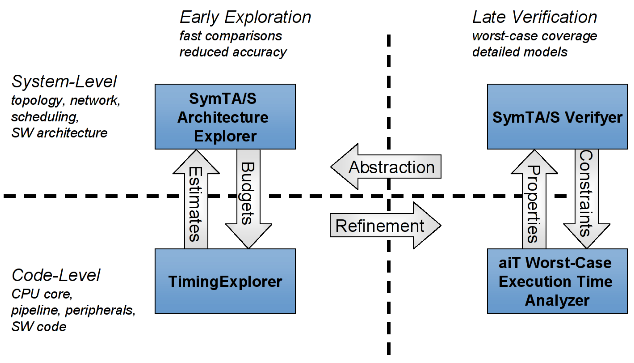

Figure 3: Code and System Analysis in Exploration and Verification

5. Tool Integration

Both mentioned technologies have their specific applications (code-level and system-level) but at the same time they fit well together (see Figure 3). In the established use for verification, worst-case execution time analysis provides timing data for individual task execution (as a sub-system property) that is required by system scheduling analysis. The other way round, SymTA/S can provide detailed constraints for parts of the software.

The new exploration variants of the tools offer new flow options. TimingExplorer can provide first execution time estimates for an early code-level exploration, while the SymTA/S Architecture Explorer can provide execution time budgets for the hardware and software selection at the system level.

For this data exchange, AbsInt and Symtavision have realized a data exchange mechanism and format called XTC (eXtensible Timing Cookie), developed in the course of the INTEREST project and extended in the ALL-TIMES project.

5.1 XTC

The system-level tool SymTA/S by Symtavision communicates with the code-level tools aiT and TimingExplorer by AbsInt, T1 by Gliwa via the XTC 2.0 interface (see figure 4). 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, as it's predecessors, open and can be , and has been, adopted by other tools not mentioned here.

XTC means "eXtensible Timing Cookies". The main idea behind these "Timing Cookies" is derived from

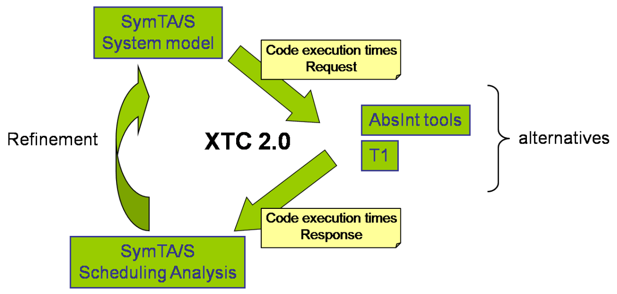
SymTA/S System model
Code execution times Request
Refinement
XTC 2.0
AbsInt tools
T1
alternatives
Code execution times Response
SymTA/S Scheduling Analysis

Figure 4: XTC connection between system level and code level

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 Figure 4, 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, response times, activation patterns aligned with the upcoming AUTOSAR 4.0 [1] and TIMMO [15] 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).

5.2 ATF

ATF was developed in the All-Times project to allow exchange of trace, or trace related, information between tools. ATF stands for "**A**ll-Times **T**race **F**ormat". In addition to the trace ATF also provides basic information on the traces system and it's elements. ATF has some similarities to the previously described XTC. In contrary to XTC the ATF is not based upon a request-response mechanism. ATF was designed to allow any tracing tool to integrate with any trace evaluation tool. A possible tool-flow can be seen in figure 5. ATF supports different types of bus protocols and operating systems. These are explicitly listed in the ATF schema and contain important OS types like OSEK [2] and AUTOSAR-OS [1].

ATF is an XML file format that has been developed within the ALL-TIMES project. Its purpose is to allow different software timing tools to exchange timing traces. A timing trace is a sequence of timestamped events that indicates how a system (in this case a software application) behaves over time. For one example, suppose that one event corresponds to the start of one task and another event corresponds to the start of another task and the tasks are supposed to run in strict alternation. A timing defect could be manifested as one of the tasks running twice in succession without the other task running in between. Even though this might be very rare and virtually impossible to capture using conventional debugging techniques, collecting and then searching a timing trace can highlight exactly when the problem occurs. For another example, suppose that events correspond to the start and end of all the tasks and interrupt service routines (ISRs) in the application. A timing trace enables measure-

