SOCA (2009) 3:239–262
DOI 10.1007/s11761-009-0050-y

SPECIAL ISSUE PAPER

Time-constrained services: a framework for using real-time web services in industrial automation

Markus Mathes · Christoph Stoidner · Roland Schwarzkopf · Steffen Heinzl · Tim Dörnemann · Helmut Dohmann · Bernd Freisleben

Received: 15 January 2009 / Revised: 11 July 2009 / Accepted: 9 September 2009 / Published online: 9 October 2009

Abstract The use of web services in industrial automation, e.g. in fully automated production processes like car manufacturing, promises simplified interaction among the manufacturing devices due to standardized protocols and increased flexibility with respect to process implementation and reengineering. Moreover, the adoption of web services as a seamless communication backbone within the overall industrial enterprise has additional benefits, such as simplified interaction with suppliers and customers (i.e. horizontal integration) and avoidance of a break in the communication paradigm within the enterprise (i.e. vertical integration). The *Time-Constrained Services* (*TiCS*) framework is a development and execution environment that empowers automation engineers to develop, deploy, publish, compose, and invoke time-constrained web services. TiCS consists of four functional layers—tool support layer, real-time infrastructural layer, real-time service layer, and hardware layer—which contain several components to meet the demands of a web service based automation infrastructure. This article gives an overview of the TiCS framework. More precisely, the general design considerations and an architectural blueprint of the TiCS framework are presented. Subsequently, selected key components of the TiCS framework are discussed in detail: the SOAP4PLC engine for equipping programmable logic controllers with a web service interface, the SOAP4IPC engine for processing web services in real-time on industrial PCs, the WS-TemporalPolicy language for describing time constraints, and the TiCS Modeler for composing time-constrained web services into a time-constrained BPEL4WS workflow.

Keywords Industrial automation and control · Real-time · Web services · Service-oriented architecture (SOA) · Time constraints

This research was partially funded by an IBM Real-time Innovation Award 2008.

M. Mathes (✉) · C. Stoidner · R. Schwarzkopf · S. Heinzl · T. Dörnemann · B. Freisleben
Department of Mathematics and Computer Science, University of Marburg, Hans-Meerwein-Str. 3, 35032 Marburg, Germany
e-mail: mathes@informatik.uni-marburg.de
URL: http://www.uni-marburg.de/fb12/vs

C. Stoidner
e-mail: stoidner@informatik.uni-marburg.de

R. Schwarzkopf
e-mail: rschwarzkopf@informatik.uni-marburg.de

S. Heinzl
e-mail: heinzl@informatik.uni-marburg.de

T. Dörnemann
e-mail: doernemt@informatik.uni-marburg.de

B. Freisleben
e-mail: freisleb@informatik.uni-marburg.de

H. Dohmann
Department of Applied Computer Science, University of Applied Sciences Fulda, Marquardstr. 35, 36039 Fulda, Germany
e-mail: Helmut.Dohmann@informatik.hs-fulda.de
URL: http://www.hs-fulda.de/index.php?id=91

1 Introduction

Industrial automation is aimed at monitoring and controlling an industrial plant via hard- and software with minimized human intervention during operation. A well-known example for automation is an assembly line in automobile manufacturing that consists of several devices, e.g. industrial robots or conveyor belts.

A main characteristic of industrial automation is the demand for real-time processing [1]. The notion *real-time*

neither means that a process is completed "fast" nor that its execution corresponds to the real time. Real-time means that a task is completed correctly within a given time constraint, i.e. it meets its deadline. A task whose execution exceeds a given deadline is treated as failed. On the other hand, a task that is completed prior to its deadline has no additional value. Real-time can be further divided into *hard real-time* (the time constraint must *always* be satisfied) and *soft real-time* (the time constraint is satisfied *most of the time*).

The *first generation* of industrial automation was hardware-dominated [2]. The processing logic was realized as a hard-wired circuit within a switching cabinet. The main disadvantage of hard-wired processing logic is its inflexibility with respect to changes of the monitored and controlled production process. Even simple changes within the production process may result in a revision of the entire hard-wired processing logic. Another disadvantage is its monolithic nature, i.e. an automation engineer is not able to define reusable modules that implement a part of the processing logic. The monolithic nature finally results in processing logic that is hardly scalable. An extension of the production process often requires a redevelopment of the entire processing logic.

To avoid the inflexibility of hard-wired processing logic and to enable a fast adaptation to changes within the production process, the *second generation* of industrial automation was software-dominated [2]. The software-dominated automation approach results in a distributed, hierarchical monitoring and control of the manufacturing process. The manufacturing process is decomposed in disjunctive steps that are called *production cells*. Each production cell contains several manufacturing devices that are controlled by a *programmable logic controller* (*PLC*). Several production cells are monitored by an *industrial PC* (*IPC*).

A PLC is specialized hardware that heavily differs from common desktop PCs. Even though modern PLCs become more and more powerful, they are not comparable to desktop PCs with regard to computing power or main memory. A PLC has several input/output modules that are connected to sensors (e.g. light barrier) and actuators (e.g. conveyor belt), respectively. PLCs operate in a loop according to the *input-processing-output* (*IPO*) model where each step is processed in a predetermined time. The PLC reads input data from its sensors, computes the necessary reaction based on a given rule base, and uses its actuators to react. An IPC is comparable to a regular desktop PC with respect to computing power and main memory, but the case design is much more robust to resist the hostile physical conditions in the manufacturing layer, e.g. temperature, vibration, or dust and dirt. Often, standardized operating systems like Microsoft Windows or Linux are used to run IPCs.

The main disadvantage of the second generation of industrial automation is the use of a vast number of different interfaces. Both, the interface between the IPC and higher layers and the interface between IPCs and PLCs are not standardized, but vendor-specific. Additionally, PLCs from different vendors offer different interfaces and protocols, increasing the communication complexity further. Consequently, the interconnection of the production process with higher layers requires expert knowledge by automation engineers and software developers from higher layers, leading to additional costs.

The *third generation* of industrial automation is currently investigated by several research projects [3–6] and primarily focuses on the use of open, standardized protocols for the interconnection of manufacturing devices and a higher flexibility of the entire organization of an industrial enterprise. Consequently, the third generation of industrial automation can be regarded as interaction-dominated. Whereas the concrete technical realization is discussed in various interest groups and standardization committees, the fundamental requirements for future industrial automation solutions are commonly accepted [7]:

- **interoperability:** All manufacturing devices have to offer a standardized interface based on a common technology to avoid breaks in the communication paradigm.
- **horizontal integration:** The communication with other enterprises, especially with suppliers and customers, has to be simplified.
- **vertical integration:** Not only the communication with other enterprises, but also the communication within the enterprise ranging from the shop floor up to the top floor, has to be simplified.
- **agility:** The automation system must be easily adaptable to new conditions within the production process caused by changes of the business competition or the raw materials processed.

We suggest to use *web services as a standardized communication backbone* within the entire enterprise to meet these requirements:

- The increasing proliferation of web services simplifies the interaction with suppliers and customers and fosters the horizontal integration of enterprises. In some branches of industry—especially automotive engineering—the use of web services for business-to-business communication is mandatory to stay competitive.
- The seamless use of web services within an enterprise avoids breaks in the communication infrastructure and therefore in the implementation and maintenance of numerous interfaces. This eases vertical integration and enhances the interoperability of software systems within the enterprise by means of open, standardized protocols.
- Today's fast moving market situation requires a flexible adaptation of enterprises. Within an enterprise, the

demand for flexibility results in engineering of new business processes and recurrent reengineering of existing ones. A business process often consists of several steps, e.g. simple basic tasks or further business processes that have to be processed in a specific order. This composite nature of a business process is reflected by web services. A web service may implement a basic task or a complex task (by using several other web services).

In this article, an approach to use web services as a seamless communication backbone in industrial enterprises is presented: the *Time-Constrained Services* (*TiCS*) framework. The TiCS framework is a development and execution environment for time-constrained web services. It enables automation engineers to develop, deploy, publish, compose, and invoke time-constrained web services. TiCS consists of four functional layers: tool support layer, real-time service layer, real-time infrastructural layer, and hardware layer. Selected key components of each of the layers are discussed: the TiCS Modeler for composing time-constrained web services into a time-constrained BPEL4WS workflow, the WS-TemporalPolicy language for describing time constraints, SOAP4IPC for processing web services in real-time on industrial PCs, and SOAP4PLC for equipping programmable logic controllers with a web service interface. The key contributions of this article are the presentation of the entire TiCS framework and a comparative performance evaluation of the SOAP4PLC and SOAP4IPC engines.

The article is organized as follows. In Sect. 2, an architectural blueprint of the entire framework is presented. The SOAP4PLC engine is discussed in Sect. 3, and the SOAP4IPC engine is described in Sect. 4. The WS-TemporalPolicy language for the description of dynamic web service properties is described in Sect. 5. Section 6 presents the TiCS Modeler for the composition of time-constrained BPEL4WS workflows. Selected implementation details of the TiCS framework are discussed in Sect. 7. Section 8 presents an evaluation of the TiCS framework. Related work is discussed in Sect. 9. Section 10 concludes the article and outlines areas for future work.

2 Architecture of the TiCS framework

The TiCS framework consists of four functional layers: tool support layer, real-time service layer, real-time infrastructural layer, and hardware layer. Each layer contains several components to meet the demands of a web service based automation infrastructure. The architectural blueprint of the TiCS framework is outlined in Fig. 1.

2.1 Hardware layer

The hardware layer is the basis for the entire TiCS framework and contains only standardized automation hardware, i.e. IPCs, PLCs, and arbitrary manufacturing devices, to guarantee backward compatibility with existing automation solutions. The technical details of the automation hardware, e.g. hierarchical arrangement or wiring of the manufacturing devices, are hidden by the hardware layer.

A key requirement of the TiCS framework is a web service interface to the hardware layer. Such an interface does not

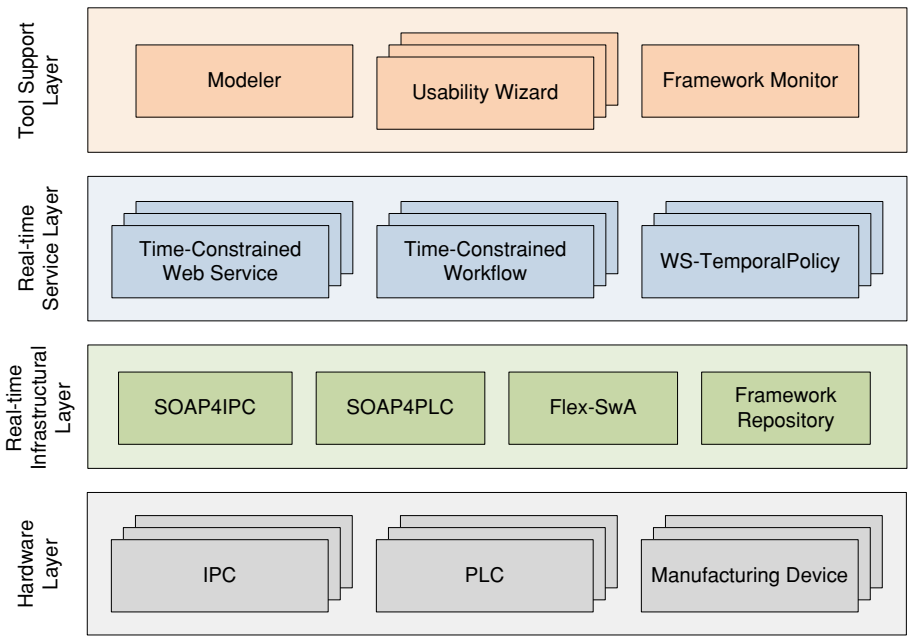


Fig. 1 Architectural blueprint of the TiCS framework

describe use case-dependent protocols or data formats, but how the hard- and software at the manufacturing layer can be empowered with web service functionality. Taking the hierarchical arrangement of production cells into account, three different approaches to realize the web service interface can be distinguished:

1. Only smart devices, i.e. sensors/actuators accessible by web services, are used at the manufacturing layer.
2. The PLCs are enhanced with web service capabilities.
3. Web services are offered by the IPCs. The invocation of a web service operation results in a call to the corresponding control function at the connected PLC.

