Runtime Verification of Web Services for Interconnected Medical Devices

Normann Decker* Franziska Kühn† Daniel Thoma*

*Institute for Software Engineering and Programming Lanugages
University of Lübeck, Germany
{decker, thoma}@isp.uni-luebeck.de

†Graduate School for Computing in Medicine and Life Science
Institute of Telematics
University of Lübeck, Germany
kuehn@itm.uni-luebeck.de

***Abstract*—This paper presents a framework to ensure the correctness of service-oriented architectures based on runtime verification techniques. Traditionally, the reliability of safety critical systems is ensured by testing the complete system including all subsystems. When those systems are designed as service-oriented architectures, and independently developed subsystems are composed to new systems at runtime, this approach is no longer viable. Instead, the presented framework uses runtime monitors synthesised from high-level specifications to ensure safety constraints. The framework has been designed for the interconnection of medical devices in the operating room. As a case study, the framework is applied to the interconnection of an ultrasound dissector and a microscope. Benchmarks show that the monitoring overhead is negligible in this setting.**

I. Introduction

Nowadays, the reliability of safety critical systems usually is ensured by applying techniques like testing or model checking to the complete system. When systems are built on-the-fly by interconnecting components that have been developed independently, the complete system is not available for analysis in advance. Applying the established techniques when interconnecting components would usually be too expensive and time consuming. Especially when the devices are interconnected in a plug-and-play fashion, there is only a very limited time frame available. On the other hand, testing components independently is not sufficient as it is not possible to derive any guarantees for the complete system from test results of single components. Ensuring correctness under these conditions is, however, increasingly important, as it is more and more common to build even safety critical systems by dynamically interconnecting independently developed components.

For many application domains, the predominant paradigm to organise such systems is that of *service-oriented architectures (SOA)*. An important technology for realising SOAs is that of *web services*. The major advantages of web services are that they are independent of a certain technology and build on protocols and data formats (e. g., HTTP and XML) which are widely available.

In the medical domain, interconnecting devices, possibly from different manufacturers, is a topic of growing importance and is addressed by several current research projects like OR.NET[1] and MD PnP[2]. For technical as well as legal reasons, however, interconnecting medical devices dynamically is in most cases still not possible in practice. Usually the legislation requires that devices used together have been tested and certified in that particular combination. Testing and certifying all possible combinations of devices in advance, though, is not a viable approach. Also, if the clinic operator would interconnect devices without respecting their intended use he would have to test the combination himself and take responsibility for any failures induced by the interconnection. Commonly, these tasks could not be handled by the clinic operator, especially, as to him a device comes as a black box.

Instead, the risks introduced by the interconnection have to be addressed differently. In [1] it has been discussed how the level of safety can be maintained. The basic idea is to certify the devices with respect to their interface specification. An interface specification contains the static definition including, e. g., available operations, their parameters and permitted value ranges as well as behavioural correctness properties. When connecting devices, it then has to be ensured, that the defined interfaces are compatible. As there is usually no complete guarantee that the devices adhere to their interface specification, the interfaces have to be monitored at runtime. Furthermore, not all behavioural constraints can be checked by observing interfaces independently. Thus, the communication between the devices has to be monitored as well to ensure correct cooperative behaviour.

In this paper, we present a framework for specifying correctness properties of the communication of web services and checking them at runtime. The framework has been designed for the interconnection of medical devices in operating rooms. To reduce the risk of errors introduced when defining properties, we apply *runtime verification* techniques to synthesise efficient *monitors* from a high-level language. Monitors are software components that observe the execution of a system and continuously assess its compliance with a specification.

We evaluate our approach in the context of the following scenario from the medical domain. An ultrasound dissector (USD) and a microscope (MS) are interconnected such that triggering the USD as well as viewing and modifying its parameters can be done via the user interfaces of the MS. Both devices have to be used by a surgeon in combination, e. g., during brain surgery. Being able to use only one user interface for the devices would simplify handling both at the same time and provide better overview. This scenario is a typical example of how medical devices are likely to be interconnected

[1] www.ornet.org [2] www.mdpnp.org

in the nearer future. The interconnection of both devices has been realised prototypically in the DOOP[3] project in a collaboration between the Möller-Wedel GmbH & Co KG and the Söring GmbH and is one of the applications we investigate further within OR.NET. Web services have been used for implementing the prototype. SOA and web services are the technology of choice in OR.NET and other projects concerned with the dynamic interconnection of medical devices. The USD is a device in one of the highest safety classes (Class IIb according to [2]) as malfunctions could directly cause bodily harm. Thus, ensuring its safety is of critical importance. Furthermore, the required correctness properties pose several interesting challenges as data, timing and distributed behaviour is involved. Consider, e. g., the property that messages have to be processed in the same order as they have been sent, i.e. the messages have to be received with an increasing sequence number. This property has to hold in order to avoid overwriting parameters with old values. While this is still a rather simple property, it already involves counting, and thus cannot be handled by formalisms limited to regular properties that are often used for runtime verification.

Therefore, we use an extension of linear-time temporal logic (LTL) [3] that allows us to use first-order formulae referring to runtime data inside LTL operators. This extension facilitates expressing complex constraints as counting, temporal order or recursive data types. This logic as well as a corresponding procedure for synthesising monitors has been introduced in [4]. In our formalism, we can express the example from above as

$$\mathrm{G}(i = \mathsf{sendNum} \Rightarrow \mathrm{X}\,\mathsf{sendNum} > i).$$

The symbol $\mathsf{sendNum}$ refers to the sequential number of the current message. It is a constant provided by the current message and may hence change over time. The free variable i is implicitly universally ($\forall$) quantified over a fixed, possibly infinite domain. Intuitively, the semantics of such a formula can be derived by instantiating the formula for all (infinitely many) values for i. Then, both (in-)equations can be evaluated for the respective messages. The domain of the free variables as well as the interpretation of the symbols $=$ and $>$, is provided by a fixed *first-order theory*. Here, G and X are the LTL-operators for *always* and *next*, respectively. For our application, all relevant constraints can be expressed using the theories of *IDs with equality* and of *linear in-equations over integers*. It is, however, possible to use others.

Our framework builds on this expressive formalism and corresponding monitoring procedures. It can observe and intercept messages and allows the user to integrate monitors into the target system. While we focus on medical applications here, our approach is suitable for many other SOAs and our implementation can be used for many web service applications based on the Java API for XML Web Services.

Following the spirit of SOAs, our framework provides the concept of a *monitoring service*. A monitoring service executes the monitors synthesised from the specification and sends results to a custom handler service as well as to the caller. Handler services are only required to implement a certain interface and may react on failures reported by the monitoring service. The system is observed by transparently attaching so called interceptors to the web service stacks used by the different components. Interceptors dispatch copies of transmitted messages to monitoring services and may process their output. In particular, interceptors may block messages when the monitoring service reports a violation, thereby preventing the violating message from being delivered. Monitoring services, handler services and interceptors may be deployed throughout the system as needed. For our application we instantiate an interceptor on each medical device, one local and one global monitoring service and a single handler service reacting to system failures.

