Predictable Assemblies using Monitored Software Components

Daniel Sundmark[†], Anders Möller[†*], Mikael Nolin[†]

[†] MRTC, Mälardalen University, Box 883, SE-721 23 Västerås, Sweden

[*] CC Systems, Home Page: http://www.cc-systems.com

`{Daniel.Sundmark; Anders.Moller; Mikael.Nolin}@mdh.se`

Abstract

We propose monitoring of software components, and use of monitored software components, as a general approach for engineering of embedded computer systems. The concept is general in the sense that it addresses the whole product life-cycle including development, debugging, testing and maintenance. Also, reuse across multiple product versions and variants are facilitated by our approach to monitor components.

The approach is mainly targeting the following 4 software engineering goals: Engineering with, and of, certified components, system level testing and debugging, run-time contract checking, and observability of system-internal behaviour. These goals are achieved by monitoring 4 aspects of component-level properties: Timing behaviour, memory usage, event ordering, and input-output sanity checks. Fulfilling these engineering goals will significantly reduce the costs, efforts and risks involved with developing dependable embedded systems.

We propose an engineering approach where a component's execution is continuously monitored and experience regarding component behaviour is accumulated. As more and more experience is collected the confidence in the component grow; with the goal to eventually allow certification of the component. Continuously monitoring is also the base for contract checking, and provides means for post-mortem crash analysis; an important prerequisite for any company to start use 3rd party component in their dependable systems.

Keywords: Component Monitoring, Predictable Assemblies, Embedded Systems, Component-Based Software Engineering, CBSE, Run-time contract checking

1. Introduction

In this paper we propose monitoring of software components and use of monitored software components as a general approach for engineering of embedded computer systems. Industrial developers of distributed, heterogeneous, reliable, resource constrained, embedded, real-time control systems (in this paper denoted embedded systems) are facing increased challenges with respect to increased demands on profitability, functionality and reliability, while at the same time having to decrease development times, project costs and time-to-market. Since development costs only constitute a fraction of the total project cost for software projects (about 20% [24]), a general approach for engineering embedded systems must consider not only the development phase; also the debugging, testing and maintenance phases need to be addressed. Furthermore, since most systems are developed incrementally, where new versions are based on previous versions, and product-line architectures [1] are becoming increasingly important, a general approach for engineering embedded systems needs to consider reuse of components between product versions and product variants. Another emerging key-issue in engineering of embedded systems is safe and predictable integration of third-party functions, and the associated legal matters regarding contract fulfilment and liability issues.

Our approach for monitoring software-components, and use of monitored software-components, will address the following key-areas within engineering of embedded systems:

- ✓ Certifiable components. By monitoring component-based software, information about the component properties can be extracted. This information can be used to fully (or partially) describe the components by their externally visible properties. These properties provide a basis for trust in components and for system predictions. By reusing certified components, predictable component assemblies are facilitated.

- ✓ System-level testing and debugging. By monitoring individual components and component interactions, errors can be found and traced. Monitoring can also be used to support *replay debugging* [17], where erroneous system-executions are recreated in a lab environment to allow tracing of bugs.
- ✓ Run-time contract checking. This will allow surveillance of third party components. Both functional (e.g. range of output values) and non-functional (e.g. memory usage) properties can be monitored. During acceptance testing, the contract checking is used to validate that a component does not violate its specification. In systems that fail after system deployment, logs from the contract checking can be used in post-mortem analysis to identify failing or contract-breaking components.
- ✓ Observability. Computer systems in general, and embedded systems in particular, are infamous for the difficulty of observing their internal behaviour. This has drawbacks throughout the whole debugging, testing and maintenance phases. Systems whose behaviour is unobservable become very difficult to analyse and validate. Also after deployment, observability is an important feature, allowing inspection and performance tuning of running systems. Aftermarket tools can be used to plug into deployed systems to extract information about both hardware and software state.