The first approach does not require additional software, since the smart devices innately contain a web service stack and offer their functionality using web services. The second and third approach require a real-time SOAP engine, i.e. a SOAP engine that processes web services within specific deadlines, tailored to the characteristics of PLCs (e.g. low processing power and main memory) and IPCs (e.g. real-time operating system used), respectively.

The TiCS framework uses both the second and third approach to permit an evolutionary shift from the second to the third industrial automation generation. PLCs with sufficient processing power are extended directly by the web service interface. Otherwise, the IPC is used to export the PLC functionality using web services. The combination of the second and third approach leads to protection of investments and to acceptance by potential users.

2.2 Real-time infrastructural layer

The real-time infrastructural layer contains the SOAP engine for IPCs called *SOAP4IPC* and the SOAP engine for PLCs called *SOAP4PLC*, the *Flex-SwA* data transmission component, and the *Framework Repository*.

The two SOAP engines are tailored to the demands of IPCs and PLCs, respectively. They permit the execution of web services in real-time and the export of PLC control functions using web services. Both engines may be used separately or in combination within a production process. Further details of both engines are discussed in Sects. 3 and 4, respectively.

The invocation of a web service may require specific input parameters. Within industrial automation, these parameters are often bulk binary data, e.g. status/error reports or parts lists in a proprietary binary format. Embedding such data in SOAP messages used for service invocation is not a reasonable approach, because the XML formats SOAP is built on are not suitable to hold (large) binary objects. The Flex-SwA data transmission component offers functionality for efficiently transmitting binary data to permit timely execution of web services. An in-depth look at Flex-SwA is presented in previous publications [8,9].

The Framework Repository provides a global view of all properties concerning the entire TiCS framework: information about all available time-constrained web services and workflows, their worst-case execution time, and the hosts these services/workflows are deployed to. The implementation of the Framework Repository can be based on several technologies, e.g. Universal Description, Discovery, and Integration (UDDI). For performance reasons [10] and for better integration within the TiCS framework, the repository has been implemented using plain web services. Further information about the Framework Repository can be found in a previous paper [11].

2.3 Real-time service layer

The real-time service layer contains *time-constrained web services*, *time-constrained workflows*, and several *temporal policies*.

A time-constrained web service is a standard web service with additional information concerning the worst-case execution time. In principle, there exists a top-down and bottom-up approach to define time-constrained web services. Both approaches are based on the fact that an automation engineer has a priori knowledge about the production process and the required time constraints.

Using the top-down approach, the automation engineer defines the acceptable worst-case execution time for a web service. After having defined the time constraints, the web services are deployed to a SOAP engine responsible for keeping the desired time constraints.

Using the bottom-up approach, the automation engineer deploys a web service to the SOAP engine that profiles the worst-case execution time. The automation engineer uses this information to build the entire production process.

The top-down approach has the main drawback that an automation engineer may define time constraints that can never be kept by the infrastructure, whereas the bottom-up approach results in technically feasible time constraints. For these reasons, the TiCS framework supports the latter approach: the automation engineer starts with implementing a web service, deploys this service to the SOAP4IPC/SOAP4PLC engine, and profiles the time constraints of the web service. If the profiled time constraints are sufficient, the web service can be used. Otherwise, the automation engineer modifies the web service and starts another implementation-deployment-profiling cycle.

A time-constrained workflow is a composition of several time-constrained web services. We use the Business Process Execution Language for Web Services (BPEL4WS) [12] as our workflow composition language due to its proliferation and acceptance. Since in BPEL4WS a workflow is exposed as

a web service, a time-constrained workflow can be handled like a time-constrained web service. More details of time-constrained web services and workflows are discussed in a previous paper [13].

To describe the timing behavior of both time-constrained web services and time-constrained workflows, an automation engineer uses WS-TemporalPolicy [14]. WS-TemporalPolicy can be used to describe arbitrary dynamic properties of a web service. The WS-TemporalPolicy language is discussed in detail in Sect. 5.

2.4 Tool support layer

Potential users of the TiCS framework are automation engineers who are domain experts with no or little experience in developing web services. Therefore, exhaustive tool support is required to ease the implementation, composition, deployment, and publication of time-constrained web services and workflows and the monitoring of the entire TiCS framework. The tool support layer offers these features by means of the *TiCS Modeler*, several *Usability Wizards*, and the *Framework Monitor*.

The TiCS Modeler is a graphical workflow editor that permits the composition of time-constrained web services via BPEL4WS. Details of the TiCS Modeler are presented in Sect. 6.

The Framework Monitor is a graphical interface to the entire TiCS framework. The current system status is outlined using information provided by the Framework Repository. For example, this information includes:

- an overview of deployed and available time-constrained web services/workflows (general description of functionality, worst-case execution time)
- an overview of IPCs/PLCs where SOAP4IPC and SOAP4PLC engines are deployed and their parameterization
- statistical information (number of invocations for each web service/workflow, number of successful/erroneous invocations, downtime/uptime information for relevant hosts)
- the wiring of IPCs, PLCs, and manufacturing devices for service deployment

Details of the Framework Monitor are discussed in a previous paper [11].

The development process of a time-constrained web service consists of three steps: implementation, deployment, and publication. These tasks are supported by three different Usability Wizards that are implemented as Eclipse (http://www.eclipse.org/) plug-ins:

- **Real-time Service Creation Wizard:** The implementation of time-constrained web services is supported by a wizard that helps the engineer to define a Java class with several methods. A class template is generated that easily can be completed by the automation engineer.
- **Real-time Service Deployment Wizard:** After having implemented a time-constrained web service, the service has to be deployed to an instance of the SOAP4LC or SOAP4IPC engine, respectively. The deployment process requires engine-specific knowledge. To ease the deployment process, the automation engineer is supported by this wizard.
- **Real-time Service Publishing Wizard:** This wizard eases the publication of a time-constrained web service in the Framework Repository.

The Usability Wizards are discussed in a previous paper [11].

2.5 Limitations

The TiCS framework is targeted towards the use of web services at the manufacturing layer. Consequently, the use of TiCS requires the general agreement to use web services as a distribution/communication technology. If another technology, e.g. messaging-based publish/subscribe, is desired, TiCS can only be used with restrictions. In such an environment, gateways become necessary to interconnect the different communication paradigms.

The TiCS framework presumes a real-time network infrastructure for data transmission. There are several interconnection networks with deterministic timing behavior, such as, for example, EtherCAT (http://www.ethercat.org/) or Profinet (http://www.profibus.com/pi/). The selection of an appropriate deterministic interconnection network is out of scope of this article. Additionally, the SOAP4IPC engine requires a completely real-time enabled software stack at the IPC consisting of a real-time operating system and a real-time JVM.

The TiCS framework does *not* support smart devices, i.e. web service enabled manufacturing devices, due to two reasons: (1) up to now, only prototypical smart devices are available in the manufacturing domain; (2) existing manufacturing devices can only be replaced by new ones with potentially high costs.

3 Web services for PLCs using SOAP4PLC

This section presents SOAP4PLC [15], a SOAP engine for programmable logic controllers (PLCs). SOAP4PLC is a key component of the real-time infrastructural layer and permits the execution of web services on PLCs in real-time. It offers a low memory footprint to respect the restricted computational power of current PLCs. Furthermore, it hides the web

service details from the automation engineer who is normally not familiar with service-oriented concepts. Additionally, it allows a seamless integration of web services into a PLC application with respect to the IEC-61131-3 PLC programming standard [16].

3.1 Industrial automation using PLCs

A PLC has several input/output modules that are connected to sensors and actuators, respectively. A sensor collects input data from the production area, whereas an actuator allows to manipulate the production process. To control a manufacturing device, the PLC reads the data from the sensors, computes the necessary reaction using a PLC application, and uses its actuators to react. The PLC application is implemented by a domain expert—the automation engineer—who maintains the production process. Most PLC vendors provide a proprietary integrated development environment allowing to implement the PLC application with respect to the IEC-61131-3 standard.

An IEC-61131-3 compatible PLC application is organized into several modules, called *program organization units* (POUs). A POU consists of multiple expressions of one of the programming languages defined by IEC-61131-3 (e.g. Function Block Diagram (FBD), Structured Text (ST), etc.). POUs may also contain calls to other POUs. A POU is defined by a unique name, several input and output variables and an instruction block. The execution of a POU is embedded into the so-called *input-processing-output* (IPO) cycle of the surrounding POU, i.e. first the values for the input variables are read, then the instruction block is executed, then the new calculated output values are written to the connected POUs.

For example, consider a PLC that is used to control the movement of a carriage. The PLC has two physical inputs `in1` (trigger new positioning of the carriage) and `in2` (define the new position) and one physical output `out1` (a lamp for signalling that the new position has been reached).

Figure 2 shows the POU `Axis` that realizes the described PLC application using the IEC-61131-3 programming language FBD. The POU consists of the on-delay timer POU `tonPos` that delays the input `in1` for the time given on `PT` and finally forwards it to `startPositioning` of the POU `axisControl`. On a rising edge on `startPositioning`, `axisControl` moves the carriage to the position given on input `in2`. When the position is reached, output `out1` becomes `TRUE`. In this example, the on-delay timer should assure that the analog value on `in2` has a stable state when the process starts. The program POU `Axis` is attached to a PLC task of the application. The PLC task executes the attached POU cyclically with a defined interval time. In every cycle, all concerned POUs (here `tonPos` and `axisControl`) are executed to read their inputs, do some processing and write their outputs.

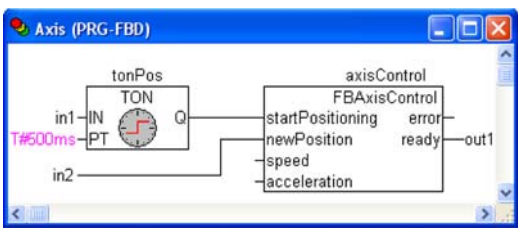


Fig. 2 A PLC control program for carriage movement

3.2 Sequence-controlled web services

The use of web services in combination with PLC applications is complicated by the fundamentally different processing paradigms of both technologies. A PLC application consists of several cyclic tasks, i.e. infinitely running IPO cycles. In contrast, a web service offers several operations that can be invoked at an arbitrary time. To permit the interaction of the PLC control application and the web service interface, the web services must write the inputs and read the outputs of the POUs. This leads to several problems regarding synchronization, data flow and real-time scheduling. These problems are solved using so-called *sequence-controlled web services* offered by SOAP4PLC.

A sequence-controlled web service is a regular web service with a specialized processing flow adapted to the PLC programming paradigm. From the client's point of view, a sequence-controlled web service is not distinguishable from a regular web service. An operation of a sequence-controlled web service is represented by a function block POU called *SOA function block*. The SOA function block forwards the input parameters of a SOAP request message to the succeeding function block. On the other hand, the SOA function block forwards the data from the preceding function block to the SOAP4PLC engine for the use as return parameters within a SOAP response message.