We generated benchmarks for our framework using the configuration required for our application and stub-implementations of the application's services. Our benchmarks show that the monitoring overhead is negligible in this setting.

Related Work: Adding runtime verification to systems based on web services has been studied before. In [5] an architecture for runtime monitoring of web services is described. The authors also use features of the web service stack to intercept messages and dispatch them to monitors. Their approach is less flexible though, as it assumes that all messages are dispatched to a single monitoring system and they do not describe how the system may react to monitoring output. Their interceptors do not provide any additional functionality such as synchronisation or rejecting messages. Furthermore, they do not describe any specification formalism but rely on monitors provided manually.

In [6], message sequence charts (MSC) are used to describe interface constraints of web services. The authors provide a monitoring synthesis procedure based on a formal semantics for MSCs. However, their approach does neither support handling data nor distributed constraints over multiple services. It also does not support rejecting violating messages.

A lot of work focuses on monitoring business processes described in the Business Process Execution Language (BPEL). BPEL comes with a specific architecture: multiple instances of a BPEL process are run and managed by a web service. Hence, it is natural to specify properties per instance and simply run the monitors in parallel inside the same web service. It is not necessary to handle any global or distributed behaviour as opposed to our setting. Such an approach is introduced in [7]. The interception of events is specific to BPEL processes and the monitors have to be given as automata that have to be specified manually. Furthermore, the authors do not give a formal semantics and do not support blocking messages. In [8], the past-time fragment of LTL is used to synthesise monitors. Again, the approach is strictly limited to BPEL. It only allows for monitoring a single service and does not allow for blocking messages. In [9], an approach is introduced that can use compensation actions to recover from a system failure. The approach is also very specific to BPEL. Furthermore, our application requires to actually prevent a dangerous action from being executed.

In [10], an approach to ensuring the safety of the dynamic interconnection of medical devices has been described, but they aim for verification of medical devices instead of using runtime monitors.

Another approach using a variant of LTL extended by

[3] www.doop-projekt.de

first-order formulae is described in [11]. The expressiveness of that logic is quite limited as it only allows to compare for equalities and only comprises quantifiers restricted to the current observation. Important aspects as counting or temporal ordering are not expressible. Furthermore, only local monitors are supported.

Our framework also supports the concept of blocking violating messages which can be seen as an instance of runtime enforcement [12].

Outline: In the following Section II, we give an overview over the specification formalism and runtime verification techniques employed by our framework. Based on this, Section III describes our approach to monitoring web services in general. In Section IV we discuss our medical scenario and its safety requirements and show how the monitoring framework is instantiated for this particular application. Finally, we present our implementation and benchmarks in Section V.

II. Runtime Verification with Data

Runtime verification can be seen as both, a lightweight verification technique as well as an architectural safety concept. The aim is to provide high-level behavioural specification formalisms in combination with algorithmic procedures to synthesize and integrate program code that monitors the execution of the program under observation and continuously verifies that it conforms to the specification.

While runtime verification itself deals with the pure detection of runtime errors, the methodology can be integrated with recovery routines reacting to the deviation and hence providing an additional layer for behavioural exception handling. The independent specification of formal runtime properties and automatic integration of monitoring code is generally a valuable concept in software engineering, in particular for safety critical systems.

A. Behavioural Specifications

The behaviour that needs to be specified and verified in our setting is the sequence of messages sent between web services. The formal specifications we consider are therefore based on *linear-time temporal logic (LTL)*, a comprehensible formalism that was proposed for program verification in [3] and has become widely used for the specification of sequential system behavior. A collection of specification patterns is also available to help developers formalizing typical properties [13]. The syntax of LTL formulae φ is defined over a finite set AP of *atomic propositions* according to the following grammar.

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathrm{X}\,\varphi \mid \varphi\,\mathrm{U}\,\varphi \mid \mathrm{true} \quad (\text{where } p \in AP)$$

We use the common abbreviation for *always* ($\mathrm{G}\,\varphi := \neg(\top\,\mathrm{U}\,\neg\varphi)$). It is also possible to add past-time operators without increasing expressiveness. For more convenience further syntactic extensions can be used, such as SALT [14].

The standard semantics of LTL is defined on infinite words over the alphabet $\Sigma = 2^{AP}$ of sets of propositions. The set of all infinite words over Σ is denoted Σ^ω and the semantics of a formula φ is given by the mapping $[\![\varphi]\!]_\omega : \Sigma^\omega \to \mathbb{B}$ with

$$\begin{aligned}
[\![p]\!]_\omega(w) &:= \top \text{ if } p \in w_0 \text{ and } [\![p]\!]_\omega(w) := \bot \text{ otherwise},\\
[\![\neg\varphi]\!]_\omega(w) &:= \neg[\![\varphi]\!]_\omega(w),\\
[\![\varphi \wedge \psi]\!]_\omega(w) &:= [\![\varphi]\!]_\omega(w) \wedge [\![\psi]\!]_\omega(w),\\
[\![\mathrm{X}\,\varphi]\!]_\omega(w) &:= [\![\varphi]\!]_\omega(w^{(1)}) \text{ and}\\
[\![\varphi\,\mathrm{U}\,\psi]\!]_\omega(w) &:= \top \text{ if } \exists_n : [\![\psi]\!]_\omega(w^{(n)}) = \top\\
&\qquad \text{and } \forall_{0\le i<n} : [\![\varphi]\!]_\omega(w^{(i)}) = \top
\end{aligned}$$

where $w = w_0w_1\ldots \in \Sigma^\omega$ ($w_i \in \Sigma$) and $w^{(n)} := w_nw_{n+1}\ldots$ is the suffix of w starting at position n.

This infinitary semantics is suitable for modelling the behaviour of, e. g., reactive systems. During on-line monitoring, however, the actually observed sequence, e. g. of system states or messages, is a finite prefix of the whole run. It is hence necessary to evaluate a property at any time using a finitary semantics that additionally takes the possibility of only incomplete observations into account. This is reflected by the maxim of *impartiality* that requires to distinguish preliminary from final verdicts [15]. A semantics is impartial if once a final verdict is declared, it must not change under any additionally observed information. It is furthermore *anticipatory* if it declares a verdict as early as possible [15]. For the purpose of this paper we use the three-valued impartial, anticipatory LTL semantics LTL_3 [16] that is defined over the truth domain $\mathbb{B}_3 := \{\top, \bot, ?\}$ where ? is *inconclusive*. For a finite alphabet Σ and $w \in \Sigma^*$, the LTL_3 semantics of a formula φ is the mapping $[\![\varphi]\!]_3 : \Sigma^* \to \mathbb{B}_3$ with

$$[\![\varphi]\!]_3(w) := \begin{cases} \top & \text{if } \forall u \in \Sigma^\omega : [\![\varphi]\!]_\omega(wu) = \top \\ \bot & \text{if } \forall u \in \Sigma^\omega : [\![\varphi]\!]_\omega(wu) = \bot \\ ? & \text{otherwise.} \end{cases} \tag{1}$$

Essentially all LTL semantics, in particular LTL_3, “access” the model only by using atomic propositions. The semantics of the temporal operators do not depend on the particular letter that is encountered at some position but only on the evaluation of propositions. We call semantics of temporal logics that enjoy this property *propositional*. It allows us to substitute propositions by more complex expressions taking data into account.