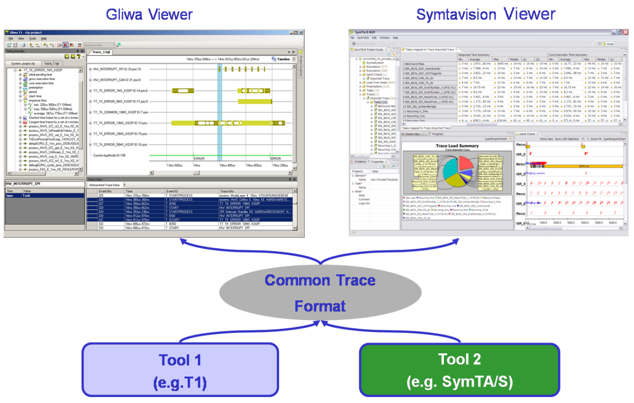

Figure 5: Example for multiple trace processing tools interacting using ATF

ment of the execution time for each task and ISR individually, separating time spent actually executing a task from time spent while that task is not executing but waiting for an interrupt to complete. Timing traces are a vital part of modern software development, debugging and verification. By having an open, publicly documented exchange format for traces, the ALL-TIMES partner tools allow sharing of trace data not only with each other but also with any other tools that can import or export ATF.

The tracing environment (system description) can be configured in the ATF. Each ATF can contain multiple traces from multiple tracing tool vendors. The trace data has been separated from the actual system description. This allows for a single event to describe a user event as well as a task activation. Some additional information on the tracing overhead can be provided in the ATF.

6. Conclusion

We have presented requirements and a methodology for exploring different hardware architectures and configurations and obtaining timing estimations in early stages of system design, and for verifying / certifying systems in later stages.

We explained the methodology using the standard scheduling analysis tool SymTA/S and WCET analysis tool aiT. We also gave an introduction into the technological background of the execution time and scheduling analysis, in particular w.r.t. the applicability in early and late phases.

7. Acknowledgment

Collaboration between AbsInt GmbH and Symtavision GmbH has been supported by the FP7 projects INTERESTED and ALL-TIMES. AbsInt's work has additionally been supported by the FP7 project PREDATOR.

8. References

[1] AUTOSAR Development Partnership. Automotive Open System Architecture (AUTOSAR). URL: http://www.autosar.org, 2003.

[2] Continental Automotive GmbH. OSEK/VDX. URL: http://www.osek-vdx.org.

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977.

[4] C. Ferdinand and R. Heckmann. Worst-case execution time – a tool provider's perspective. In *11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing ISORC 2008, Orlando, Florida, USA*, May 2008.

[5] C. Ferdinand, R. Heckmann, D. Kästner, and S. Nenova. Architecture exploration and timing estimation during early design phases. *Embedded World Congress, Nuremberg*, Mar. 2010.

[6] J. Gustafsson, B. Lisper, M. Schordan, C. Ferdinand, P. Gliwa, M. Jersak, and G. Bernat. ALL-TIMES – a European project on integrating timing technology. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13-15, 2008. Proceedings*, volume 17 of *Communications in Computer and Information Science*, pages 445–459. Springer, 2008.

[7] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis – the SymTA/S approach. *IEEE Proceedings on Computers and Digital Techniques*, 152(2), Mar. 2005.

[8] M. Joseph and P. K. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[9] M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. In *Proceedings of the 9th International Static Analysis Symposium SAS 2002*, volume 2477 of *Lecture Notes in Computer Science*, pages 294–309. Springer-Verlag, 2002.

[10] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, 1995.

[11] K. Richter. *Compositional Scheduling Analysis Using Standard Event Models – The SymTA/S Approach*. PhD thesis, Technical University of Braunschweig, Germany, 2005.

[12] K. Richter, M. Jersak, and R. Ernst. Learning early-stage platform dimensioning from late-stage timing verification. In *Design, Automation and Test in Europe, DATE 2009, Nice, France, April 20-24, 2009*, pages 851–857. IEEE, 2009.

[13] J. Schneider and C. Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, volume 34, pages 35–44, May 1999.

[14] H. Theiling and C. Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 144–153, Madrid, Spain, Dec. 1998.

[15] TIMMO Consortium. TIMMO – Timing Model. URL: http://www.timmo.org, 2009.

[16] K. Tindell. Adding time-offsets to schedulability analysis. Technical Report YCS 221, University of York, 1994.

[17] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.

9. Glossary

aiT:	AbsInt’s Timing analyzer
CPU:	Central Processing Unit
ISR:	Interrupt Service Routine
RTOS:	Real-Time Operating System
SymTA/S:	Symbolic Timing Analysis for Systems
T1:	Tracing tool by Gliwa
WCET:	Worst-Case Execution Time
WCRT:	Worst-Case Response Time
XML:	Extensible Markup Language
XTC:	XML Timing Cookie