Additionally, the SOA function block contains a boolean chipselect input (`cs-in`) and chipselect output (`cs-out`). On an incoming request, the SOAP4PLC engine sets the chipselect output to `TRUE`. In contrast, the PLC application sets the chipselect input to `TRUE` when the operation has been processed. Figure 3 shows the SOA-ready implementation of the carriage example. The SOA function block `moveTo` is offered as a web service. Its `cs_out` output triggers `startPositioning` on an incoming request. The new position is provided by `position` that comes from the input parameters of the SOAP request message. The `cs_in` input is triggered by the `ready` output of POU `axisControl`, i.e. the position has been reached and the carriage is ready for new commands. When `cs_in` is `TRUE`, a response will be sent to the web service consumer.

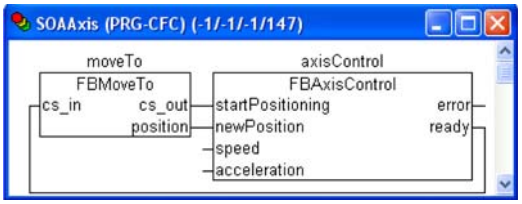


Fig. 3 Example of a simple SOA-ready PLC application

For each SOA function block, SOAP4PLC generates a unique web service operation. The signature of the operation is generated according to the signature of the SOA function block. On start-up, all required web services are deployed automatically. A WSDL description is also generated and provided via HTTP.

The SOAP4PLC engine allows an automation engineer to use standard automation tools and languages (according to IEC-61131-3) for the definition and implementation of web services on a PLC. Thus, the automation engineer operates in his/her well-known development environment. Since the SOAP4PLC engine offers a standardized service-oriented interface based on web services, business engineers can use their well-known tools to access the services provided by the PLC.

4 Web services for industrial PCs using SOAP4IPC

This section focuses on SOAP4IPC, the TiCS real-time SOAP engine for industrial PCs [17]. SOAP4IPC is a key component of the real-time infrastructural layer and permits the execution of web services on an IPC in real-time, i.e. a web service is executed within a predefined time constraint. To the best of our knowledge, SOAP4IPC is the first profiling-and-monitoring based real-time SOAP engine for IPCs presented in the literature. The core functionality of the engine—execution of web services in real-time—is extended by versatile functions that ease the work of developers and administrators, such as automated deadline calculation and hot deployment/undeployment of web services.

4.1 Design considerations

Two important characteristics for the design of a real-time SOAP engine are the *mode of operation*, i.e. how time constraints are guaranteed, and the *level of concurrency*, i.e. how many SOAP messages are processed concurrently.

4.1.1 Mode of operation

The mode of operation may be either an analytical or a profiling-and-monitoring approach:

- **analytical approach**: The timing behavior of the SOAP engine is determined analytically during the development process, i.e. source code analyzers are used to determine the worst-case execution time for each statement, method, and class within the engine.
- **profiling-and-monitoring approach**: The timing behavior of the engine is determined experimentally after the development process has taken place. The obtained results are used to monitor the engine during runtime and compensate potentially occurring deadline violations.

The main advantage of the analytical approach is the total avoidance of deadline violations. Since the worst-case execution time is calculated based on each statement, it is impossible to violate the determined deadlines for the overall software system. Unfortunately, worst-case execution time calculation based on each statement is challenging (consider, for example, conditional branches based on the input data) and often impossible for large software systems. The use of virtual machine-based programming languages like Java and closed-source operating systems further complicate the analysis.

The profiling-and-monitoring approach produces information about feasible platform- and use case-dependent deadlines. These deadlines are used at runtime to monitor the actual timing behavior of the engine. Deadline violations are automatically logged and handled. In safety-critical environments like industrial automation, the profiling-and-monitoring approach is more suitable than the analytical approach, since a completely analytical determination of the timing behavior in such domains is either not possible or not sufficient. Provided that the profiling step was performed with sufficient accuracy, almost all deadlines can be met. The missed deadlines are automatically compensated or result in a defined fault.

SOAP4IPC is based on the second approach. After installing the engine on a specific target platform, two profiling steps are necessary to determine the timing behavior of the engine. The first profiling step—*engine profiling*—determines the latency introduced by the engine, i.e. the worst-case delay until an incoming SOAP invocation is processed. The second profiling step—*service profiling*—determines the worst-case execution time for each web service. Both profiling steps are only conducted once. Based on the results of both profiling steps, the worst-case execution time is calculated for each web service with respect to the engine configuration and the target platform.

4.1.2 Level of concurrency

The level of concurrency may be either strictly sequential or concurrent processing of incoming messages:

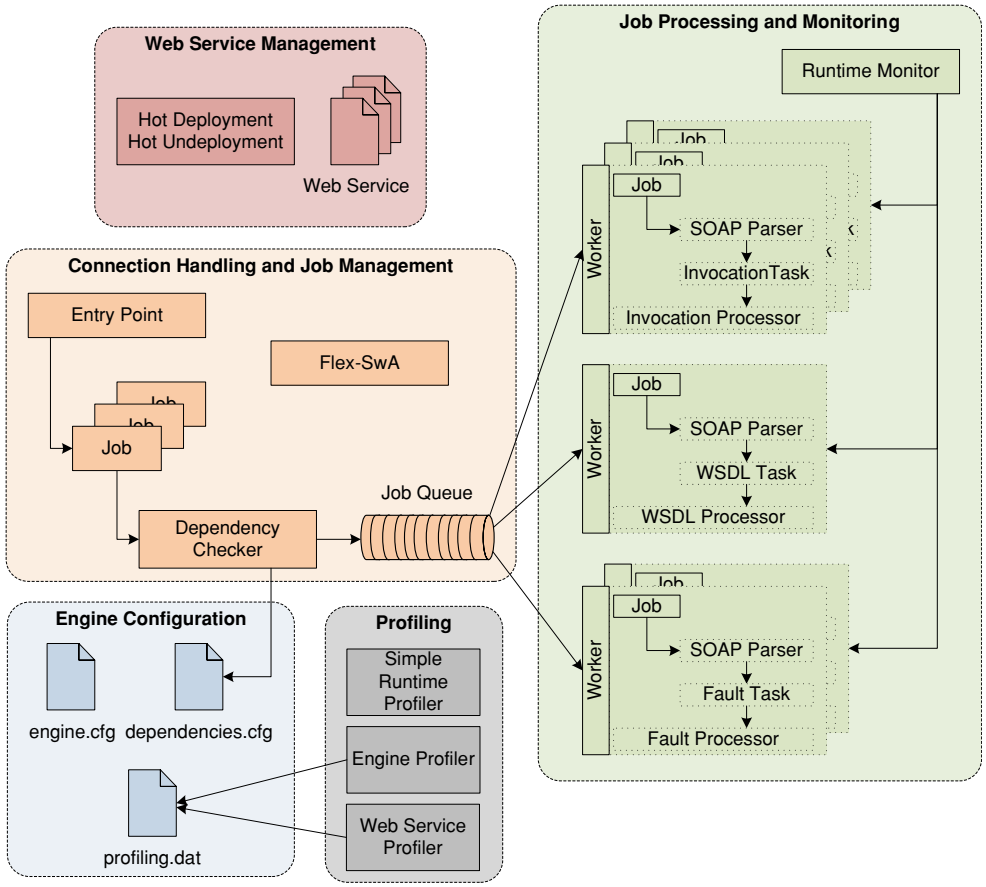


Fig. 4 Architecture of the SOAP4IPC engine

- **sequential processing**: The incoming SOAP messages are processed strictly sequentially, i.e. in the order of their arrival.
- **concurrent processing**: The incoming SOAP messages are processed concurrently, i.e. the processing of several SOAP messages overlaps.

Sequential processing can be easily implemented and avoids race conditions. Consider a web service to control a hoisting platform offering two operations `lower` to lower and `lift` to lift the platform. Using sequential processing, either the invoke message for `lower` is processed prior to the invoke message for `lift` or vice versa. Using concurrent processing may lead to an interleaving of both operations and therefore to an undefined behavior of the hoisting platform.

The SOAP4IPC engine implements concurrent processing of incoming SOAP messages for three reasons. First, immediate reaction to important incoming messages like emergency stops is only possible using concurrent processing. Second, concurrent processing results in a higher throughput of SOAP messages from the perspective of the clients. Third, concurrent processing allows to guarantee shorter deadlines, which broadens the applicability of the engine. Note that concurrent processing may degrade to sequential processing if many dependencies among the involved web services exist.

4.2 Architecture and functional components

The SOAP4IPC engine is a generic, multi-threaded, real-time enabled queuing system consisting of several functional components, as shown in Fig. 4. Its key characteristics are: deterministic timing behavior (real-time processing) and adoption of an arbitrary processing logic.

The deterministic timing behavior, obviously depending on the performance of the hardware used, is guaranteed by means of a profiling-and-monitoring approach. The results of the engine profiling and service profiling runs are used to calculate the deadlines for each service operation and are stored persistently. Using these profiling data (maximum engine delay, worst-case execution time of each service operation), the runtime monitor component checks the current execution

time against the profiling data and takes appropriate action if a deadline violation occurs.

The second main feature of the engine—adoption of an arbitrary processing logic—is realized via replaceable `Parser`, `Task`, and `Processor` objects (shown as dotted lines in Fig. 4). The incoming data is interpreted by a specific parser that depends on the field of application. The engine presented here uses a `SOAPParser` to interpret incoming messages as SOAP messages, i.e. the parser implements the SOAP protocol, more precisely SOAP over HTTP. Depending on the incoming message, it uses an `InvocationTask` (the incoming message is a service invocation), `WSDLTask` (the incoming message requests a WSDL description of a service), or `FaultTask` (the incoming message is incorrect, e.g. unknown target service or operation) to encapsulate all relevant information as an object. For each kind of task, there is a specific processor that knows how this task is processed.

The engine operates in an event-driven manner, i.e. the `EntryPoint` waits for new incoming connections and encapsulates them in `Job` objects. Since the engine supports concurrent processing of jobs, the dependencies among the jobs are checked by the `DependencyChecker`. If some jobs cannot be processed concurrently, the `DependencyChecker` delays their processing by means of an internal buffer. The uncritical jobs are put in the `JobQueue`. The `JobQueue` is a circular queue whose maximum capacity is defined in the engine configuration. The `JobQueue` is processed by several workers that are instantiated during engine startup and are passively waiting in a worker pool. When a new `Job` arrives in the `JobQueue`, one of the available workers is activated to process the `Job`. The `Job` resides within the `JobQueue`, if no worker is available.

The entire engine configuration is read during start-up from a configuration file (engine.cfg). This configuration file contains several parameters as name/value-pairs, e.g. a listen port for incoming connections, maximum size of incoming SOAP messages, hot deployment/undeployment delay. The engine configuration is read only once at start-up and offered as a singleton to all engine components.

Another feature of the engine is support for hot deployment/undeployment of web services, i.e. a new web service can be deployed to the engine and an existing web service can be undeployed from the engine without restart.

5 Description of dynamic web service properties using WS-TemporalPolicy

In this section, the WS-TemporalPolicy language is introduced to describe temporal web service aspects that define dynamic, non-functional properties of web services [14]. WS-TemporalPolicy enables a web service developer to attach a validity period to the properties described in a WS-Policy by means of an expiration date, a start and end date, or a duration. Additionally, an event/action-mechanism permits the description of dependencies between several WS-TemporalPolicies and/or WS-Policies.

5.1 Dynamic web service properties

There are several web service specifications that provide a rich, well-defined set of features for loosely-coupled and standardized applications. These specifications are highly extensible to cover requirements that the authors of the specifications could not anticipate.