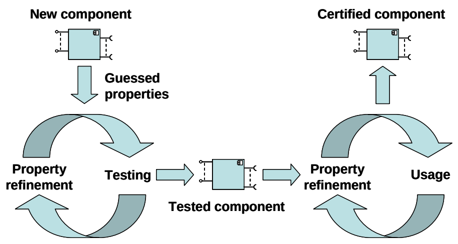

Figure 1: *A conceptual overview of monitoring software components*

The ultimate goal of component monitoring is to be able to compose predictable assemblies by reusing information gathered from well-tested software components. The proposal of this paper is that this can be achieved by the iterative process of refinement described in

Figure 1. When a new component (or a 3rd party component) is included in an assembly, its run-time properties (such as execution time or memory consumption) are estimated by well-founded guesses. During testing (component-level and assembly-level), these guesses are validated and refined, producing a tested component. As the tested component is deployed in a target-system assembly, its behaviour is continuously monitored, allowing for further refinement of the component run-time behaviour description. This refinement process will eventually lead to a certified component, which can be used to compose predictable assemblies. In order to achieve a high level of predictability, all four key-areas mentioned above needs to be considered.

Monitoring of components will allow information about the dynamic behaviour of the component to be recorded. This information allows static and dynamic properties of newly (or partly) constructed systems to be predicted. Interesting aspects to monitor (on component level) and predict (on system level) include timing properties, such as end-to-end response times, and resource utilisation, such as memory consumption.

The outline of the rest of this paper is as follows: Section 2 describes properties of embedded systems. In Section 3, we present a survey of related work in built-in monitoring support for component-based systems and existing monitoring practices in commercial component technologies. In Section 4, the impacts of monitorable components on predictable assemblies are discussed. In Section 6, we discuss the industrial benefits using monitored components, and finally, in Section 7, we summarise and present our ideas on future work.

2. Embedded Systems

In this paper we are addressing software engineering of resource-constrained, embedded, distributed real-time control systems. We will in this section discuss the prerequisites for Component-Based Software Engineering (CBSE) for such embedded systems. We will also give a brief example of a typical embedded system and an introduction to component monitoring.

2.1. CBSE for Embedded Systems

In CBSE, software applications are built by composing software components into component assemblies. CBSE is gaining more and more acceptance in the business segment of office/Internet applications [9][11]. Unfortunately, the market segment of embedded real-time systems is, to a large extent, left behind this positive development. Reusing components, i.e. one of the main drivers for introducing CBSE, is both complex and expensive for embedded real-time systems [2], since such components must work together to meet functional and temporal requirements in a resource constrained environment, while at the same time prohibiting functional errors from propagating and leading to unsafe states.

However, by building embedded-system software out of well-tested components, we could gain an increase in the predictability of the behaviour of the software; provided that experience from component behaviour has been collected. In the area of embedded real-time systems, predictable run-time behaviour is crucial. A component assembly is predictable if its run-time behaviour can be predicted from the properties of its components and their patterns of interactions [15]. Predictability requires analysis, and analysis techniques require information about the system.

When analysing a system built from well-tested and functionally correct components, the main issues are associated with composability. The composition process must guarantee non-functional aspects of the system, such as communication, synchronisation, memory, and timing [2]. However, research projects tend to focus on how to design and analyse component technologies, leaving predictable assemblies using run-time information gathered from well-tested and trusted components unexplored [7]. Thus, very few component technologies include support for run-time monitoring.

2.2. Embedded System Example

In order to exemplify the typical settings, in which the software components are considered, we have studied some characteristic vehicular electronic solutions [12]. An electronic vehicular control-system can be characterised as a resource constrained, safety-critical, distributed real-time system. The computer nodes, called Electronic Control Units (ECUs), are distributed to reduce cabling and to allow for division into subsystems. Vehicular systems are usually heterogeneous, meaning that nodes of different architecture and computational power cooperate in controlling the vehicle. The ECUs vary from extremely light-weighted nodes, like intelligent sensors (i.e. processor-equipped, bus-enabled sensors), to PC-like hardware for non-control applications, such as telematics, and information systems.

Figure 2 gives an overview of the hardware resources of a typical ECU, with requirements on sensing and actuating, and with a relatively high computational capacity (this example is from a power train ECU).

Example Power train ECU in a Vehicular Control-System

- Processor: 25 MHz 16-bit processor
- Memory devices:
  - ✓ Flash: 1 MB used for applications
  - ✓ RAM: 128 kB used for the runtime memory usage
  - ✓ EEPROM: 64 kB used for system parameters
- Serial interfaces: RS232 or RS485, used for service purpose
- Communications: Controller Area Network (CAN) (one or more interfaces)
- I/O: A number of digital and analogue in and out ports

Figure 2: Specification of a typical power train ECU [12]

An example of a typical vehicular system communication solution is shown in Figure 3, where two buses are separated by a gateway. This is an architectural pattern that is used for several reasons, e.g., separation of criticality and real-timeliness, increased available bus bandwidth, increased fault tolerance, or compatibility with standards [22][23]. Communicating functions may require support for global synchronisation or fault tolerance mechanisms.

