A comparison framework for runtime monitoring approaches

Rick Rabiser [a,*], Sam Guinea [b], Michael Vierhauser [a], Luciano Baresi [b], Paul Grünbacher [a]

[a] Christian Doppler Laboratory MEVSS, Institute for Software Systems Engineering, Johannes Kepler University Linz, Altenberger Str. 69, 4040 Linz, Austria
[b] Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Piazza L. Da Vinci 32, 20133 Milano, Italy

ABSTRACT

The full behavior of complex software systems often only emerges during operation. They thus need to be monitored at run time to check that they adhere to their requirements. Diverse runtime monitoring approaches have been developed in various domains and for different purposes. Their sheer number and heterogeneity, however, make it hard to find the right approach for a specific application or purpose.

The aim of our research therefore was to develop a comparison framework for runtime monitoring approaches. Our framework is based on an analysis of the literature and existing taxonomies for monitoring languages and patterns. We use examples from existing monitoring approaches to explain the framework.

We demonstrate its usefulness by applying it to 32 existing approaches and by comparing 3 selected approaches in the light of different monitoring scenarios. We also discuss perspectives for researchers.

1. Introduction

The full behavior of a complex software system often only emerges during operation. As a result, testing is not sufficient to determine its compliance with the defined requirements. Instead, engineers and maintenance personnel must always keep track of a system's behavior during operation and check the interactions that occur between its components, as well as between the system and its environment. This is commonly referred to as runtime monitoring.

Many research communities have developed monitoring approaches for various kinds of systems and purposes. Examples include requirements monitoring (Maiden, 2013; Robinson, 2006), monitoring of architectural properties (Muccini et al., 2007), complex event processing (Völz et al., 2011), and runtime verification (Calinescu et al., 2012; Ghezzi et al., 2012), to name but a few. The desired runtime behavior is often formally expressed using temporal logic (Viswanathan and Kim, 2005; Gunadi and Tiu, 2014; Bauer et al., 2006), or through the use of domain-specific (constraint) languages (Robinson, 2006; Baresi and Guinea, 2013; Phan et al., 2008; Vierhauser et al., 2015). Defined constraints are checked based on events and data collected from systems at runtime, e.g., through instrumentation (Mansouri-Samani and Sloman, 1993).

* Corresponding author. Fax: +4373224684341.
E-mail addresses: rick.rabiser@jku.at (R. Rabiser), sam.guinea@polimi.it (S. Guinea).

Existing approaches are very diverse. Some provide end-user tool support (Robinson, 2006; Ehlers and Hasselbring, 2011), while others require expert domain knowledge by their users (Viswanathan and Kim, 2005); some cover specific architectural styles (Baresi and Guinea, 2013), while others are general-purpose (Robinson, 2006); some automatically generate monitors based on models (Robinson, 2006), while others require that probes be manually developed (Vierhauser et al., 2016b). Approaches also differ regarding their expressiveness (Dwyer et al., 1999), e.g., the degree of support to check the occurrence and/or order of runtime events (temporal behavior), the interactions occurring between different (sub-)systems (structural behavior), and/or the properties held by certain runtime data (data checks).

This variety makes it hard to analyze and compare existing approaches. While some efforts have been made to discuss existing work on runtime monitoring – e.g., to create a taxonomy of tools (Delgado et al., 2004) and of (property specification) languages (Dwyer et al., 1999) – there is still no systematic (and easy) way to analyze and compare existing approaches. The main contribution of this paper is, therefore, *a comparison framework for runtime monitoring approaches*. We have developed it based on the results of a systematic review of existing literature (Vierhauser et al., 2016a), building on existing taxonomies for monitoring languages and patterns (Delgado et al., 2004; Dwyer et al., 1999), and taking inspiration from comparison frameworks from other domains such as software architecture or software product lines (Medvidovic and Taylor, 2000; Matinlassi, 2004).

Published Journal Article available at: http://dx.doi.org/10.1016/j.jss.2016.12.034