However, this does not hold for properties concerning the timing behavior of a web service. Since the workload of the infrastructure directly influences the timing behavior of the web services, these properties may vary over the time. Consequently, there is a need to describe the *dynamic properties* of a web service.

Up to now, these properties can only be provided in the functional layer of the web service by an operation, i.e. an operation is used by a consumer to retrieve the current timing behavior. However, conceptually, the timing behavior of a web service should be provided as part of the web service's metadata, i.e. using an appropriate policy language.

To describe those dynamic properties, we propose a new policy language called *WS-TemporalPolicy*. A WS-TemporalPolicy extends a WS-Policy [18] by temporal aspects and permits to describe dependencies between several WS-TemporalPolicies and/or WS-Policies.

5.2 Structure and use of WS-TemporalPolicy

The proposed WS-TemporalPolicy language permits to define the validity period of a WS-Policy or another WS-TemporalPolicy. This can be done by using the elements `expires`, `startTime` and `endTime`, or `duration`. The `expires` attribute defines how long a WS-TemporalPolicy is valid by providing an end time (via XML Schema's [19] "dateTime" data type), whereas the `startTime` and `endTime` attributes define a time slot during which the policy is valid. The `duration` attribute is used to specify a relative amount of time for the validity of the policy (via XML Schema's "duration" data type). Each WS-TemporalPolicy has a `name` attribute that defines the unique name of this policy (via XML Schema's "anyURI" data type) and an optional `keywords` attribute that eases the retrieval of a WS-TemporalPolicy from a policy repository. A WS-TemporalPolicy is linked to a WS-Policy or another WS-TemporalPolicy using the `policyRef` attribute and to a WSDL description of a service using the `serviceRef` attribute.

Table 1 Overview of events concerning WS-TemporalPolicy

Event	Description
`onActivation`	The `onActivation` event occurs when a WS-TemporalPolicy is activated. The referenced policies are attached to the WSDL description of the referenced web service.
`onExpiration`	The `onExpiration` event occurs when the validity period of a WS-TemporalPolicy expires.
`onRenewal`	The `onRenewal` event occurs when a WS-TemporalPolicy is renewed, i.e. its validity period is modified.
`onDeactivation`	This event occurs when a WS-TemporalPolicy is deactivated. As a result, the referenced WS-Policy is detached from the referenced WSDL description.

Table 2 Overview of actions concerning WS-TemporalPolicy

Action	Description
`activate`	A WS-TemporalPolicy is activated, i.e. an `onActivation` event occurs.
`renew`	The validity period of a WS-TemporalPolicy is modified, i.e. an `onRenewal` event occurs.
`deactivate`	A WS-TemporalPolicy is deactivated, i.e. an `onDeactivation` event occurs.

It is also possible to activate/renew/deactivate a WS-TemporalPolicy depending on another WS-TemporalPolicy using *actions* and *events*. Dependencies between WS-TemporalPolicies are described by the definition of an `event` element and a corresponding `action` element. Table 1 gives an overview of possible events, whereas Table 2 gives an overview of possible actions.

The event/action-mechanism enables the definition of complex policy dependencies that can be visualized via a dependency tree. An example tree with five WS-TemporalPolicies and five WS-Policies is shown in Fig. 5. The events are written in *italics*, whereas the corresponding actions are written in **bold**. As shown in the dependency tree, the use of WS-TemporalPolicy induces several logical layers ranging from static to dynamic with regard to the validity period. In the static web service layer, the operations are located. Above the static web service layer, the dynamic meta web service layer is located that can be divided into n sub layers. Each sub layer handles a different temporal dimension. At the top layer, there might be a WS-TemporalPolicy that defines its (long-lasting) validity period by referencing itself and handles other WS-TemporalPolicies. These again could manage other WS-TemporalPolicies or WS-Policies that are, for example, valid for months, weeks, days and so on. In this way, different layers can be built to enable a fine-grained (hierarchical) management of the validity of policies.

The use of events and actions is exemplified in Listing 1. The defined sample temporal policy `sampleTP3` does not influence a concrete WS-Policy or web service (though this would be possible as well), but the WS-TemporalPolicies `sampleTP1` and `sampleTP2`. It defines, that on its activation the `sampleTP1` is also activated, whereas the `sampleTP2` is deactivated. On its deactivation, the `sampleTP1` is also deactivated, whereas the `sampleTP2` is activated. Furthermore, a modification of the validity period of this WS-TemporalPolicy results in a modification of the validity period of `sampleTP1` and `sampleTP2`.

6 Composition of BPEL4WS workflows using the TiCS modeler

This section presents the TiCS Modeler that supports the visual composition of existing web services to a more powerful, value-added workflow using the de-facto industry standard for workflow composition: the Business Process Execution Language for Web Services (BPEL4WS) [12]. The TiCS Modeler is an extension to our previously published Domain-adaptable, Visual Orchestrator (DAVO) [20].

6.1 Timing behavior of BPEL4WS activities

A BPEL4WS workflow is composed of several activities that can be distinguished into basic activities and structured activities. A *basic activity* is used, for instance, to invoke a web service or to copy the value of one variable to another. A *structured activity* contains an arbitrary set of basic or structured activities and is used to model the control flow within a business process. BPEL4WS offers 8 basic activities (receive, reply, invoke, assign, throw, terminate, wait, and empty) and 7 structured activities (sequence, switch, while, pick, flow, scope, and compensate).

In the following, a formal derivation of the timing behavior of two selected BPEL4WS activities, namely invoke and sequence, is presented. The timing behavior of all activities is analyzed in a previous paper [13].

For industrial automation purposes, the worst-case execution time (wcet) and average execution time (aet) of each activity are of particular interest. The worst-case execution time defines an upper bound for hard real-time processing, whereas the average execution time indicates possible deadlines for soft real-time processing.

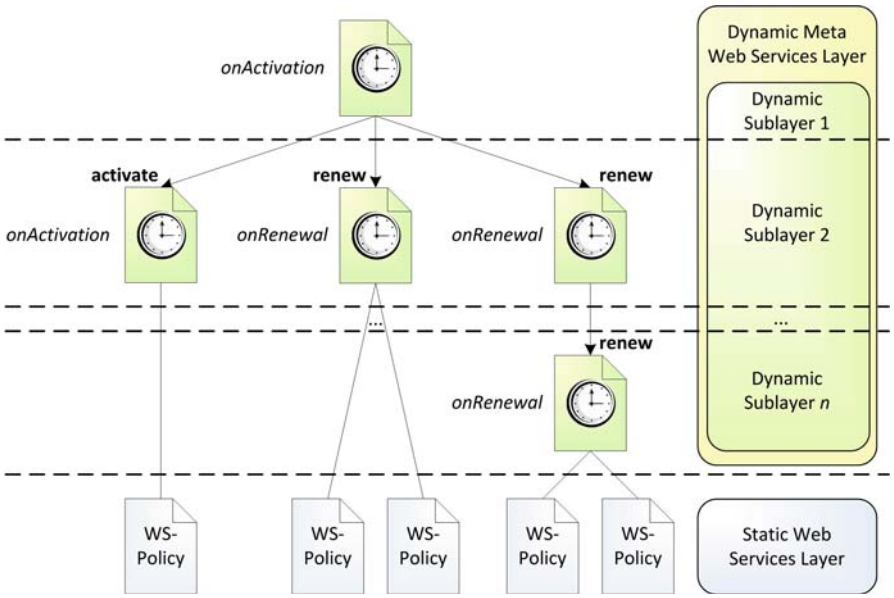


Fig. 5 Dependency tree for several WS-TemporalPolicies and WS-Policies

Listing 1 Example of a WS-TemporalPolicy that affects other WS-TemporalPolicies.

```
<temporalPolicy name="http://fb12.de/sampleTP3">
  <expires>2009-01-01T00:00:00</expires>
  <onActivation>
    <activate ref="http://fb12.de/sampleTP1"/>
    <deactivate ref="http://fb12.de/sampleTP2"/>
  </onActivation>
  <onRenewal>
    <renew ref="http://fb12.de/sampleTP1">
      <expires>2009-07-01T00:00:00</expires>
    </renew>
    <renew ref="http://fb12.de/sampleTP2">
      <startTime>2009-07-01T00:00:00</startTime>
      <endTime>2009-09-01T00:00:00</endTime>
    </renew>
  </onRenewal>
  <onDeactivation>
    <activate ref="http://fb12.de/sampleTP2"/>
    <deactivate ref="http://fb12.de/sampleTP1"/>
  </onDeactivation>
</temporalPolicy>
```

6.1.1 The invoke activity

An invoke activity is used to invoke a web service in a synchronous request/response or an asynchronous one-way fashion. To send input data to and receive output data from the invoked web service, an input and output variable must be defined, respectively.

The worst-case execution time for a successful synchronous invoke is the sum of the transmission time of the input parameters for the invoked web service, the processing of the target web service, and the transmission of the result of the invoked web service (see Eq. 1).

$$\begin{aligned}&\text{wcet (invoke)}\\ &:= \begin{cases} \text{snd}_{\text{wcet}}\ (\text{input parameters}) + \\ \text{wcet}_{\text{service}}\ (\text{input parameters}) + \\ \text{rcv}_{\text{wcet}}\ (\text{return value}) & \text{, synchronous invoke} \\ \text{snd}_{\text{wcet}}\ (\text{input parameters}) & \text{, asynchronous invoke} \end{cases}\\ &\text{aet (invoke)}\\ &:= \begin{cases} \text{snd}_{\text{aet}}\ (\text{input parameters}) + \\ \text{aet}_{\text{service}}\ (\text{input parameters}) + \\ \text{rcv}_{\text{aet}}\ (\text{return value}) & \text{, synchronous invoke} \\ \text{snd}_{\text{aet}}\ (\text{input parameters}) & \text{, asynchronous invoke} \end{cases}\end{aligned} \tag{1}$$

We use a function snd that maps the size of the input parameters to a duration and a function rcv that maps the size of the result to a duration. Both functions are specific to the workflow engine used to execute the workflow. Obviously, the execution time of the web service also depends on the size of the input parameters. Therefore, we use service-specific functions ($\text{aet}_{\text{service}}$ and $\text{wcet}_{\text{service}}$) to calculate the average and worst-case execution time depending on the size of the input parameters. The average/worst-case execution time for an asynchronous invoke—only consisting of the transmission time of the input parameters—is defined using the snd function only.

6.1.2 The sequence activity

A sequence activity contains an arbitrary set of activities that are processed in the given order.

The average and worst-case execution time of a sequence activity with n nested activities are defined as the sum of the average and the sum of the worst-case execution times of all these activities, respectively (see Eq. 2).

$$\begin{aligned} \text{wcet (sequence)} &:= \sum_{i=1}^{n} \text{wcet}\left(\text{activity}_i\right) \\ \text{aet (sequence)} &:= \sum_{i=1}^{n} \text{aet}\left(\text{activity}_i\right) \end{aligned} \tag{2}$$

6.2 Calculation of time constraints

The TiCS Modeler computes the average and worst-case execution times of individual activities and the entire workflow using the equations shown partially in the previous section. Therefore, the TiCS Modeler uses an analytical, bottom-up computation approach. An analytical approach is preferable compared to a profiling-and-monitoring approach since BPEL4WS only offers a small set of activities. Using a bottom-up computation approach, i.e. propagating time constraint relevant property changes from individual activities up to the overall workflow, the TiCS Modeler is able to give immediate feedback to the developer, i.e. reporting any time constraint violation.