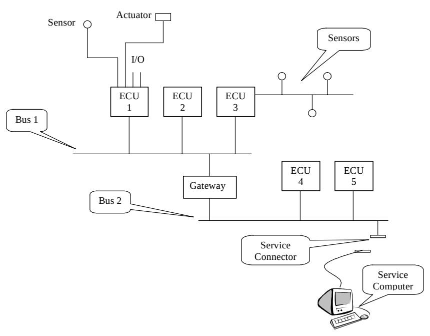

Figure 3: *Example sketch of a vehicle network [12].*

Looking at the software part of the system, there are a few aspects of interest when building the assembly out of monitorable components. A source of uncertainty is the level of interrupts in the assembly. Typically, a vehicular system is heavily loaded with interrupts. When interrupts hit the assembly, these will pre-empt the execution of the running component, thereby possibly perturbing its monitoring.

Dynamic memory allocation (and the garbage collection that this brings) is usually not allowed in control applications, since it compromises the determinism and predictability of the application behaviour. The only type of memory that is allowed to dynamically shrink and grow in the system is the stack space (albeit within a statically allocated stack memory area).

2.3. Monitoring Embedded System Components

Monitoring component-based software requires support in the component technology, and the framework used during run-time. Usually, when looking at today's component technologies suitable for embedded systems with resource constrained ECUs, considerable code optimisations are done during compile time. This is mainly done to minimise the size of the application source code. This code optimisation might lead to a loss of the design-time component concept, meaning that clearly identifiable components with specified in- and out-ports are reduced to regular source code functions, subjected to, e.g., function in-lining and redundant instruction-sequence coalescing.

Thus, to be able to monitor the components in the form described during design-time, and to be able to reuse the information gathered during run-time in the next generation of applications, information about the design-time components have to be included in the source code. This should however not be a problem, if the component technology satisfies the requirements described in [12], i.e. a straight forward port-based object approach, illustrated in Figure 4, using a pipes-and-filters model of computation.

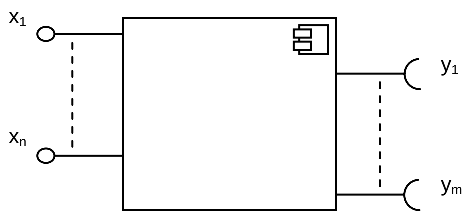

Figure 4: *Component with required in-ports x_1 - x_n and provided out-ports y_1 - y_m*

3. Related Work

Some existing component technologies include support for component monitoring. These methods, as well as existing techniques for software monitoring, are described in Section 3.1 and Section 3.2.

3.1. Monitoring Techniques for Component-Based Systems

Currently, a few component technologies provide support for run-time monitoring of component behaviour. However, there is a multitude of ways of performing this monitoring and there is a multitude of run-time aspects to monitor.

Gao *et al.* identify three different methods for component tracking and monitoring [7]: (A) framework-based code insertion, where monitoring code (e.g. from a class library) can be inserted by component engineers, (B) automatic code insertion, where monitoring code is inserted into the program by a specialised monitoring tool, and (C) automatic component wrapping, where monitoring code is automatically added to the external interface of components.

According to Gao *et al.*, each of these methods has its own pros and cons. As for framework-based code insertion, it is highly flexible and can be used for all types of monitoring. However, the method requires access to the component source code, and the programming overhead is high. Automatic code insertion also requires access to the source code, and is much more complex and inflexible compared to the framework-based code insertion. However, the programming overhead is low, since the tracking code is automatically inserted. Automatic component wrapping, on the other hand, has no need for component source code in order to insert tracking code. Therefore, not only in-house components, but also Commercial-Off-The-Shelf (COTS) components can be monitored. On the downside, automatic component wrapping is not suitable for monitoring anything within components, since the monitoring is performed exclusively outside the component.