Reasoning on Data: A very powerful formalism to reason on data and data structures is provided by first-order (FO) logic. Dedicated FO theories such as arithmetics, arrays, lists or uninterpreted functions are suitable for a large class of data structures used in modern software systems and can be handled by today's SMT solvers (cf. [17], [18], [19]).

The syntax of FO formulae is defined over some signature $\mathcal{S} = (P, F, ar)$ comprised of predicate symbols P and function symbols F of arity defined by the mapping $ar : P \cup F \to \mathbb{N}$. Additionally, formulae can contain variables from a set W and expressions that evaluate to concrete values. Formulae χ and expressions e are built according to the following grammar where $x \in W$, $p \in P$ and $f \in F$.

$$\begin{aligned}
\chi &::= p(e_1, \ldots, e_{ar(p)}) \mid \neg\chi \mid \chi \wedge \chi \mid \forall_x\chi\\
e &::= x \mid f(e_1, \ldots, e_{ar(f)})
\end{aligned}$$

We use common abbreviations such as $\exists_x\chi := \neg\forall_x\neg\chi$, and

$\chi_1 \vee \chi_2 := \neg(\neg\chi_1 \wedge \neg\chi_2)$. A variable x is called *free*, if it is not within the scope of some quantifier ($\forall$ or $\exists$).

For a signature $\mathcal{S}$, an $\mathcal{S}$*-structure* is a tuple $s = (U, \mathfrak{s})$ where U is a *universe* and $\mathfrak{s}$ is an *interpretation* that maps function symbols $f \in F$ to functions $\mathfrak{s}(f) : U^{ar(f)} \to U$ and predicate symbols $p \in P$ to relations $\mathfrak{s}(p) \subseteq U^{ar(p)}$ of their respective arity. The semantics of FO formulae over signature $\mathcal{S}$ and variables W is defined over tuples (s, θ), where s is an $\mathcal{S}$-structure and $\theta : W \to U$ is a (partial) *valuation* of variables. Expressions are evaluated as $[\![x]\!](s, \theta) := \theta(x)$ for variables $x \in W$ and for function symbols $f \in F$ as $[\![f(e_1, \dots, e_n)]\!](s, \theta) := \mathfrak{s}(f)([\![e_1]\!](s, \theta), \dots, [\![e_n]\!](s, \theta))$. Then, for $p \in P$, $x \in W$ and formulae χ, χ', we define the *models* relation $\models$ by

$$\begin{array}{ll}
(s,\theta) \models p(e_1, \dots, e_n) & \text{if } ([\![e_1]\!](s,\theta), \dots, [\![e_n]\!](s,\theta)) \in \mathfrak{s}(p), \\
(s,\theta) \models \neg\chi & \text{if } (s,\theta) \not\models \chi, \\
(s,\theta) \models \chi \wedge \chi' & \text{if } (s,\theta) \models \chi \text{ and } (s,\theta) \models \chi', \\
(s,\theta) \models \forall_x \chi & \text{if } (s,\theta[x \mapsto u]) \models \chi \text{ for all } u \in U.
\end{array}$$

For reasoning in a specific setting, the interpretation of some dedicated symbols, as well as the universe is fixed in terms of a *theory*. It is formally represented by a fixed structure over some signature $\mathcal{T}$. For example, consider the theory of natural numbers with equality. It is represented by a structure with universe $\mathbb{N}$ over a signature only defining a predicate symbol $=$ of arity 2.

Temporal Data Logic: Combining FO theories with temporal logic provides a formalism that allows for expressing a wide range of realistic properties of the communication in a distributed system. Recently, a generic monitoring approach has been proposed for such a combination of first-order and temporal logic [4]. This framework is particularly useful for monitoring the communication of web services since transmitted messages provide a discrete sequence of observations where the (XML) message structure and internal data values are of significant importance for verifying the correctness of the communication. In the remainder of this section, we summarize the approach in the light of this application.

Given a theory t over a signature $\mathcal{T}$, we extend $\mathcal{T}$ with additional symbols called *observation predicates* and *observation functions*. We then model actual observations by structures g that are equal to t but additionally provide an interpretation for the observation symbols. We use Γ to denote the set of all such observation structures which hence only differ in terms of their interpretation of observation symbols. We call FO formulae using such additional observation symbols *data formulae* since we use them to specify properties on the data that is transmitted in observed messages.

To evaluate data formulae in practice, the user is required to provide the information on how to map runtime data to the interpretations of observation symbols. That way, the behavioral specification itself is completely decoupled from the concrete system. We show later how this is done in the concrete setting of web services.

As mentioned earlier, the LTL_3 semantics is propositional. This means we can exchange propositions in LTL by more complex properties, expressed by data logic formulae, without affecting the temporal aspect. We combine the temporal- and data logic to what we call *temporal data logic (TDL)* that is interpreted over words from Γ^*.

Free variables in data formulae are considered to be universally quantified. This provides the freedom to specify dynamic dependencies between messages. Returning to the example from the introduction, we observe that it can be formulated as a TDL formula using the theory of (in)equalities over natural numbers and an observation symbol $\mathsf{sendNum}$. If observed messages provide a message number n, they can be represented as structures $m = (\mathbb{N}, \mathfrak{m})$ interpreting the observation symbol by $\mathfrak{m}(\mathsf{sendNum}) = n \in \mathbb{N}$.

Given a variable valuation $\theta : W \to U$, the semantics of a TDL formula φ is defined as a mapping $[\![\varphi]\!]^\theta : \Gamma^* \to \mathbb{B}_3$ based on an infinitary semantics $[\![\varphi]\!]^\theta_\omega$, just as LTL_3. We only replace the definition for atomic propositions by

$$[\![\chi]\!]^\theta_\omega(w) := \top \text{ if } (w_0, \theta) \models \chi \text{ and otherwise } [\![\chi]\!]^\theta_\omega(w) := \bot$$

for data logic formulae χ. Based on $[\![\varphi]\!]^\theta_\omega$, the three-valued semantics $[\![\varphi]\!]^\theta$ is defined in analogy to Equation 1. Since free variables are considered universally quantified, we let the general TDL semantics of a formula φ be defined as the conjunction (infimum) over all possible valuations $\theta \in U^V$

$$[\![\varphi]\!](w) := \prod_{\theta \in U^V} [\![\varphi]\!]^\theta(w).$$

B. Monitoring

Executable monitoring code is very specific and therefore the behavioural specification is first translated into an intermediate, abstract computational model, that we call the *monitor*. Monitor constructions are either rewriting-based (see, e. g., [20], [21]) or automata-based and exist for various temporal logics, including LTL (see, e. g.,[16], [22], [23], [24]). We apply automata-based techniques that can be seen as an additional optimization step. It may require more complex computations for synthesis but reduces runtime overhead.