Many of the equations used internally by the TiCS Modeler to calculate the average and worst-case execution time require further input from an automation engineer. The TiCS Modeler automatically inserts some of the missing parts: (1) the engine configuration that—among other things—configures the snd_{aet}, snd_{wcet}, rcv_{aet}, and rcv_{wcet} functions, is entered by the automation engineer once and reused for every workflow, (2) the functions used to compute the service execution time ($\text{aet}_{\text{service}}$ and $\text{wcet}_{\text{service}}$) are fetched from the Framework Repository, and (3) the amount of data involved in activities like invoke is computed almost automatically.

The computation of the amount of data involved in some activities requires only little input by the automation engineer. For all activities receiving data (receive, pick, and invoke), the estimated amount of received data has to be specified. In addition, for a specific type of the assign activity that assigns only a part of a (message type) variable to another variable, the automation engineer has to estimate the size of the copied part (relative to the source variable). With this information, the TiCS Modeler is able to calculate the amount of data involved in any activity in a workflow. Additionally, it can easily be checked whether the time constraint is held for greater amounts of input data, by just changing the values at the initial receive activity. The complete workflow is then re-verified for compliance with the constraints.

For other activities, the automation engineer has to provide further input manually. This includes hints on the number of iterations of while activities (if it depends on information available only at runtime), the distribution of incoming messages and average waiting times of pick activities, as well as the distribution of the values used as condition in switch activities.

Time constraints for the average or worst-case can be applied to the overall workflow or dedicated structured activities. The latter allows fine-grained modeling and verification of time constraints. This simplifies the isolation of the problem, if time constraint violations occur.

7 Implementation

This section presents implementation details of the following components and features of the TiCS framework: SOAP4IPC (engine- and service-profiling, monitoring of job execution time), data model of the TiCS Modeler (for representing execution time related properties).

7.1 SOAP4IPC

In this section, implementation issues of the SOAP4IPC engine are presented. More precisely, the engine and web service profiler and the runtime monitor are presented. The section starts with a short description of the Java Virtual Machine (JVM) used to implement SOAP4IPC.

7.1.1 JamaicaVM

The SOAP4IPC engine has been implemented using Java as the programming language and the aicas JamaicaVM [21] as the real-time JVM. The aicas JamaicaVM is a JVM implementation compatible with J2SE v1.2 supporting the Real-time Specification for Java (RTSJ) v1.0.2 (http://www.rtsj.org/). It is a cross-platform development environment, i.e. a real-time application is implemented on a development

Listing 2 Example output of the `SimpleProfiler`.

```
PROFILING INFORMATION

  Total processed tasks:    10000
    Invocation tasks:       10000 (0.00% exceeded deadline!)
    WSDL tasks:             0
    Fault tasks:            0

  Average processing time:  47.579 msec (47578717 nanosec)
  Standard deviation:        2.164 msec  (2163928 nanosec)
  Min. processing time:     41.993 msec (41993000 nanosec)
  Max. processing time:     99.985 msec (99985000 nanosec)
```

Listing 3 The `handleAsynchEvent` method of `ShutdownHandler`.

```
public void handleAsyncEvent() {
  String profile;
  Profiler profiler = SimpleProfiler.getInstance();
  engineCfg = EngineConfiguration.getInstance();

  if(engineCfg.engineProfilingEnabled()) {
    profiler = EngineProfiler.getInstance();
  }
  else if(engineCfg.serviceProfilingEnabled()){
    profiler = ServiceProfiler.getInstance();
  }

  profile = profiler.getProfile();
  System.out.println("Shutting down...");
  System.out.println(profile);
  System.out.println("Bye!");
  System.exit(0);
}
```

operating system (e.g. Linux, Solaris, or Windows) and built for a specific target operating system (e.g. VxWorks, Real-time Linux, or INTEGRITY) and processor architecture (e.g. PowerPC, x86, or Sparc). Depending on the target operating system, soft real-time constraints (using a non real-time operating system) or hard real-time constraints (using a real-time operating system) can be guaranteed.

7.1.2 Profiling modes

SOAP4IPC supports two different profiling modes: service profiling by the `ServiceProfiler` and engine profiling by the `EngineProfiler`. If the profiling mode is set to "service", the `ServiceProfiler` is used to measure the execution time for each web service operation, whereas if the profiling mode is set to "engine", the `EngineProfiler` is used to measure the engine overhead. Otherwise, the `SimpleProfiler` is used to collect statistical information about all processed SOAP messages. This information is merged and output at engine shut-down. Listing 2 shows an example output of the `SimpleProfiler`.

To recognize engine shut-down, a `ShutdownHandler` is bound to the SIGINT signal, i.e. as soon as the keystroke [CTRL]+[C] appears, the `handleAsyncEvent` method of the `ShutdownHandler` is called (see Listing 3) and the statistical data is written to the console.

7.1.3 Monitoring of job execution time

The execution time of each job within the engine has to be monitored to guarantee its worst-case execution time. In general, there are three possible strategies to realize execution time monitoring:

1. The execution time of each job is monitored at specific locations within the SOAP engine, e.g. before and after web service invocation.

Listing 4 Adding a `OneShotTimer` to a `Task` object.

```
[...]
// calculate deadline
AbsoluteTime deadline = new AbsoluteTime(entryTime.add(
  services.getAllowedTime(service),0));
// configure ExceededHandler
exceededHandler = new ExceededHandler(this);
exceededHandler.setSchedulingParameters(engineConfiguration.
  getParameterSet().getExceededSchedParams());
exceededHandler.setReleaseParameters(engineConfiguration.
  getParameterSet().getExceededRelParams());
// start OneShotTimer
oneShotTimer = new OneShotTimer(deadline, exceededHandler);
oneShotTimer.start();
[...]
```

2. In addition to the worker threads that process jobs, another high-prioritized monitoring thread exists that knows each worker thread and its worst-case execution time. The monitoring thread controls each worker thread and handles deadline violations.
3. For each job, an alarm is set. As soon as the actual execution time exceeds the worst-case execution time, the alarm is raised.

The first strategy can be implemented easily, but may result in a delayed detection of deadline violations. Consider, that the current execution time is measured before and after service invocation to detect a deadline violation by the web service. Although the deadline violation is detected—provided that the web service terminates and does not run indefinitely—the detection may take an arbitrary duration that is unacceptable in the real-time processing domain. Execution time measurement at further locations, i.e. insertion of additional measuring statements within the engine, does not solve the problem. In contrast, the insertion of further execution time measurements leads to a mixup of functional engine/web service and monitoring code.

The second strategy results in a clear design. Only one additional thread is required to monitor the execution time of each job thread within the engine. As soon as a new job enters the engine, this job is registered at the monitoring thread. The monitoring thread works periodically, i.e. it checks the execution time of each registered worker thread in equidistant time intervals for time constraint violations. If a time constraint violation has occurred, the monitoring thread stops the corresponding worker thread. The implementation of this strategy within Java is problematic, since a thread cannot innately be safely stopped by another thread.

The third strategy, which we call *time constraints piggybacking*, leads to a clear design and implementation of execution time monitoring. The SOAP4IPC engine utilizes this strategy by means of a `OneShotTimer` (part of the RTSJ specification) that internally uses the system clock. A `OneShotTimer` takes a deadline and an `ExceededHandler`. The `ExceededHandler` is an `AsyncEventHandler` whose `handleAsyncEvent` method is called as soon as the deadline is violated. When the job is successfully processed within its deadline, the `OneShotTimer` is deactivated and detached from the `Task` object. Listing 4 exemplifies the use of a `OneShotTimer` to monitor the execution time of tasks.

7.2 TiCS modeler

The TiCS Modeler is technically based on the *Domain-adaptable Visual Orchestrator* (*DAVO*) [20], a domain-adaptable, graphical BPEL4WS workflow editor. The key benefits that distinguish DAVO from other graphical BPEL4WS workflow editors are the *adaptable data model* and *user interface* that permit customization to specific domain needs. These benefits were exploited for the implementation of the TiCS Modeler. Due to space restrictions, only selected parts of the implementation of the TiCS Modeler are explained in this section.

The TiCS Modeler extends each BPEL4WS activity with a `wcExecTime` and `avgExecTime` property that stores the worst-case execution time and the average execution time, respectively. For basic activities, the values of these properties depend on the action realized by this activity. For a structured activity, the values of these properties depend on the `wcExecTime`/`avgExecTime` property values of the child activities contained in the structured activity and have to be calculated individually for different activities.

To be able to extend the data model, an implementation of the `IModelExtender` interface has to be provided by the TiCS Modeler. It is used to create the `Element Extensions` for given `Elements`. The three element

Listing 5 The element extension for plain elements.

```
public class ElementExt extends ElementExtension {

  public ElementExt() {
    super(Activator.PLUGIN_ID);
  }

  protected void applyExtension() {
    PropertyGroup group = new PropertyGroup(Activator.PLUGIN_ID);
    group.add(new ElementWCETProp());
    group.add(new ElementAETProp());
    getExtendedElement().addPropertyGroup(group);
  }

  protected void removeExtension() {
    getExtendedElement().removePropertyGroup(Activator.PLUGIN_ID);
  }
}
```

Listing 6 The worst-case execution time property that is applied to all elements that are neither containers nor connected elements.

```
public class ElementWCETProp extends Property<Integer> {

  public static final String ID = "de.fb12.tics.modeler";
  public static final String DESCRIPTION = "worst-case execution time";
  public static final String CATEGORY = "TiCS";

  public ElementWCETProp() {
    super(ID, DESCRIPTION, CATEGORY, Integer.class);
    setValue(0);
  }
}
```

extensions `ElementExt`, `ConnectedElementExt`, and `ContainerElementExt` are almost identical, thus only one is shown in Listing 5. The most interesting method of this class is `applyExtension` that modifies the actual element. In this case, it creates two new properties, an instance of `ElementWCETProp` and `ElementAETProp` respectively, adds these properties to a newly created `PropertyGroup` that again is added to the `Element` (representing a basic activity). A `PropertyGroup` is just a collection of properties to simplify the reuse and handling of groups of properties, i.e. it is possible to remove multiple properties at once by the `removeExtension` method.

Listing 6 shows the worst-case execution time property in its simplest form. It has an ID to address it, a description and category used to identify it within the property view and is of the type Integer. After its creation, it is initialized with the value 0. By default, this value is editable in the property view.

The `ConnectedElements` (representing basic elements with dependencies towards external sources, e.g. partner links) are extended with another property shown in Listing 7. Because the worst-case execution time of a `ConnectedElement` depends on the specific operation it is connected to, this property depends on the operation name. This dependency is modeled by the implementation of the `IPropertyValueDependent` interface that consists of the two public methods `getPropertyValueDependencyIDs` and `relevantPropertyValueChange`. The first method returns a list of the properties this property is depending on. The second method is invoked when the value of one of the depending properties is changed. Since the value of the property is determined automatically, it is not editable.