Considering the use of these methods with respect to the restrictions posted by component-based embedded systems, it should be noted that automatic component wrapping can not be used in order to extract any component information other than that available at the component ports. This makes the method unsuitable for monitoring other component properties than those available from outside the component. Automatic code insertion, on the other hand, could be used for all types of monitoring, but would introduce a trade-off between the complexity of the instrumentation tool and the amount of data needed to record. Ideally, especially in resource-constrained systems, the amount of data to record should be minimised. However, this calls for an elaborate analysis of the internal workings of the component, requiring an inflexible (with respect to portability) and highly advanced instrumentation tool. Using framework-based code insertion, no instrumentation tool is required, allowing ad-hoc optimisations in the monitoring code. In a resource-constrained environment, this might be useful, but it must be kept in mind that such optimisations might lead to unpredictable side-effects in system ordering and timing.

Jhumka *et al.* [10] propose the use of executable assertions in order to monitor component behaviour. The assertions are included in component wrappers, enabling them to test the validity of the input and output values of the component. By using these wrapper assertions, the pre- and post-condition sanity checks transforms a regular component into a fault-detecting component while at the same time simplifying unit-, integration- and system-level testing due to standardised means of extracting test information at component interfaces. Being relatively small and straightforward, executable assertions could well be used in order to perform sanity checks of embedded system components. However, executable assertions can not be used in order to monitor other properties, such as execution time or memory usage.

Hörnstein and Edler [8] propose the use of Built-In Test (BIT) components in the Component+ model [3] for reducing the time spent testing prefabricated components in new environments. In order to perform these built-in tests, the Component+ model makes use of three different types of components: BIT Components, Testers and Handlers. BIT Components are regular software components with built-in test mechanisms, Testers are special components that use the BIT testing interfaces of the BIT components and Handlers are special components that can be used to obtain fault-tolerant systems by handling error signals from BIT or Tester components. On the assumption that BIT and Tester components are light-weighted, this can be an effective way of performing component sanity checks or run-time contract checking. Even though Handler components may be effective for achieving fault-tolerant systems, this is not the primary subject of this paper.

Traditionally, software monitoring can be performed by using hardware- or software-based probes. Hardware probes come in the form of lab instrumentation tools, such as In-Circuit Emulators (ICE:s) or logic analysers, or in the form of System-On-Chip (SOC) solutions [4]. ICE:s or logic analysers are not suitable for component monitoring, since they cannot be included in deployed assemblies. SOC-based monitoring tools, however, are designed to be resident in deployed systems. Unfortunately, being designed for system-level event monitoring (e.g., task-switches), these tools are still far too inflexible for component-level monitoring. Therefore, today, software probes seem to be the preferred alternative for component monitoring. However, software-based monitoring is not without drawbacks. By including software monitoring in the component technology, we also introduce problems concerning instrumentation perturbation. Software-based monitoring is performed by means of software probes inserted in the code. These probes will consume execution time and memory space; increasing the spatial and temporal resource consumption of the components. If the execution times of the probes are non-constant, the probes themselves will reduce the testability of components and assemblies [16]. Probes should be left permanently in deployed components for two reasons: (1) If the probes are removed, the testing performed on the component might no longer be valid [5], and (2) by leaving the probes in the deployed component, information concerning execution behaviour can be gathered over long periods of time, while the component operates in its field environment.

3.2. Monitoring Support in Commercial Component Technologies

There are a handful of available component technologies suitable for distributed embedded real-time systems. Some of these technologies include various supports for monitoring the software. We have chosen to study two of these technologies, evaluated with respect to industrial requirements in [13], in more detail. The reason for choosing these is that they are deployed in industry today, and that they well satisfy the industrial requirements stated by the embedded-system domain [12].

The Rubus Component Model (CM) [18] and the Rubus Operating System (OS) [19] have support for some of the described monitoring aspects. Rubus CM and OS are developed by Arcticus Systems[1] and are used for developing heavy vehicle software systems by, e.g., Volvo Construction Equipment[2] (VCE). When using the Rubus CM and OS, all resource allocation of the application and the operating system is done pre-run-time. To facilitate this, information from an executing system can be downloaded using after-market tools.

The temporal properties needed to obtain static timing analysis and schedule generation, Best-Case Execution Time (BCET) and Worst-Case Execution Time (WCET), are monitored by the Rubus OS during runtime. Apart from the temporal aspects of the software, maximum stack usage for each thread and the peak usage of, e.g., queues can be monitored using Rubus. The OS also gives support for monitoring the CPU utilisation.