For LTL_3, a monitor can be synthesized as a finite-state Moore machine [16]. Basically, this can be done by transforming an LTL formula into a Büchi automaton and performing emptiness checks to ensure an impartial and anticipatory evaluation of the property. As finite-state Moore machines can be determinized and minimized this allows for the construction of very efficient monitors. This construction can be lifted to obtain a monitor construction for TDL. Considering the data logic formulae $\chi_1,\dots,\chi_n$ that occur in some TDL formula φ as if they were atomic symbols, we obtain a plain LTL formula over the set of propositions $AP_\varphi = \{\chi_1, \dots, \chi_n\}$. To this formula, the monitor construction for LTL_3 can be applied yielding a Moore machine over the symbolic alphabet $\Sigma_\varphi = 2^{AP_\varphi}$. For the example formula $\mathrm{G}(i = \mathsf{sendNum} \Rightarrow \mathrm{X}\,\mathsf{sendNum} > i)$, we treat $i = \mathsf{sendNum}$ and $\mathsf{sendNum} > i$ as two atomic propositions χ_1 and χ_2. This yields a plain LTL formula $\mathrm{G}(\chi_1 \to \chi_2)$ over $AP_\varphi = \{\chi_1, \chi_2\}$ from which the Moore machine presented in Figure 1 is constructed.

The free variables in the TDL formula are universally quantified, meaning the property must be checked for all valuations θ. This can be achieved by instantiating the symbolic monitor for all valuations. In such an instance, the values of the free variables are thus fixed and any proposition $\chi \in AP_\varphi$ can be evaluated wrt. an observation. For example, binding the

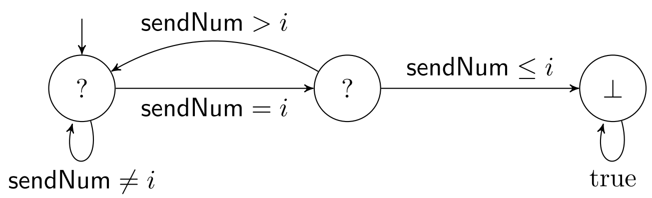

Figure 1. A symbolic Moore machine.

variable i to the value 5 in Figure 1, this instance verifies the property $\mathrm{G}(5 = \mathsf{sendNum} \Rightarrow \mathrm{X}\,\mathsf{sendNum} > 5)$ along a sequence of observations. The obtained data formulae (i. e., $5 = \mathsf{sendNum}$ and $\mathsf{sendNum} > 5$) can be evaluated independently for each observation $g \in \Gamma$ providing an interpretation, i. e., a value, for the observation symbol sendNum. The remaining reasoning (e. g., if $6 \leq 5$) only relies on the underlying theory and can be delegated to an SMT solver. While there are usually infinitely many valuations, the corresponding monitor instances can be finitely represented and executed using a suitable data structure (cf. [4] for further details).

III. Runtime Verification for Web Services

Complex applications based on web services often implement the concepts of a SOA. SOAs are based on services implementing a coherent set of functionalities. Single services can be aggregated to new services that provide more complex functionalities. The interconnection between services does not have to be static. Instead, at runtime, services might be discovered and new connections may be established. The service functionalities are usually provided over a defined interface. It remains independent of the technology or programming language used to implement a service.

Web services are usually implemented by exchanging messages serialised as XML over a network using HTTP. The most popular protocol for message-based web services is SOAP [25]. Typically, interfaces are defined using the *Web Services Description Language (WSDL)* [26] and messages are validated by the web service stack. This kind of validation, though, is limited to simple type and range checks for the methods in the statically defined interface. It does not support checking dynamic constraints and temporal correctness properties of interaction of services. To this end, our approach aims at integrating runtime verification techniques seamlessly within the setting of SOAs, particularly SOAs for interconnecting medical devices.

A common feature of web service stacks and frameworks is the ability to intercept messages in a manner that is transparent to the web service running on top of the stack. We use such a mechanism to inject interceptors for runtime verification basically copying all transmitted messages and dispatching them to some monitoring service. The messages dispatched to the monitoring services are enriched with additional information, i.e. where it has been intercepted, a local sequence number and whether it has been intercepted during sending or receiving. Figure 2 depicts all messages that, depending on the configuration, may be transmitted when *Client A* calls a function of *Service B*. Monitoring services can be deployed as needed (e.g. on the same application server or even on separate

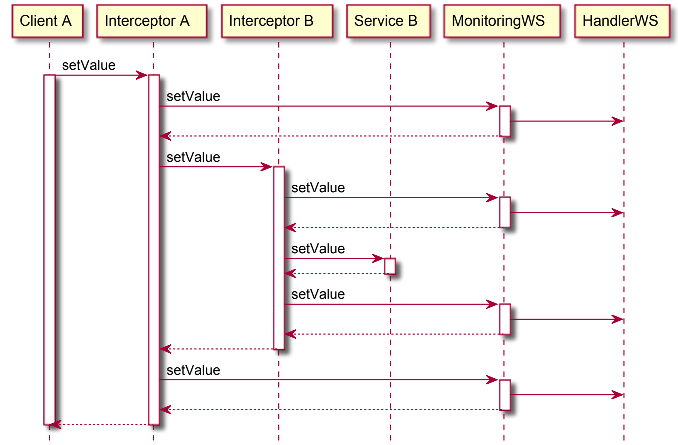

Figure 2. Dispatching of messages for monitoring

monitoring hardware) and execute possibly multiple monitors. Both, interceptors and monitoring services are configurable components, thus runtime verification can be transparently integrated into an existing system by means of configuration. The overall architecture is shown in Figure 3.

For performance reasons and to simplify temporal specifications, the messages relayed to the monitors can be filtered using the configuration of an interceptor as well as of a monitoring service. Thereby, only relevant messages are transmitted to a monitoring service, that executes a monitor only for messages relevant to the respective property. The interceptors can be configured to block messages until it receives the corresponding monitor outputs. In case a monitor reports a violation, the message will not be transmitted to its actual target and the respective action will not be executed. Here we rely on the properties of the underlying monitoring approach, impartiality and anticipation. Impartiality guarantees that a report of a violation cannot be taken back and blocking a message will not interrupt a conforming execution. Anticipation guarantees at least for the temporal aspects that the violation is reported as early as possible and thus after blocking the message the system can still be guided into a safe state. This allows for using runtime monitoring to prevent the execution of dangerous actions.