The `ContainerElements` (representing structured activities) are also extended with a specific property shown in Listing 8. The worst-case execution time of a `ContainerElement` depends on the type of the container (e.g. sequence) and the worst-case execution times of its children. Consequently, this property does not depend on the values of other properties in the same `Element`, but on

Listing 7 The worst-case execution time property that is applied to all `ConnectedElements`.

```
public class ConnectedElementWCETProp extends ElementWCETProp
  implements IPropertyValueDependent {

  public ConnectedElementWCETProp() {
    setEditable(false);
  }

  private final static String[] PROP_DEP_IDS =
    new String[] {ConnectedElement.OPERATION_NAME_PROP};

  public String[] getPropertyValueDependencyIDs() {
    return PROP_DEP_IDS;
  }

  public void
  relevantPropertyValueChange(ValueChangeEvent event) {
    /* retrieve worst-case execution time */
  }
}
```

Listing 8 The worst-case execution time property that is applied to all `ContainerElements`.

```
 public class ContainerElementWCETProp extends ElementWCETProp
    implements IExternalEventDependent {
  private final static Class<IExternalEvent>[]
  EXT_EVT_DEP_CLS = (Class<IExternalEvent>[])
    new Class[] { ContainerChangeEvent.class };

  public Class<IExternalEvent>[] getExternalEventDepencies() {
    return EXT_EVT_DEP_CLS;
  }

  public void relevantExternalEvent(IExternalEvent event) {
    if (event instanceof ContainerChangeEvent) {
      ContainerChangeEvent ccEvent =
        (ContainerChangeEvent) event;

      boolean update = true;

      if (ccEvent.getType() == ContainerChangeEvent.Type.PROPERTY &&
          !((ContainerChildPropertyChangeEvent) event)
            .getEvent().getPropertyName().equals(ID))
          update = false;

      if (update) {
        setValue(getContainerCalculator()
          .calculateWCET(getContainerElement()));
        System.out.println("[" + getPropertySet() +
          "] updated, new value: " + getValue());
      }
    }
  }
}
```

Listing 9 The calculator for sequence activities.

```
public class SequenceCalculator implements IContainerCalculator {

  public Integer calculateWCET(ContainerElement containerElement) {
    Integer result = 0;
    for (Element child : containerElement.getChildren()) {
      result += (Integer) child.getPropertyValue(ElementWCETProp.ID);
    }
    return result;
  }
}
```

property values of other `Elements`. This is called an *external dependency* and realized by implementing the interface `IExternalEventDependent`. This interface defines the methods `getExternalEventDependencies` and `relevantExternalEvent`. The first method returns a list of event classes that this property needs to be notified about. A `ContainerChangeEvent` is fired by a container when a child element is added or removed, whereas the property change in a child triggers a `ContainerChildPropertyChangeEvent` (a subclass of `ContainerChangeEvent`). The latter method is invoked when such an event occurs. In this case, some further tests are necessary to determine whether the value of the property has to be updated.

The actual calculation is done by an implementation of the `IContainerCalculator` interface that is provided by a special factory (not shown here), depending on the type of the container. As an example of such a calculator, the `SequenceCalculator` is shown in Listing 9.

8 Evaluation

This section presents a qualitative and quantitative evaluation of selected parts of the TiCS framework. At first, by means of an experimental setup, it is demonstrated that web service based access to the manufacturing layer is technically feasible by using the SOAP4PLC engine. In the following, the performance of the SOAP4PLC and SOAP4IPC engine on specific target platforms is evaluated. Due to space restrictions, the performance of both engines is exemplified by means of three simple services (`EchoService`, `PowService`, and `MathService`). The `EchoService` offers an operation `echo` that takes a string as input parameter and immediately returns this string as return parameter. The `PowService` offers an operation `pow` which takes three input parameters (*basis*, *exponent*, and *cnt*). This operation calculates $cnt \cdot basis^{exponent}$ and subsequently returns "Done!". The `MathService` offers the operations `add` that returns the sum of its two input parameters and `sub` that returns the difference between the two input parameters.

8.1 Experimental setup

The use of SOAP4PLC and sequence-controlled web services to access the manufacturing process is shown by means of the example from Sect. 3: carriage movement using web services. The experimental setup is shown in Fig. 6 and consists of:

(1) a PLC (based on Beck IPC@CHIP SC143)
(2) a servo controller (Festo SEC-AC-305-PB)
(3) a toothed belt axis (Festo DGE-ZR)

The axis consists of a one-dimensional rail on which a carriage is installed. The carriage is driven by a high voltage servo motor. The servo controller, that manages the power supply, is connected to the PLC via Profibus—an industrial real-time bus.

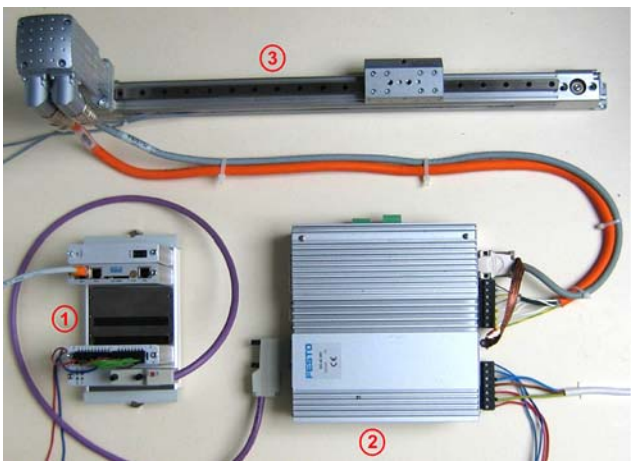


Fig. 6 Overview of the experimental setup

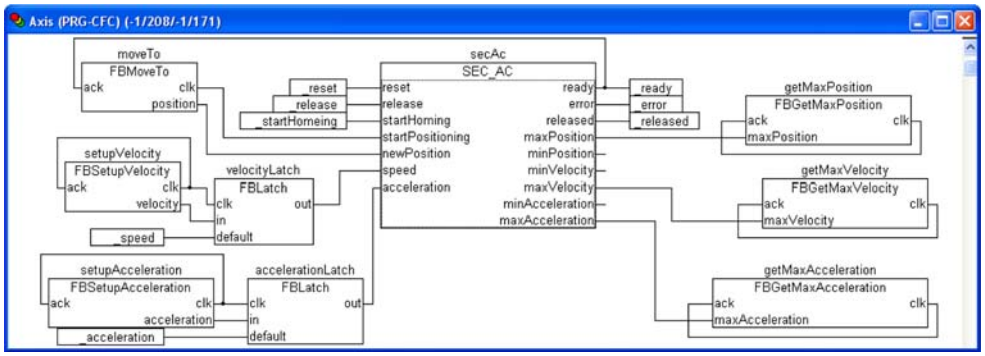


Fig. 7 Service-oriented PLC control program

The PLC control program for this example (Fig. 7) supports a web service with several operations. The POU `secAc` encapsulates the low-level control code for the servo axis. Using its inputs (`reset`, `release`, `startHoming`, etc.) and outputs (`ready`, `error`, `released`, etc.) the remaining POUs (e.g. `moveTo`) can access and move the carriage. These POUs are realized as a SOA function block as introduced in Sect. 3 and represent a specific web service operation. The operation `moveTo` can be used to move the carriage to a specific position. The maximum valid position can be retrieved using the `getMaxPosition` operation. The operations `setupAcceleration` and `setupVelocity` can be used to define the acceleration respectively the velocity of the carriage. To retrieve the maximum acceleration and maximum velocity, the operations `getMaxAcceleration` and `getMaxVelocity` can be used.

8.2 SOAP4PLC

The evaluation of the SOAP4PLC engine was realized on an *IPC@CHIP*-based PLC from Beck similar to the experimental setup in the previous subsection. As the PLC development environment, the IEC 61131-3-based programming system *CoDeSys* (http://www.3s-software.com/) was used.

8.2.1 Profiling I

The latency L to invoke a web service operation directly depends on the WSDL description of the service. The more operations a web service offers (or arguments are required by the operations), the more extensive is the parsing of the corresponding WSDL description and the higher is the latency. As a consequence, the latency has to be measured individually for each web service.

8.2.2 Profiling II

The execution of a service operation consists of two steps: (1) transferring the input parameters from the SOAP request message to the SOA function block; (2) waiting for results from the SOA function block and transferring the results to the SOAP response message. The worst-case execution time WC of both steps is deduced from the cycle time t_{cycle} of the PLC task that processes the corresponding SOA function block. For the first step, the engine needs to wait until the PLC task is in a valid state for parameter transmission—this occurs once in every cycle. For the second step, the engine waits until the SOA function block signals that the result of the operation is available. The required time for this step is defined by the number of cycles #cyc needed by the operations implementation inside the PLC application. Consequently, the worst-case execution time WC is defined as in Eq. (3).

$$\text{WC} = 1 \cdot t_{\text{cycle}} + \#\text{cyc} \cdot t_{\text{cycle}} = (1 + \#\text{cyc}) \cdot t_{\text{cycle}} \tag{3}$$

8.2.3 Deadline calculation

The deadline D is the maximum delay between the request of a service operation and the corresponding response. Using the results of profiling step I and II, the deadline can be calculated. The example below describes the deadline calculation for the simple test services `EchoService`, `PowService`, and `MathService`. For the three services, the measured latencies are identical:

$$L_{\text{EchoService}} = L_{\text{PowService}} = L_{\text{MathService}} = 1.8\,\text{msec}$$

The SOA function blocks of the services are implemented in a PLC task with 10 msec cycle time. The `EchoService` and the `MathService` require one cycle only. The number of cycles required by the `PowService` depends on the `cnt` argument.

$$\text{WC}_{\text{EchoService}} = \text{WC}_{\text{MathService}} = 2 \cdot 10\,\text{msec} = 20\,\text{msec}$$
$$\text{WC}_{\text{PowService}} = (1 + cnt) \cdot 10\,\text{msec}$$

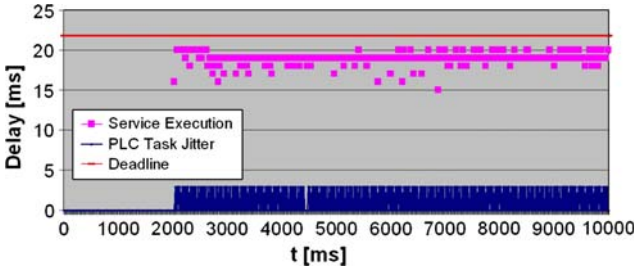


Fig. 8 Test run of the `EchoService` on the SOAP4PLC engine

This results in the following deadlines:

$$D_{\text{EchoService}} = \text{WC}_{\text{EchoService}} + L_{\text{EchoService}} = 21.8\,\text{msec}$$
$$D_{\text{MathService}} = \text{WC}_{\text{MathService}} + L_{\text{MathService}} = 21.8\,\text{msec}$$
$$D_{\text{PowService}} = \text{WC}_{\text{PowService}} + L_{\text{PowService}}$$
$$= 1.8\,\text{msec} + (1 + cnt) \cdot 10\,\text{msec}$$

8.2.4 Test run

By means of a test run, it is demonstrated that the calculated deadlines are met by the SOAP4PLC engine. Therefore, monitoring code was added at important locations [called test points (TP) in the following] of the SOAP4PLC engine. The first TP records the time stamp of incoming SOAP requests. The second TP records the time stamp of the corresponding SOAP responses. The delay between these two time stamps represents the time needed by the service. A third TP records the time stamp when the PLC task was triggered. This allows to measure the PLC task jitter while service processing. Since the PLC task is configured with 10 msec cycle time, the task has to be triggered once within every 10 msec interval. Figure 8 shows the measured values of the first 10,000 msec of the `EchoService` test. The service consumer starts to send SOAP requests after 2,000 msec from test beginning. The lower graph shows the PLC task jitter. When the service consumer starts consuming, the worst-case jitter is 3 msec, which does not violate the 10 msec task cycle. The scatterplot above represents the delay of processed services. The deadline above the service delay shows that all requests are processed within the calculated deadline.