In multi-threaded embedded software, various types of relations, such as precedence and exclusion relations, exist. To be able to guarantee the behaviour of the system with respect to these issues, the Rubus CM includes support for monitoring event traces of the program execution, i.e., the execution order and the release times of the components. This information is dumped on an external interface (e.g., CAN or a serial interface like RS485) during run-time. Since events are related only via time-stamps, this service requires a high-resolution hardware timer to work. There will be a significant amount of data associated with this service, and the accuracy of the log reflects the size of the buffer used to store it.

PECOS[3] (PErvasive COmponent Systems) [20][21] is a collaborative project between ABB

[1] Arcticus Systems, www.arcticus.se/
[2] Volvo CE, www.volvo.com/
[3] The PECOS Project, www.pecos-project.org/

Corporate Research Centre[4] and academia. The goal for the PECOS project is to enable component-based technology for embedded systems, especially for field devices, i.e., embedded reactive systems. The project tries to consider non-functional properties, such as memory consumption and timeliness, very thoroughly in order to enable assessment of the properties during construction time.

Non-functional properties cannot only be attached to components, but also to ports and connectors, e.g., examining the min and max values for an out port, (i.e., a built in sanity check). Since PECOS is developed to support resource constrained embedded real-time systems, scheduling information and memory consumption are crucial properties to monitor. Hence, PECOS enables support for instrumenting components during runtime. Every component is instrumented to extract information about the WCET and its cycle time. The components are also instrumented with respect to their code size and data (i.e., information on the heap).

4. Monitoring Software Components

Although a multitude of component properties are of interest when building reliable and reusable software components, there are some aspects that would significantly help increasing reusability and lower the time spent on integration testing. We have identified four main aspects of interest.

4.1. Temporal Behaviour

Having knowledge of the temporal behaviour of an execution is particularly important for real-time systems. If the worst-case and best-case execution times of a set of reusable components are known, the possibility of successfully predicting the temporal behaviour of the component assembly will radically increase. Also, other execution time metrics, such as average execution time, standard deviation, execution time histogram or other types of statistical representations of component execution time behaviour can be helpful to estimate statistical temporal properties of component assemblies [14].

When considering timeliness for embedded real-time systems, it is important to be able to verify (1) that each component meets its timing requirements, (2) that each node (which is built up from several components) meets its deadlines, and (3) to be able to analyse the end-to-end timing behaviour of functions in a distributed system. In order to make sure that all deadlines are met, temporal analysis is needed.

This type of analysis is performed using schedulability analysis techniques, and requires information about the component's execution time. Ideally, the bounds for worst-case and best-case execution times should be statically computed by an analysis tool; this is the only way to be sure that the execution-time bounds are safe (i.e. guaranteed not to be violated at run-time), see e.g. [5]. Unfortunately, tools for execution-time analysis are immature and few commercial tools exist. Hence, the industrial practice is to rely on measurement of execution-times. However, structured measurement of execution-times is a tedious, error-prone and expensive process, which has to be re-done after each modification to a component. Using monitored components, the correctness of the execution time values can be improved gradually, i.e., the more execution hours, the better the accuracy [14]; this without any extra labour for execution-time measurement.

In general, execution behaviour information is used for schedulability analysis and scheduling. In hard real-time systems, where it is imperative that deadlines are met, deterministic schedulability analysis and scheduling (worst-case assumptions and information regarding execution times) is preferable. However, in practice, many systems would settle for high probabilities instead of absolute deadline guarantees. Therefore, stochastic schedulability analysis and scheduling can be used. Depending on the type of analysis intended, either worst-case or statistical timing metrics should be collected during monitoring.

4.2. Memory Usage

Since we are targeting resource-constrained systems, it is important to be able to analyse the memory consumption and to check the sufficiency of the system memory, as well as the ROM memory. This check should be done pre-runtime to avoid failures during runtime. Memory is allocated in a static (pre-runtime or during run-time initialisation) or a dynamic (run-time) fashion. As mentioned in Section 2.2, dynamic memory allocation is usually not allowed when developing embedded real-time systems. In order to improve the possibility of achieving predictable assemblies, information of static memory allocation (e.g., component binary size) is necessary, but since this information can be provided by means of compiler output, this property is typically not necessary to monitor.