Furthermore, blocking the message until it is processed by the monitor, guarantees that the monitor observes the messages in a causally consistent order, i.e. it does not observe that a message has been received before observing that it has been sent. When the interceptor processes a message without blocking, messages from the same interceptor are still guaranteed to be received by the monitor in the correct order. Messages from different interceptors though might be received in a causally inconsistent order. Thus, when a monitor receives messages from multiple interceptors only those messages can be allowed to be processed without blocking, where the (global) order is irrelevant to the corresponding property. In case it is known in advance whether the order of certain messages is irrelevant for a property, blocking of those messages can be relaxed reducing the performance impact of monitoring. Additionally, the output of the monitors may be sent to a custom service, e. g., writing it to a log file or alert the user. It could also lead the system back into a safe state.

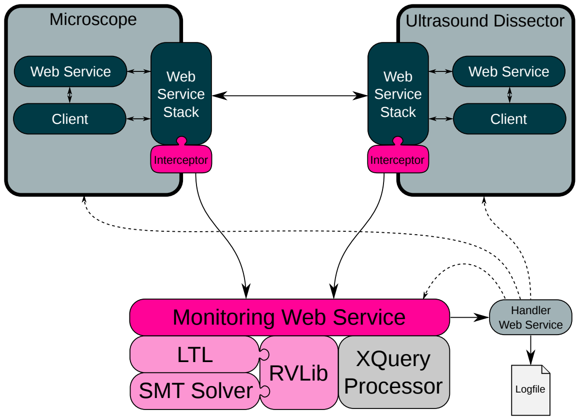

Figure 3. Architecture with central monitoring service

In principle, the monitoring service could be implemented manually but this would come with several major disadvantages. Primarily, programming at a low level of abstraction is very prone to errors. Furthermore, as the monitor would be realised at the same abstraction level as the implementation, it is likely, that similar errors would occur. Also, it is potentially very cost intensive. Instead, our approach is to synthesise monitors from a high-level specification. Many typical correctness properties can be directly expressed in such a formalism. It is less error prone than expressing the same property, e. g., in a programming language and it is easier to validate the specification. Additionally, the monitors can be generated and optimised automatically.

From a methodical perspective, another advantage is, that crucial safety constraints can already be addressed at the design or architectural level of a system. When breaking the system's functionality down to several services and designing their interfaces and interaction protocols, one also has to specify the corresponding safety constraints. These constraints can be modelled at the interface level and the corresponding monitors can be synthesised. Therefore, errors in the implementation of the services no longer affect the safety of the system as a whole.

We generate monitors using the approach described in Section II. We obtain deterministic finite Moore machines interpreted symbolically using an SMT solver. These monitors can be executed efficiently. Furthermore, they are impartial in the sense that they only return a true or false verdict when this verdict is certain, and anticipatory in the sense that they report (temporal) violations as early as possible. We require impartiality, as we reject messages causing a violation. If the monitor might change its verdict in the future, we could not determine whether a message has to be blocked. Anticipation facilitates reacting to violations in time.

IV. Runtime Verification for Interconnected Medical Devices

We realised our framework in the context of the following scenario where an MS and a USD are used during brain surgery to remove a meningeom. The MS system consists of several components including a foot and hand switch and several displays, typically only showing MS settings. The USD system consists of the actual dissector (hand piece), a control unit/generator with a display, and a foot switch.

The surgeon uses the MS to observe the operating area and, at the same time, he is holding the hand piece. When he wants to change some parameters (e.g. ultrasound power, aspiration power, irrigation volume) of the USD he has to instruct the operating personnel, as the USD's control unit is not sterile and thus the surgeon must not touch it. Even, if he only wants to reassure himself that their values are correct, he has to turn away from the MS and focus on the USD or ask the operating personnel. As triggering the USD requires the surgeon to use the respective foot switch, he has to change between both foot switches, in case he wants to control the MS via foot switch as well. Conversely, parameters of the MS (e.g. zoom, focus) cannot be changed using the USD's control unit.

Thus, it would be beneficial if the surgeon could control the USD using the MS. The parameters of the USD should be shown on displays of the MS and it should be possible to adjust them using the hand or foot switch of the MS. This would enable the surgeon to control all parameters himself and would only require him to use a single display and foot switch. This scenario has been realised by interconnecting both devices via a network. To facilitate the communication between different medical devices, a service-oriented architecture is used. Technically, the communication is realised using web services. To this end, the *Devices Profile for Web Services*, a selection of web service standards for devices with limited resources, has been adapted to the medical setting [27].

Figure 4 shows the communication between the MS and the USD on the logical level as well as the interaction with them by the surgeon and the operating personnel, respectively. When the surgeon uses, e. g., the hand switch of the MS to change the ultrasound power to 60 %, the MS sends a corresponding message to the USD. After changing the parameter, the USD notifies all registered devices, including the MS, that the value has been changed to 60 %. Now, when the operating personnel changes the ultrasound power to 80 % at the USD, the USD sends a corresponding message to itself. Again, after changing the parameter, the USD notifies all registered devices. When the surgeon triggers the USD by pressing the foot switch of the MS it sends a corresponding *on*-message. It continues to send *continue*-messages at least every 200 ms until the foot switch is released and a final *off*-message is sent. As a safety measure, the USD remains only active for 250 ms after receiving the most recent message to ensure it deactivates itself in case the connection is interrupted.

Interconnecting medical devices dynamically, introduces several additional risks. Following the approach outlined in [1] it has to be verified at runtime, that the participating devices respect the defined interfaces and correctness properties. The following properties are the most crucial for our application.

(1) The parameters that can be changed have to stay in predefined ranges (e.g. between 0 % and 100 %) and may sometimes only be changed in certain steps (e.g. in steps of 5). While ranges are usually already checked by the web service stack, this is not the case when the constraint is not part of the statically defined interface.
(2) The messages have to be received by the USD in the same order as they have been sent by the MS. Otherwise,