8.3 SOAP4IPC

The engine was evaluated using aicas' JamaicaVM as the real-time JVM on QNX Neutrino v6.3.2 (http://www.qnx.com/products/neutrino_rtos/) as real-time operating system. The target system was a regular desktop PC with an AMD Athlon XP processor with 1,150MHz actual clock speed, 512MB main memory, and a 3COM 3C905B network adapter.

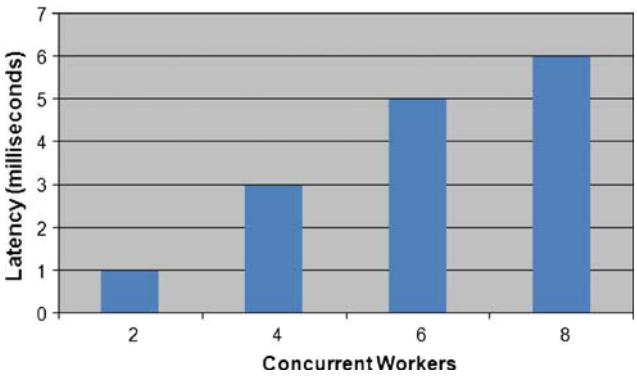


Fig. 9 Results of profiling II: concurrency level and latency

QNX Neutrino is a micro-kernel operating system that offers preemptive thread-based scheduling and mechanisms for priority inversion avoidance to permit hard real-time processing.

8.3.1 Profiling I

We evaluated 2, 4, 6, and 8 concurrent workers to process incoming messages. In general, the number of concurrent workers is a trade-off between worst-case execution time and throughput of SOAP messages. More concurrent workers result in a higher throughput, but also in a higher worst-case execution time, whereas less concurrent workers result in a lower throughput, but also in a lower worst-case execution time.

The maximum latency for each level of concurrency is shown in Fig. 9. A concurrency level of two workers merely introduces an additional overhead of 1 msec. The graph outlines that the latency increases almost linearly with the number of concurrent workers.

8.3.2 Profiling II

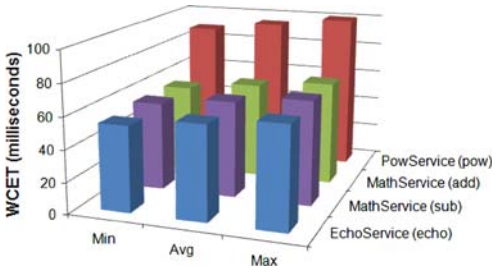
The second profiling step determines the worst-case execution time for each service, more precisely, for each operation with specific input parameters of each service. In Fig. 10, the minimum, maximum, and average execution time, as well as the standard deviation of the example services for each operation in milliseconds are shown. The standard deviation, minimum, and maximum values show that the execution time of each operation does not vary much, i.e. the average execution time is very stable, and the engine works nearly deterministically.

8.3.3 Deadline calculation

The results of both profiling steps are written to a file named `profiling.dat`. For each operation of each service, profiling.dat contains a line of the structure:

Fig. 10 Results of profiling II: minimum, maximum, and average execution time (in milliseconds) for each operation

Op	Min	Max	Avg	Dev
echo	53.991	63.991	59.364	2.873
pow	88.986	99.985	94.666	2.907
add	55.991	65.990	61.362	2.860
sub	55.991	65.990	61.157	2.900



Listing 10 Example for profiling.dat.

```
engine-overhead=1
EchoService.echo=64
PowService.pow=100
MathService.add=66
MathService.sub=66
```

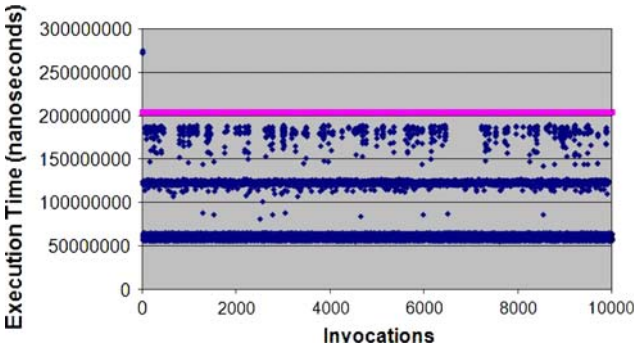


Fig. 11 Execution times during test run

```
service.operation=profiling-max
```

Accordingly, we select the maximum profiled execution time for deadline calculation. Listing 10 shows `profiling.dat` for the example services. Note that it contains one additional line that describes the overhead per worker thread introduced by the engine. The entries in `profiling.dat` are used to calculate the deadline for each service/operation depending on the number of concurrent worker threads.

8.3.4 Test run

By means of the `EchoService`, it will be shown that the calculated deadlines are met. Therefore, three concurrent consumers invoke the `echo` operation 3,333 times with a 10-character string. Figure 11 shows the execution time for each invocation and the calculated deadline. The engine is parameterized to use three concurrent worker threads to process incoming messages. The resulting scatterplots depend on the number of concurrently processed invocations, i.e. the lower scatterplot represents one, the middle scatterplot represents two, and the upper scatterplot represents three concurrently processed invocations.

Once the engine is shut-down, the statistical information shown in Listing 11 is generated. Three invocations exceeded their deadline (0.03% of all invocations). The deadline violations are caused by class loading, if a service/operation is invoked for the first time. To avoid these deadline violations, the engine may use "dummy" invocations at start-up time to trigger class loading.

8.3.5 Deadline violation

To provoke deadline violations, the `echo` operation was invoked 100 times, each time with a string with increasing length. As soon as the actual string length exceeds the profiled string length, the `echo` operation violates its deadline profiled before. The engine recognizes these deadline violations and dumps the corresponding `Task` objects to the console (cp. Listing 12).

9 Related work

There are three major research projects with similar goals as the TiCS framework: Service Infrastructure for Real-time Embedded Networked Applications (SIRENA), Industrial Machinery Normalization Process (IMNP), and Service-Oriented Cross-Layer Infrastructure for Distributed Smart Embedded Devices (SOCRADES). The characteristics of these three projects are discussed within this section.

9.1 SIRENA

The *Service Infrastructure for Real-time Embedded Networked Applications* (*SIRENA*) project [5,6,22]—a part of the Information Technology for European Advancement (ITEA) research program—is aimed at the development of a framework for the integration of heterogeneous, resource-constrained embedded devices from the industrial and home

Listing 11 Statistical information of the test run.

```
Total processed tasks:      9999
  Invocation tasks:         9999 (100.00%) - 3 (0.03%) exceeded deadline!
  WSDL tasks:               0 (0.00%)
  Fault tasks:              0 (0.00%)

  Average processing time: 106.206 msec (106205852 nanosec)
  Standard deviation:       36.068 msec (36067708 nanosec)
  Min. processing time:     54.991 msec (54991000 nanosec)
  Max. processing time:     274.958 msec (274958000 nanosec)
```

Listing 12 Excerpt of the deadline violation output.

```
[...]
Task@96b8be0: 63.990 msec (63990000 nanosec)
Task@969bce0: Deadline exceeded!
Task@969bce0: 70.990 msec (70990000 nanosec)
Task@967e980: 63.991 msec (63991000 nanosec)
Task@96610a0: Deadline exceeded!
Task@96610a0: 71.989 msec (71989000 nanosec)
[...]
```

automation, automotive, and telecommunication domains. SIRENA's integration efforts are based on two key assumptions: (1) integration is based on service-orientation, more precisely web services and (2) embedded devices offer enough computing power to process web services.

The SIRENA framework consists of the SIRENA Basic Framework, the SIRENA Framework Enhancements, and the SIRENA Framework Extension Interface. The SIRENA Basic Framework uses the Devices Profile for Web Services (DPWS) [23,24] to integrate embedded devices. DPWS is a combination of several web service specifications—a so-called *web service profile*—that meets integration demands. More precisely, DPWS permits secure message exchange among web services, description and dynamic discovery of web services, and publish/subscribe based communication. The SIRENA DPWS implementation is based on gSOAP [25] and is implemented in C. The SIRENA Framework Enhancements are a set of tools to ease the development, deployment, integration, and maintenance of devices within a SIRENA-based network. The SIRENA Framework Extension Interface describes the requirements for non-SIRENA-enabled devices to be integrated in the SIRENA framework.

SIRENA distinguishes *controlling devices* and *controlled devices* and six interaction patterns between these devices:

- **addressing:** Each controlling and controlled device is assigned a unique address to enable communication (IPv4 or IPv6 addresses are used).
- **discovery:** A controlled device that enters a SIRENA network advertises its services, whereas a controlling device searches for services if it enters a SIRENA network.
- **description:** A controlling device requires detailed information about the properties of a controlled device, e.g. offered services, manufacturer, version, or serial number. This metadata is queried by the controlling device, the controlled device answers with a description containing its metadata.
- **control:** The controlling device sends a control message to the controlled device to trigger a service. The controlled device may answer with a response message.
- **eventing:** Eventing permits asynchronous communication between controlling devices and controlled devices. A controlled device offers events that correspond to its internal state. A controlling device subscribes to a specific event. Once a new event is published by a controlled device, each controlling device that has subscribed to this event receives a notification.
- **presentation:** Controlled devices offer a presentation interface, e.g. for maintenance purposes, to permit a status request by controlling devices.

The SIRENA project has run out in September, 2005. As a follow-up project, the *Service-Oriented Device and Delivery Architecture* (*SODA*) [26] continued research and development in the device integration area.

At first glance, SIRENA shows several similarities to the TiCS framework: embedded devices that can also be PLCs

within industrial automation are interconnected using web services; the SIRENA Framework Enhancements offer tools to integrate devices in SIRENA-based networks. It is comparable to the TiCS tool support layer. Web services are processed on the embedded devices (e.g. PLCs) directly; this is also possible with the TiCS SOAP4PLC engine. However, the concrete realization of both frameworks differs heavily. SIRENA only uses a restricted set of web service protocols (DPWS), whereas the TiCS framework permits to use the entire web service protocol stack. Even though SIRENA targets the industrial automation domain, it neither offers functionality to handle time constraints (especially real-time constraints) nor functionality to compose several web services to a workflow. Finally, SIRENA only outlines a conceptual blueprint for integration of embedded devices. No prototypical implementation other than the DPWS stack exists.

9.2 IMNP

Gilart-Iglesias et al. have introduced the *Industrial Machinery Normalization Process* (*IMNP*) [3,4,27,28]. It defines a service model for industrial machinery with the primary objective to raise abstraction. This normalization process is divided into three steps:

- **physical normalization:** The physical normalization step equips an industrial device with communication and computation functionalities via the use of specialized embedded devices.
- **middleware normalization:** Within the middleware normalization step, a minimal service container is implemented. It can be used to deploy and invoke services at the industrial machinery. For this purpose, the embedded device implements a complete web service protocol stack to enable the deployment and invocation of web services.
- **services normalization:** The services normalization step defines all services necessary to expose the industrial machinery's functionality. Gilart-Iglesias et al. distinguish between production, management, and utility services. A production service exposes the core functionality of the industrial device, a management service permits monitoring of the device, whereas the utility services are internally used to access actuators and sensors.

After successful realization of IMNP, an arbitrary industrial device can be accessed directly by the business layer using services. Consequently, IPCs and PLCs are not longer required and abolished by using IMNP.

Despite the fact that IMNP and TiCS are fueled by the same vision—arbitrary industrial devices are seamlessly integrated using web services—the focus of both projects differs. TiCS uses an evolutionary approach that integrates manufacturing devices by extending IPCs and PLCs with a web service interface. Consequently, TiCS promises increased protection of investment and acceptance by automation engineers compared to IMNP. Unfortunately, IMNP completely ignores the time constraints of production processes. Although Gilart-Iglesias et al. stipulate provision for real-time constraints during the normalization process, they present no solution to meet these requirements.