The stack memory, however, is statically allocated, but used in a dynamic fashion. In order not to end up in a stack overflow situation, stack size is often pessimistically over-dimensioned during system configuration. In resource-constrained environments, this might lead to a situation where the high percentage of unused memory leads to

[4] ABB Corporate Research, www.abb.com/

increased requirements on the system hardware. Therefore, monitoring the stack usage per component is most important, since this information can be used to predict the stack usage behaviour of future assemblies. Due to the high criticality of stack overflow, we are not interested in anything but worst-case usage during the execution of the component. However, in a system allowing dynamic allocation, also heap size monitoring would be important.

4.3. Event Ordering

When testing and debugging software, it is often helpful to be aware of the occurrence and ordering of system events, such as mutex- and semaphore operations, message receipts and interrupt occurrences. Using the information provided by an event log, system designers are able to detect improperly synchronised accesses to shared data or illegal pre-emption of non-reentrant code. In addition, by including event monitoring in the component model, we ensure that all components conforming to the model will produce event-trace logs of similar formats. This will reduce the problem of obscure tracing code inserted by system developers.

Using event traces, we can gain substantial insight regarding the internal workings of current assemblies. This information can be used in order to guarantee precedence relations, mutual exclusion and to enhance the efficiency of shared resource usage (e.g., field bus- or third-level storage usage) in future assemblies.

This type of monitoring provide a foundation to include full support for a *replay debugging* method [17][25][26] in the component technology. Replay debugging is a general term denoting methods for recording the execution behaviour of multi-tasking or truly parallel systems in order to use this information to reproduce system failures during debugging. Most replay methods require both event ordering information (such as interrupt, context switch and synchronisation information) and data flow information (such as task state and external input information) in order to reproduce executions. Provided that the assembly infrastructure (e.g., real-time operating system mechanisms) includes support for replay debugging, including sufficient monitoring in the components will ensure that the entire assembly can be debugged by means of execution replay.

4.4. Sanity Check

A sanity check is a way of determining the soundness of the functional operation of a component during run-time with respect to the component input and its current state. In other words, given a specific input, is the corresponding output realistic? During testing, having access to the input values of the component that produce erroneous output facilitates a more efficient process. This type of monitoring could also include properties like Mean-Time Between Failures (MTBF).

Monitoring this during run-time will allow us to store erroneous operations of the component and (hopefully) to correct these errors in future assemblies. If we are unable to correct the faulty component, we could still able to prevent unsafe system behaviour by taking appropriate actions based on knowledge of the errors.

In addition, this type of monitoring could be used to ensure that 3rd party software components provide the service they are supposed to. Typically a component is equipped with a *provided* interface, specifying the services provided by that component, and a *required* interface, specifying the resources needed by the component in order to provide the correct services. Formalising and standardising these interfaces allows for contractual-based component development, where the behaviour of 3rd party components included in the assembly can be specified by contracts. Using sanity checks of the inputs and outputs at the component interfaces allows for run-time contract checking of 3rd party components.

5. Making use of Monitored Information

In Section 1, four key-areas that would benefit from component-level monitor support were listed. Table 1 maps these areas to the four monitored component aspects discussed in this Section. For instance, execution time- and memory information can be used in order to check whether 3rd party components do not violate their required interface (e.g., by memory leaks or deadline misses), and sanity checks can be used to check the provided interface during run-time. By using event ordering and sanity check traces, the observability and ability to easily test and debug the assembly can be considerably enhanced. Regardless of whether replay debugging methods are used or not, event traces are helpful during debugging in order to visualise the behaviour of the component assembly during run-time.

As for certifiable components, all monitoring aspects can be helpful in order to successfully predict the future behaviour of components in different types of assemblies. Including monitoring in a component technology will ensure that all components conforming to that technology will include identical monitoring support. Hence, component properties can be easily compared using standardised means of comparison.

	Execution Time	Memory	Event Ordering	Sanity Checks
Run-time contract checking	x	x		x
Observability			x	x
Debug/Testing			x	x
Certifiable Components	x	x	x	x

Table 1: *Mapping key-areas of interest to key component aspects*

6. Discussion of the Industrial Benefits of using Monitored Components

It is of great interest for embedded-system developers to be able to build predictable component assemblies using 3rd party software components, in the same way as done in the office/Internet domain (e.g. using EJB, COM or CORBA components to develop desktop computer applications) [12].