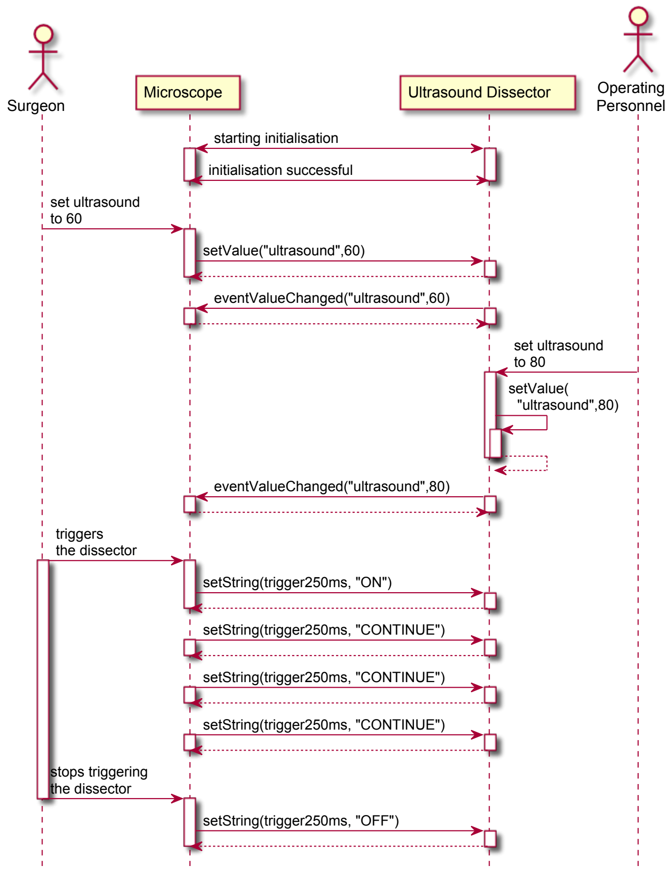

Figure 4. Interaction between USD and MS

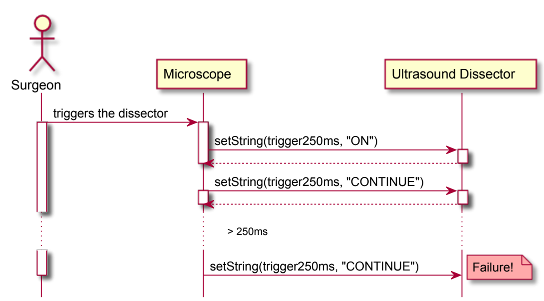

Figure 5. Violation of property (3)

e. g., when changing parameters a new value might be overwritten by an older message.

(3) To avoid the USD switching its trigger state to off while it is intended to be on, it has to be ensured that a *continue*-message is received at least every 250 ms (see Figure 5).

(4) When the USD is triggered, for every parameter the value displayed at the USD has to be the same as at the MS. Otherwise, the surgeon could assume a wrong value to be set.

In the example, both devices are exposing their interfaces as web services. The web service of the USD provides methods to set and get parameters, trigger the USD and to subscribe to parameter change notifications. The web service of the MS provides a callback method to receive such notifications. Both devices are acting as service users and call the methods of the USD web service.

Let us consider the properties described above. The first two can be expressed as

$$\begin{aligned} &\mathrm{G}(\mathsf{name} = \mathsf{volume} \\ &\qquad \Rightarrow (0 \leq \mathsf{value} \leq 100 \wedge \mathsf{value} \bmod 5 = 0)), \end{aligned} \tag{1}$$

$$\mathrm{G}(i = \mathsf{sendNum} \Rightarrow \mathrm{X}\,\mathsf{sendNum} > i). \tag{2}$$

Recall, that the interpretation of observation symbols such as name and sendNum is provided by observations, i. e., the observed messages, while the free variable i is quantified universally. Consider for property (2) a particular observation yielding, e. g., a value 5 for sendNum. For a valuation of i with $i \neq 5$ the constraint is vacuously true at this moment. The interesting valuation is the one that assigns the value 5 to i. Under this binding, the message observed *next* must provide a value for sendNum that is larger than 5. In that way, the free variables are used to express the relation of observed data values at different points in times, i. e., in different observed messages.

In a similar fashion, the third property can be specified by the formula

$$\begin{aligned} \mathrm{G}\Big(&((\mathsf{on} \vee \mathsf{cont}) \wedge t = \min(\mathsf{sendTime}, \mathsf{receiveTime})) \\ &\Rightarrow \mathrm{X}\,\mathsf{receiveTime} - 250 \leq t \Big). \end{aligned} \tag{3}$$

The non-trivial case for the free variable t is when it is bound to the minimum of the send and receive time of a message. It ensures that the delay between sending the current and receiving the next message is at most 250 ms, assuming the clocks of the sending and receiving devices are in sync. Even if they are not, the expressed guarantee is still that a message is received at least every 250 ms and the network delay does not exceed a certain limit.

The last property is more complex. We express it as

$$\begin{aligned} &\mathrm{G}\big(\mathsf{on} \Rightarrow (\\ &\qquad (\neg\mathsf{set}(c))\,\mathrm{S}\,(\mathsf{set}(c) \wedge v = \mathsf{value}) \\ &\quad \Leftrightarrow (\neg\mathsf{changed}(c))\,\mathrm{S}\,(\mathsf{changed}(c) \wedge v = \mathsf{value})\ )\big). \end{aligned} \tag{4}$$

By $\mathsf{set}(c)$ and $\mathsf{changed}(c)$ we denote that the current message is a *set* and *change value* operation, respectively, for a parameter c. By using a variable c for the parameter name, we specify the property for all parameters at once. For better readability, we use the past-time operator $\varphi\,\mathrm{S}\,\psi$ meaning that ψ did hold once and φ has been satisfied for every position since. The property can, however, also be reformulated using only the future-time operators defined above.

The only property requiring the monitor to observe messages on both devices is (4). The *set*-messages have to be intercepted at the USD and the *changed*-messages at the MS. We use a central monitoring service depicted in Figure 3 receiving messages from interceptors on both devices to monitor property (4). Properties (1) to (3) only require to observe messages on the USD. We monitor them using a dedicated monitoring service running directly on the USD. Note that by running the monitor locally on the USD we do not have any network delay for dispatching messages to the monitor. Hence, property (3) can be monitored correctly. It is also possible to run the local monitoring service on a dedicated hardware component to avoid interference with the system.

Triggering the USD when one of the properties is violated is potentially dangerous. In that case, we use the ability of the interceptor to reject messages based on the monitor output to prevent activation of the USD. The handler service is used to reset the system in case property violations occur. Disabling a system or certain functionalities is a typical measure to ensure safety of medical devices. When we reset the system we also set the monitors back to their initial states. Resetting system and monitors at the same time ensures that the monitoring verdicts remain valid.

V. Implementation and Experimental Results

We have implemented our monitoring framework for web services based on the Java API for XML Web Services (JAX-WS) [28]. JAX-WS allows for attaching interceptors at both, the client and the server side in a way that is transparent to the actual application. We use this API to attach our interceptors to web services and clients. While JAX-WS maps automatically between Java objects and their XML representation, it also provides direct access to the underlying SOAP messages. We use this facility to implement the monitoring service. The monitoring service reads its specification from an XML file, generates the corresponding monitors and executes them when an arbitrary message is received. For the synthesis of monitors we rely on *RVLib* the backend of *jUnit^{RV}* [29]. A monitor is specified using an XML based syntax for TDL formulae. For data formulae XQuery expressions are used which either result in a boolean value or an FO formula. We also use XQuery expressions wherever messages can be filtered. Our implementation uses Apache CXF[4] as an implementation of JAX-WS and Saxon[5] as an XQuery processor. Figure 3 depicts the overall architecture and the components used to implement the monitoring service. For more complex settings it could be cumbersome to set-up all required interceptors using the API. JAX-WS itself allows to configure interceptors via Java annotions or XML configuration files. Furthermore, an integration framework like Apache Camel[6] may be used.