9.3 SOCRADES

The European Union funded *Service-Oriented Cross-Layer Infrastructure for Distributed Smart Embedded Devices* (*SOCRADES*) project [24,29–32] is based on the principle of collaborative automation and targets three main objectives:

- definition of an architecture for a web service based communication infrastructure within industrial enterprises
- description of web services with agent-interpretable semantic markup to ease their composition
- investigation of existing and development of new wireless communication protocols for the interconnection of embedded devices

SOCRADES divides the industrial enterprise in a device layer, composition layer, middleware layer, and an application layer. The device layer contains web service enabled embedded devices. Since SOCRADES exploits the results of SIRENA, these devices use DPWS to expose web services to the higher layers. The composition layer combines several embedded devices to offer value-added functionality. This functionality is also offered as web services using DPWS. The middleware layer realizes the integration of the device and composition layer with the application layer. The application layer corresponds to the business layer in a traditional industrial enterprise.

The layers of SOCRADES show similarities to the TiCS layers: the SOCRADES device layer resembles the TiCS manufacturing layer, whereas the composition and middleware layer of SOCRADES offer similar functionality as the TiCS real-time infrastructural and real-time service layer. An analogon to the SOCRADES application layer does not exist within the TiCS framework. However, SOCRADES completely disregards a key requirement of industrial automation: real-time processing. Up to now, SOCRADES completely lacks a prototypical implementation, proof of concept, and evaluation.

9.4 Summary

The comparison of TiCS with SIRENA, IMNP, or SOCRADES shows several similarities, but also fundamental differences. TiCS' main focus is to describe and keep deadlines

of web services within the production process. For this purpose, TiCS takes an evolutionary approach that extends IPCs and PLCs with a web service interface, offers technologies to keep time constraints, and empowers automation engineers by several tools to describe and model time-constrained web services and workflows. SIRENA is an early approach to use service-orientation for integration purposes within industrial automation. Consequently, SIRENA does not have a technical realization. The follow-up project SOCRADES offers first prototypical implementations with a focus on integration. IMNP and SIRENA also have a visionary character, but no technical realization is provided. TiCS is based on the same vision as SIRENA, IMNP, and SOCRADES, but has a higher level of implementation maturity.

10 Conclusions

This article has presented the TiCS framework, a development and execution environment for web services with real-time constraints. It is tailored to the specific needs of industrial automation. TiCS is the first technical foundation for the third generation of industrial automation that focuses on the use of open, standardized protocols and a flexible integration of manufacturing devices.

The TiCS framework consists of four functional layers—hardware, real-time infrastructural, real-time service, and tool support layer—that contain several functional components for the development, description, deployment, composition, and execution of time-constrained web services and workflows. Four selected components have been presented in this article: SOAP4PLC, SOAP4IPC, WS-TemporalPolicy, and TiCS Modeler.

SOAP4PLC permits the execution of web services on programmable logic controllers, whereas *SOAP4IPC* permits the execution of web services on industrial PCs. Both engines provide web service execution in real-time and ease-of-use for automation engineers. *WS-TemporalPolicy* is a policy language suitable for the description of the dynamic timing behavior of time-constrained web services and workflows. The *TiCS Modeler* is a graphical BPEL4WS workflow editor that allows the composition of several time-constrained web services into a multi-step, value-added workflow.

A main area for future work is the integration of security features within the TiCS framework. In general, the TiCS framework permits a web service based access to the manufacturing layer of an industrial enterprise. Since the manufacturing process is mission critical for industrial enterprises, i.e. production downtimes are not acceptable, it has to be protected against malicious damage and operating errors. Examples for malicious damage are virus and worm infections of the IT infrastructure or hacker attacks. Operating errors are, for example, a wrongly composed workflow or the invocation of a web service with incorrect input parameters. Consequently, future releases of the TiCS framework must support security with respect to the infrastructure and the user. Infrastructural security may be realized using a specific network topology, e.g. separated business and manufacturing networks coupled via a security gateway, and specific network protocols, e.g. HTTPS instead of HTTP as the transfer protocol. To avoid operating errors, a formal description of the correct system behavior is important. This allows to automatically check and possibly correct misconfigured web services and workflows.

References

1. Stankovic J (1988) Misconceptions about real-time computing: a serious problem for next-generation systems. Computer 21:10–19
2. Murugesan R (2006) Evolution of industrial automation. Int J Comput Appl Technol 25:169–174
3. Gilart-Iglesias V, Macia-Perez F, Marcos-Jorquera D, Mora-Gimeno F (2007) Industrial machines as a service: modelling industrial machinery processes. In: Proceedings of the 5th IEEE international conference on industrial informatics (INDIN). IEEE Computer Society Press, New York, pp 737–742
4. Gilart-Iglesias V, Macia-Perez F, Mora-Gimeno F, Berna-Martinez J (2006) Normalization of industrial machinery with embedded devices and SOA. In: Proceedings of the IEEE conference on emerging technologies and factory automation (ETFA). IEEE Computer Society Press, New York, pp 173–180
5. Jammes F, Smit H (2005) Service-oriented architectures for devices—the SIRENA view. In: Proceedings of the 3rd international IEEE conference on industrial informatics (INDIN). IEEE Computer Society Press, New York, pp 140–147
6. Jammes F, Smit H (2005) Service-oriented paradigms in industrial automation. IEEE Trans Ind Inform 1(1):62–69
7. Shen W, Norrie D (2001) Dynamic manufacturing scheduling using both functional and resource related agents. Integr Comput Aided Eng 8:17–30
8. Heinzl S, Mathes M, Friese T, Smith M, Freisleben B (2006) Flex-SwA: flexible exchange of binary data based on SOAP messages with attachments. In: Proceedings of the IEEE international conference on web services (ICWS). IEEE Computer Society Press, New York, pp 3–10
9. Mathes M, Heinzl S, Friese T, Freisleben B (2006) Enabling Post-invocation parameter transmission in service-oriented environments. In: Proceedings of the international conference on networking and services (ICNS). IEEE Computer Society Press, New York, pp 55–60
10. Saez G, Sliva A, Blake M (2004) Web services based data management: evaluating the performance of UDDI registries. In: Proceedings of the IEEE international conference on web services (ICWS). IEEE Computer Society Press, New York, pp 830–831
11. Mathes M, Heinzl S, Freisleben B (2008) Towards a time-constrained web service infrastructure for industrial automation. In: Proceedings of the 13th IEEE international conference on emerging technologies and factory automation (ETFA). IEEE Computer Society Press, New York, pp 846–853
12. Andrews T, Curbera F, Dholakia H, Goland Y, Klein J, Leymann F, Liu K, Roller D, Smith D, Thatte S, Trickovic I, Weerawarana S (2003) Business process execution language for web services—version 1.1. http://www.ibm.com/developerworks/library/specification/ws-bpel/

13. Mathes M, Schwarzkopf R, Dörnemann T, Heinzl S, Freisleben B (2008) Orchestration of time-constrained BPEL4WS workflows. In: Proceedings of the 13th IEEE international conference on emerging technologies and factory automation (ETFA). IEEE Computer Society Press, New York, pp 1–4
14. Mathes M, Heinzl S, Freisleben B (2008) WS-TemporalPolicy: a WS-Policy extension for describing service properties with time constraints. In: Proceedings of the 1st IEEE international workshop on real-time service-oriented architecture and applications (RTSOAA) of the 32nd annual IEEE international computer software and applications conference (COMPSAC), pp 1180–1186
15. Mathes M, Stoidner C, Heinzl S, Freisleben B (2009) SOAP4PLC: web services for programmable logic controllers. In: 17th Euromicro international conference on parallel, distributed, and network-based processing (Euromicro PDP). Springer, Berlin, pp 210–219
16. International Electrotechnical Commission (IEC) (2003) Programmable controllers—Part 3: programming languages (IEC 61131-3). http://www.iec.ch/
17. Mathes M, Gärtner J, Dohmann H, Freisleben B (2009) SOAP4IPC: a real-time SOAP engine for industrial automation. In: 17th Euromicro international conference on parallel, distributed, and network-based processing (Euromicro PDP). Springer, Berlin, pp 220–226
18. W3C (2007) Web services policy framework 1.5. http://www.w3.org/TR/ws-policy/
19. W3C (2004) XML schema part 1: structures, 2nd edn. http://www.w3.org/TR/xmlschema-1/
20. Dörnemann T, Mathes M, Schwarzkopf R, Juhnke E, Freisleben B (2009) DAVO: a domain-adaptable, visual BPEL4WS orchestrator. In: Proceedings of the IEEE 23rd international conference on advanced information networking and applications (AINA). IEEE Computer Society Press, New York, pp 121–128
21. Siebert F (1999) Hard real-time garbage collection in the Jamaica virtual machine. In: Proceedings of the 6th international conference on real-time computing systems and applications (RTCSA). IEEE Computer Society Press, New York, pp 96–102
22. Bohn H, Bobek A, Golatowski F (2006) SIRENA—service infrastructure for real-time embedded networked devices: a service oriented framework for different domains. In: Proceedings of the international conference on networking, international conference on systems, and international conference on mobile communications and learning technologies (ICN/ICONS/MCL). IEEE Computer Society Press, New York, pp 43–47
23. Microsoft Corporation (2006) Devices profile for web service specification. http://specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf
24. Jammes F, Mensch A, Smit H (2007) Service-oriented device communications using the devices profile for web services. In: Proceedings of the 21st international conference on advanced information networking and applications (AINA). IEEE Computer Society Press, New York, pp 947–955
25. van Engelen R, Gallivan K (2002) The gSOAP toolkit for web services and peer-to-peer computing networks. In: Proceedings of the 2nd IEEE/ACM international symposium on cluster computing and the grid (CCGRID). IEEE Computer Society Press, New York, pp 128–135
26. de Deugd S, Carroll R, Kelly K, Millett B, Ricker J (2006) SODA: service-oriented device architecture. Pervasive Comput 5:94–96
27. Gilart-Iglesias V, Macia-Perez F, Capella-D'alton A, Gil-Martinez-Abarca J (2006) Industrial machines as a service: a model based on embedded devices and web services. In: Proceedings of the 4th IEEE international conference on industrial informatics (INDIN). IEEE Computer Society Press, New York, pp 630–635
28. Vicente Berna-Martinez J, Macia-Perez F, Ramos-Morillo H, Gilart-Iglesias V (2006) Distributed robotic architecture based on smart services. In: Proceedings of the 4th IEEE international conference on industrial informatics (INDIN). IEEE Computer Society Press, New York, pp 480–485
29. Karnouskos S, Baecker O, de Souza L, Spiess P (2007) Integration of SOA-ready networked embedded devices in enterprise systems via a cross-layered web service infrastructure. In: Proceedings of the 12th IEEE international conference on emerging technologies and factory automation (ETFA). IEEE Computer Society Press, New York, pp 293–300
30. Karnouskos S, Colombo A, Jammes F, Strand M (2007) Towards service-oriented smart items in industrial environments. Microsyst Technol (MST) 2:11–12
31. Karnouskos S, Tariq M (2008) An agent-based simulation of SOA-ready devices. In: Proceedings of the 10th international conference on computer modeling and simulation (UKSIM). IEEE Computer Society Press, New York, pp 330–335
32. Sa de Souza L, Spiess P, Guinard D, Köhler M, Karnouskos S, Savio D (2008) SOCRADES: a web service based shop floor integration infrastructure. In: Proceedings of the 1st international conference internet of things (IoT). Springer, Berlin, pp 50–67