The main reason that Commercial-off-the-Shelf (COTS) components are not commonly used when developing embedded systems, is that there are no guarantees of the behaviour of the COTS components, especially with respect to the non-functional aspects of the component (e.g. temporal behaviour and memory usage). However, many types of software developing companies, ranging from subcontractors or consulting agencies to Original Equipment Manufacturers (OEMs), would benefit substantially from using certified software components from different suppliers, to compose reliable and predictable software systems (see, e.g., the projects EAST[5] and Autosar[6]).

There are, however, no well-tested and reliable techniques available to achieve this COTS component reuse in a predefined manner, mainly because it is hard to guarantee the temporal and functional behaviour of the 3rd party software components in various environments. It is obvious that subcontractors would gain from buying/selling COTS components, but it is also our strong belief (and also an upcoming industrial requirement, when talking to companies within the vehicular industry) that OEMs will have to consider using 3rd party software components in a larger extent to accomplish customer demands and to achieve cost-effectiveness. Monitoring, as suggested in this paper, is a technique that enables use of 3rd party software, by predicting and analysing the non-functional properties of the software components.

Other important requirements elicited from embedded-system developers are that system testability and debugability must not suffer when introducing software components in the development process [12]. Debugging and testing can be enhanced when using monitored software components, since the event ordering and the sanity check can be used to facilitate, e.g., replay debugging [17] and provides observabiliy of component behaviour.

Analysability of an embedded system requires information about the components with respect to timing and memory usage. As discussed in [12], analysability is considered highly attractive by industry, but the lack of pertinent information often makes it unsuitable to use in practice. The temporal aspects, e.g. worst-case execution-time and best-case execution time) of embedded-systems are essential to be able to schedule the assembly. These properties are very hard to calculate, and pessimistic estimations are often used in practice. However, by monitoring the system timeliness and reusing the information extracted to predict the behaviour of next generation software, analysability can be improved.

An extension of the sanity check can be used to add fault-tolerance. There are two different types of features for fault tolerant components: detectors (used to detect the fault) and correctors (used to correct the fault). System safety can be enhanced, by using the built in sanity-checks proposed in this paper. In a longer perspective, it is also desirable to be able to use the sanity check, and the fault-tolerance mechanisms, in order to be able to analyse quality issues like system reliability and safety.

7. Conclusion and Future Work

In this paper we have proposed monitoring of software components, and reuse of monitored components, as a general approach towards engineering of resource constrained, embedded, distributed, real-time control systems. The concept is general in the sense that it addresses not only the development phase; rather the whole product life cycle, including debugging, testing and maintenance, is considered. The concept also extends well into product-line settings, where components and architectures are reused over a set of related product and product variants.

We have identified four main component-aspects that are of particular interest to monitor: (1) the execution time behaviour of the components, (2) the static and dynamic memory usage, (3) the event ordering of the execution and (4) a sanity check of the components output based on the input. We have also discussed how these aspects can be used to enhance four different key-areas within engineering of embedded systems: (*i*) certifiable components, (*ii*) system-level testing and debugging, (*iii*) run-time contract checking, and (*iv*) observability.

We have provided a summary of the state-of-the-art of monitoring support for component models and presented a brief survey of the prac-

[5] EAST Project, www.east-eea.net

[6] Autosar Project, www.autosar.org

tices used in today's component models for embedded real-time systems.

As for future work, there are a number of issues we would like to address. We intend to look further into the problems of using the same component on top of different hardware platforms, where some old monitoring information might be reused, while other information needs to be discarded on the new platform. Furthermore, the trade-off between minimisation of monitoring memory and CPU usage and the level of detail of monitor information will be investigated. In order to evaluate our ideas, we plan to build a test-bed implementation to verify the benefits of component monitoring.

8. References

[1] Clements P, Northrop L M; *Software Product Lines: Practices and Patterns*; ISBN 0-201-70332-7, Addison Wesley, 2001

[2] Crnkovic I. and Larsson M.; *Building Reliable Component-Based Software Systems*, 2002, ISBN 1-58053-327-2

[3] EC IST-1999-20162, *Component+*, www.-component-plus.org, February 2004

[4] El Shobaki M.; *A Hardware and Software Monitor for High-Level System-on-Chip Verification*; In Proc. IEEE International Symposium on Quality Electronic Design. San Jose, USA, March 2001.