The following listing shows the specification corresponding to Property (2) described in Section IV encoded in XML.

```
1  <G><OR>
     <NOT><P><sym><xquery><![CDATA[
3      <eq><intVar>i</intVar><int>
         {xs:integer(//*:sendSequenceNumber/text())}
5      </int></eq>
     ]]></xquery></sym></P></NOT>
7    <WX><P><sym><xquery><![CDATA[
       <lt><intVar>i</intVar><int>
9        {xs:integer(//*:sendSequenceNumber/text())}
       </int></lt>
11   ]]></xquery></sym></P></WX>
   </OR></G>
```

The XML elements G, OR, NOT encode the corresponding LTL operators, the implication has been expressed using disjunction. Further, WX represents a variant of the standard X operator that evaluates to true in case the system terminates and there is no further step. The P tag encloses the propositional part, more precisely, the data logic formulae that are evaluated wrt. the individual messages.

[4] cxf.apache.org [5] www.saxonica.com [6] camel.apache.org

In this example, the data logic formulae consisting of just a single unary observation predicate $\mathsf{sym}(i)$ represented by the XML element sym. Inside the sym tags, an XQuery expression defines the free variable i and thereby the arity of the predicate. Given an observation in terms of a SOAP message, the expression yields a first-order formula, again encoded in XML. This formula, generated from the message, precisely characterises the predicate that is the interpretation of the symbol sym under this observation. The expression in line 4 constrains the free integer variable x to be equal to the sequence number assigned at the sending location. The expression in line 9 constrains x to be less than the sequence number.

Evaluation: To validate the feasibility of our approach, we implemented a testing environment based on the JAX-WS stack simulating the scenario presented in Section IV. I.e. a typical usage scenario where two kinds of messages, setting a parameter and triggering the USD, are sent with equal average frequency. Figure 6 shows the results when sending those messages from the MS to the USD repeatedly. The number of calls is the number of actual calls between the MS and the USD, not counting return messages for synchronous calls or monitoring messages. We executed the benchmark under different parameters.

The setting closest to our application is *sync, mixed*. In this setting, the application is monitored and the monitor blocks for all messages where an error might be reported. The local monitor on the USD is accessed using a local transport transmitting messages in memory. Any other communication is transmitted via HTTP. As a single message requires only 12 ms even with monitoring, message transmission is by far fast enough for our application. In the base-line scenario *HTTP* monitoring has been deactivated completely. It shows that most of the execution time is due to overhead caused by the web service stack. Comparing the execution time of the other scenarios to this base-line scenario shows that monitoring increases the transmission delay by a factor 2.4. Medical devices are usually interconnected using small, short distance networks where transmission times are not an issue. Thus, this factor is still negligible. E.g. in our application we have a time constraint of 250 ms which is an order of magnitude above the transmission time of 12 ms.

In scenario *dummy, mixed* monitoring is enabled but the monitoring services execute no monitors. This shows, that about half of the overhead is caused by, e. g., messaging and interception and only the other half is due to actually verifying the properties. In the scenario *async, mixed*, blocking of messages is almost completely disabled resulting in almost no monitoring overhead (a factor 1.1). In this case the handler service would still receive the correct monitoring output, but in case of an error a message would not be blocked immediately. For our application this would not be acceptable, but for many applications a reaction to an error with a slight delay is sufficient.

Figure 6 also shows the times required to actually execute the monitors for Properties (1) to (4). These measurements are less accurate as they had to be measured summing up the times for each monitoring step and can thus not directly be compared to the other results. They do show, however, that most of that overhead is caused by Properties (2) and (3) for which

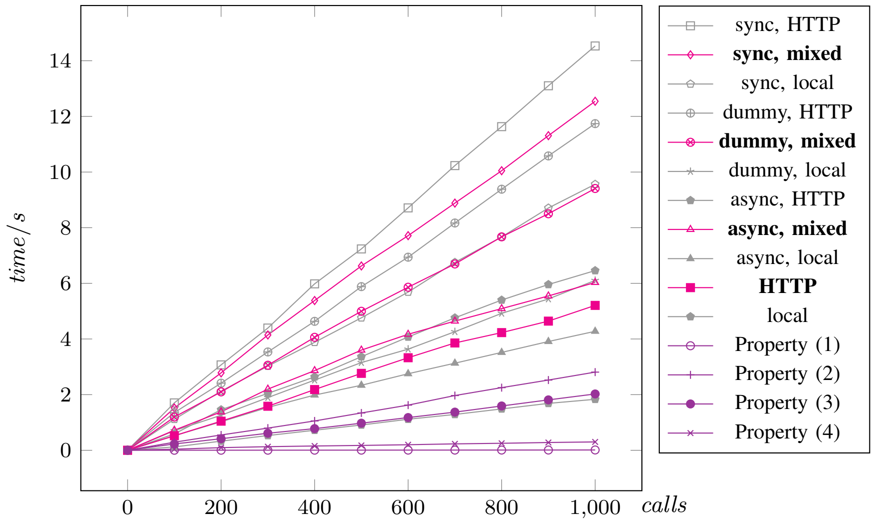

Figure 6. Experimental results, calls from MS to USD.

we actually need to call an SMT solver whereas Properties (1) and (4) only need identity which is handled directly in our implementation. The other scenarios show variations using only HTTP or only the local in-memory transport. They are not relevant to our application but still might be of interest for other applications.

While these results cannot directly be transferred to other application domains, they still give some indication. They show, that the monitoring overhead can be controlled by instantiating the framework appropriately (avoiding unnecessary messages and synchronisation). Certainly, the framework can be used for monitoring applications where message transmission times are of minor importance.

The implementation, the testing environment and the benchmarks are available for download[7].

VI. Conclusion

We presented an approach to ensure the safety of service-oriented architectures for medical devices using runtime verification techniques. The approach is also suitable for a wide range of application domains where message transmission times are of minor importance. We applied our approach to a safety-critical scenario of interconnecting medical devices that will become increasingly important in the near future. We gain several valuable insights from this case study. First, it is important that a monitoring framework does not rely on a single centralised monitoring service, but allows for the deployment of multiple services as needed. In our application, some properties require a centralized monitor receiving events form different devices as well as a local monitor receiving events directly without network delay. Second, the additional functionality to reject violating messages can be useful to guarantee the safety of the system. While blocking messages can in some cases introduce additional risks, in many applications like ours, safety can be ensured by simply preventing the execution of dangerous actions. Third, an expressive specification formalism is essential as many common properties cannot be expressed without handling data, e. g., in terms of IDs or sequential numbering.

We were able to handle all important constraints for our application, which shows that our technique works particularly well for securing the dynamic interconnection of medical devices. Furthermore, our approach is open for other specification formalisms on two levels: our procedure for synthesising monitors allows for using other temporal logics than LTL (e. g., CaRet [30] for context-free constraints) and our architecture allows to use arbitrary monitor synthesis procedures. It is possible, as our benchmarks show, to keep monitoring overhead small enough to not influence the application. It is easier to convince oneself of the safety of a system when correctness properties can already be incorporated at an architectural level. We believe that in the context of the risk management for a medical device, monitoring using our approach would be well suited as a risk control measure.

Acknowledgements: We thank André Dauenheimer (Söring GmbH) and Stefan Lembke (Möller-Wedel GmbH & Co KG) for providing the application and for insightful discussions of its technical details.

References

[1] F. Kühn and M. Leucker, "OR.NET: Safe interconnection of medical devices - (position paper)," in *FHIES*, ser. Lecture Notes in Computer Science, J. Gibbons and W. MacCaull, Eds., vol. 8315. Springer, 2013, pp. 188–198.

[2] "Richtlinie 93/42/EWG des Rates vom 14. Juni 1993 über Medizinprodukte," Jul. 1993.

[3] A. Pnueli, "The temporal logic of programs," in *FOCS*. IEEE Computer Society, 1977, pp. 46–57.

[4] N. Decker, M. Leucker, and D. Thoma, "Monitoring modulo theories," in *TACAS*, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 341–356.

[5] K. Bratanis, D. Dranidis, and A. J. H. Simons, "An extensible architecture for run-time monitoring of conversational web services," in *Proceedings of the 3rd International Workshop on Monitoring, Adaptation and Beyond*, ser. MONA '10. New York, NY, USA: ACM, 2010, pp. 9–16. [Online]. Available: http://doi.acm.org/10.1145/1929566.1929568

[7] www.isp.uni-luebeck.de/wsrv

[6] J. Simmonds, Y. Gan, M. Chechik, S. Nejati, B. O'Farrell, E. Litani, and J. Waterhouse, "Runtime monitoring of web service conversations," *IEEE T. Services Computing*, vol. 2, no. 3, pp. 223–244, 2009.

[7] S. Kallel, A. Charfi, T. Dinkelaker, M. Mezini, and M. Jmaiel, "Specifying and monitoring temporal properties in web services compositions," in *ECOWS*, R. Eshuis, P. W. P. J. Grefen, and G. A. Papadopoulos, Eds. IEEE Computer Society, 2009, pp. 148–157.

[8] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti, "Run-time monitoring of instances and classes of web service compositions," in *ICWS*. IEEE Computer Society, 2006, pp. 63–71.

[9] J. Simmonds, S. Ben-David, and M. Chechik, "Monitoring and recovery of web service applications," in *The Smart Internet*, ser. Lecture Notes in Computer Science, M. H. Chignell, J. R. Cordy, J. Ng, and Y. Yesha, Eds., vol. 6400. Springer, 2010, pp. 250–288.

[10] A. L. King, L. Feng, O. Sokolsky, and I. Lee, "A modal specification approach for on-demand medical systems," in *FHIES*, ser. Lecture Notes in Computer Science, J. Gibbons and W. MacCaull, Eds., vol. 8315. Springer, 2013, pp. 199–216.

[11] S. Hallé and R. Villemaire, "Runtime enforcement of web service message contracts with data," *IEEE T. Services Computing*, vol. 5, no. 2, pp. 192–206, 2012.

[12] Y. Falcone, L. Mounier, J.-C. Fernandez, and J.-L. Richier, "Runtime enforcement monitors: composition, synthesis, and enforcement abilities," *Formal Methods in System Design*, vol. 38, no. 3, pp. 223–262, 2011.

[13] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *ICSE*, B. W. Boehm, D. Garlan, and J. Kramer, Eds. ACM, 1999, pp. 411–420.

[14] A. Bauer, M. Leucker, and J. Streit, "SALT—Structured assertion language for temporal logic," in *ICFEM*, ser. Lecture Notes in Computer Science, Z. Liu and J. He, Eds., vol. 4260. Springer, 2006, pp. 757–775.

[15] A. Bauer, M. Leucker, and C. Schallhart, "Comparing LTL semantics for runtime verification," *J. Log. Comput.*, vol. 20, no. 3, pp. 651–674, 2010.

[16] ——, "Monitoring of real-time properties," in *FSTTCS*, ser. Lecture Notes in Computer Science, S. Arun-Kumar and N. Garg, Eds., vol. 4337. Springer, 2006, pp. 260–272.

[17] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *CAV*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 171–177.

[18] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The mathsat5 smt solver," in *TACAS*, ser. Lecture Notes in Computer Science, N. Piterman and S. A. Smolka, Eds., vol. 7795. Springer, 2013, pp. 93–107.

[19] L. M. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[20] A. Goldberg and K. Havelund, "Automated runtime verification with Eagle," in *MSVVEIS*. INSTICC Press, 2005.

[21] H. Barringer, D. E. Rydeheard, and K. Havelund, "Rule systems for runtime monitoring: From Eagle to RuleR," in *RV*, ser. LNCS. Springer, 2007.

[22] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for LTL and TLTL," *ACM Trans. Softw. Eng. Methodol.*, 2011.

[23] W. Dong, M. Leucker, and C. Schallhart, "Impartial anticipation in runtime-verification," in *ATVA*, ser. LNCS. Springer, 2008.

[24] N. Decker, M. Leucker, and D. Thoma, "Impartiality and anticipation for monitoring of visibly context-free properties," in *RV*, ser. LNCS. Springer, 2013.

[25] N. Mitra and Y. Lafon, "SOAP version 1.2 part 0: Primer (second edition)," W3C, Tech. Rep., Apr. 2007, http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

[26] K. Liu and D. Booth, "Web services description language (WSDL) version 2.0 part 0: Primer," W3C, W3C Recommendation, Jun. 2007, http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626.

[27] S. Pöhlsen and S. Schlichting, "An architecture for distributed systems of medical devices in high acuity environments - a proposal for standards adoption," *Health Level Seven International*, 2014.

[28] J. Kotamraju, "The Java API for XML-based web services," Oracle, Java Specification Request, 2011.

[29] N. Decker, M. Leucker, and D. Thoma, "jUnit$^{\mathrm{RV}}$—Adding runtime verification to jUnit," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science, G. Brat, N. Rungta, and A. Venet, Eds., vol. 7871. Springer, 2013, pp. 459–464.

[30] R. Alur, K. Etessami, and P. Madhusudan, "A temporal logic of nested calls and returns," in *TACAS*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 467–481.