[5] Engblom J, Ermedahl A, Nolin M, Gustafsson J and Hansson H; *Worst-Case Execution-Time Analysis for Embedded Real-Time Systems*; Software Tools for Technology Transfer, 2001

[6] Gait J.; *A Probe Effect in Concurrent Programs. Software – Practice and Experience*, 16(3), March 1986

[7] Gao J., Zhu E. and Shim S.; *Tracking Component-Based Software*; ICSE2000's COTS Workshop: Continuing Collaborations for Successful COTS Development, 2000

[8] Hörnstein J. and Edler H.; *Test Reuse in CBSE Using Built-in Tests*; In Proceedings of Workshop on Component-based Software Engineering, April 2002

[9] Java Enterprise Beans, http://java.sun.com/

[10] Jhumka A., Hiller M. and Suri N.; *An Approach to Specify and Test Component-Based Dependable Software*; In Proceedings of the 7th IEEE International Symposium on High Assurance Systems Engineering (HASE'02), 2002

[11] Microsoft .NET, http://www.microsoft.com/net/

[12] Möller A, Fröberg J and Nolin M; *Industrial Requirements on Component Technologies for Embedded Systems*; International Symposium on Component-Based Software Engineering (CBSE7), Springer Verlag, Edinburgh, Scotland, May, 2004

[13] Möller A, Åkerholm M, Fredriksson J, Nolin M; *An Industrial Evaluation of Component Technologies for Embedded-Systems*; Submitted for publication, available as Technical Report: MRTC report ISSN 1404-3041, ISRN MDH-MRTC-155/2004-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, February, 2004

[14] Nolte T, Möller A and Nolin M; *Using Components to Facilitate Stochastic Schedulability Analysis*; In Proceedings of the WiP Session of the 24th IEEE Real-Time System Symposium, Cancun, Mexico, December, 2003

[15] PACC Project Home Page: http://www-.sei.cmu.edu/pacc/

[16] Thane H, Hansson H; *Testing Distributed Real-Time Systems*; Journal of Microprocessors and Microsystems, Elsevier, 24:463 – 478, February 2001

[17] Thane H, Sundmark D, Huselius J.G. and Pettersson A; *Replay Debugging of Real-Time Systems Using Time Machines*; In Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03), presented at the First International Workshop on Parallel and Distributed Systems: Testing and Debugging (PAD-TAD), pages 288-295, April 2003. ACM

[18] Lundbäck K-L., Lundbäck J., Lindberg M.; *Component based development of dependable real-time applications*; Arcticus Systems, Home Page: http://www.arcticus.se

[19] Rubus OS Reference Manual, General Concepts, Arcticus Systems, Home Page: http://www-.arcticus.se

[20] Genssler T., Christoph A., Schuls B., Winter M., Stich C.M., Zeidler C., Müller P., Stelter A., Nierstrasz O., Ducasse S., Arevalo G., Wuyts R., Liang P., Schönhage B. and van den Born R.; *PECOS in a Nutshell*, PECOS project homepage: http://www.pecos-project.org

[21] Winter M., Genssler T., Christoph A., Nierstrasz O., Ducasse S., Wuyts R., Arévalo G., Müller P., Stich C. and Schönhage B.; *Components for Embedded Software – The Pecos Approach*; Second International Workshop on Composition Languages, In conjunction with 16th European Conference on Object-Oriented Programming (ECOOP) Málaga, Spain, June 11, 2002

[22] CANopen, Home Page: http://www.canopen.org

[23] SAE Standard, SAE J1939, Joint SAE/TMC Electronic Data Interchange Between Microcomputer Systems In Heavy-Duty Vehicle Applications, www.sae.org

[24] NIST Report. The Economic Impacts of Inadequate Infrastructure for Software Testing; May 2002.

[25] LeBlanc T.J. and Mellor-Crummey J.M.; *Debugging Parallel Programs with Instant Replay*; IEEE Transactions on Computers, 36(4):471-482, April 1987.

[26] Tai K.-C., Carver R.H. and Obaid E.E.; *Debugging Concurrent Ada Programs by Deterministic Execution*; IEEE Transactions on Software Engineering, 17(1):45-63, January 1991.