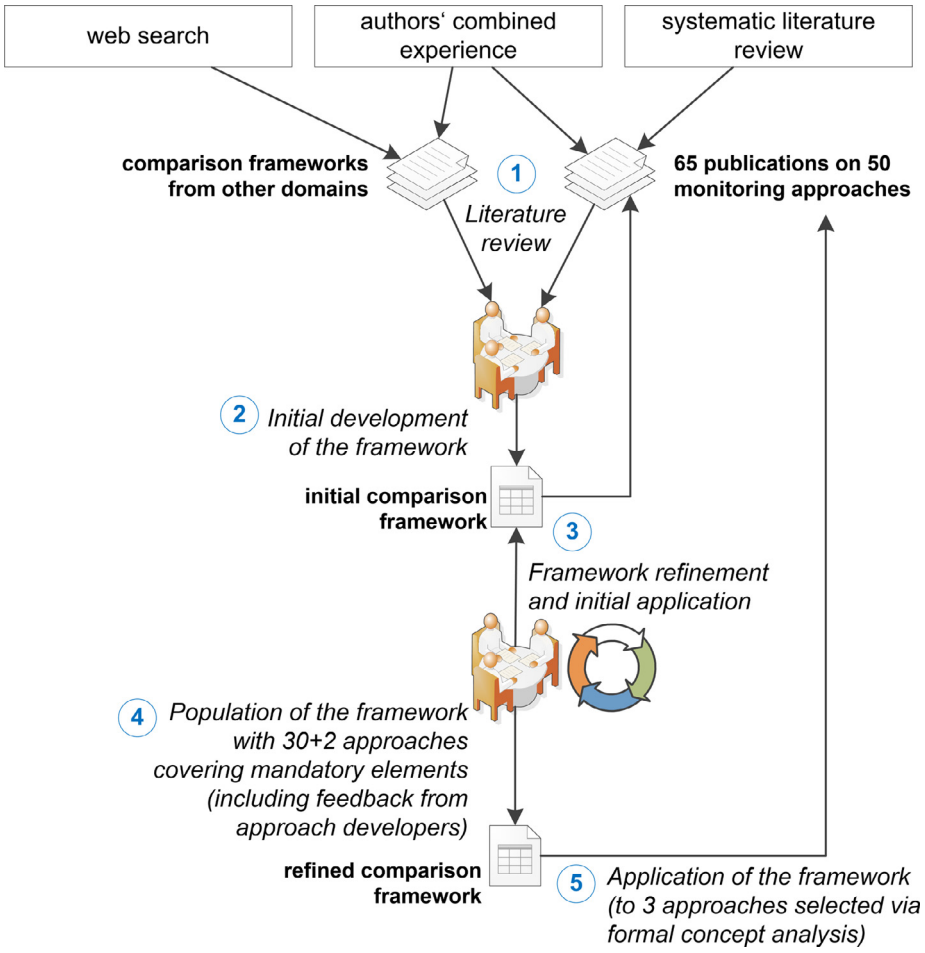

Fig. 1. Research approach.

Specifically, we claim the following contributions: (i) we propose a framework that supports analyzing and comparing runtime monitoring approaches using different dimensions and elements; (ii) we demonstrate how the framework can be applied to analyze existing monitoring approaches found in the literature, and to compare them in detail; and (iii) we discuss perspectives and potential future applications of our framework, e.g., to support the selection of an approach for a particular monitoring problem or application context.

The remainder of this paper is structured as follows. In Section 2 we describe our research approach for developing the comparison framework. In Section 3 we describe the framework by referring to existing runtime monitoring approaches to motivate and explain its different dimensions and elements. In Section 4 we demonstrate the feasibility of the framework by applying it to analyze 32 existing approaches, and by comparing 3 selected approaches in detail in the light of different monitoring scenarios. We conclude by discussing our evaluation results and perspectives for future applications of the framework in Section 5.

2. Research approach

Fig. 1 depicts our research approach, we conducted the research described in this paper in five phases:

(1) *Literature review.* The authors have performed research on runtime monitoring for several years: specifically, three of the authors developed an approach for monitoring requirements of systems of systems in the automation software domain (Vierhauser et al., 2016b), while the other two authors developed approaches supporting runtime monitoring of multi-layered service-based systems (Baresi and Guinea, 2013; Seracini et al., 2014). As part of their research on runtime monitoring, three of the authors recently conducted a systematic literature review (SLR) following existing guidelines (Kitchenham, 2007), to identify, describe, and classify existing approaches for requirements monitoring of software systems at runtime. The aims of this SLR, published 2016 in the Journal on Information and Software Technology (Vierhauser et al., 2016a) were (i) to analyze the characteristics and application areas of the monitoring approaches that have been proposed in different domains, (ii) to systematically identify frameworks supporting requirements monitoring according to the framework definition provided by Robinson (2006), and (iii) to analyze the extent to which these frameworks support requirements monitoring in systems of systems.

In the SLR, searching the digital libraries of IEEE, ACM, ScienceDirect, and Springer (search string: *("run-time" OR "run-time") AND ("monitor" OR "monitoring")*; cf. the appendix of the SLR (Vierhauser et al., 2016a) for detailed search strings split by digital library targeted) resulted in 2201 publications published between 1994 and 2014. When reading the titles and abstracts of the 2201 papers, the authors of the SLR first excluded duplicates and clearly out-of-scope papers, e.g., hardware monitoring approaches, resulting in 1235 publications to be inspected in more detail. The three authors of the SLR then voted on inclusion/exclusion for the 1235 publications based on reading their abstracts applying different criteria. Specifically, they assessed for each of these publications if it describes work that supports continuous requirements

monitoring of software systems at runtime (inclusion criterion). They excluded publications on offline analysis and debugging, pub lications that have not been peer-reviewed, that are not available for download, or that are written in a language other than English.

Publications were only excluded when all authors agreed on exclusion. All 2:1 voting cases were discussed among all authors before making a decision. While voting, the authors also indicated when they were unsure about a vote, which also led to a discussion. Ultimately voting resulted in a selected set of 356 contributions on runtime monitoring of software. The authors then analyzed these remaining papers and selected 65 publications, describing 50 distinct approaches considering at least two of the monitoring framework layers discussed by Robinson (2006). For these 65 publications, the authors extracted detailed information for the SLR. While the SLR assessed and discussed existing support for monitoring in the specific context of systems of systems, the purpose of this paper is to present a general comparison framework for monitoring approaches. We only use the 65 publications identified in the SLR as a basis to develop and assess our comparison framework for monitoring approaches, i.e., both, the SLR and this paper, share a common data set, but report completely different results. For more details on the SLR, we refer the reader to the respective article (Vierhauser et al., 2016a).

(2) *Initial development of the framework.* To develop our comparison framework we took inspiration from existing comparison frameworks for architecture description languages (Medvidovic and Taylor, 2000) and product line architecture design (Matinlassi, 2004). Specifically, we read the 65 identified papers to determine candidate elements for the comparison framework, guided by the dimensions used in Matinlassi's framework (Matinlassi, 2004). This led to an initial list of dimensions (e.g., user, content) and elements (e.g., monitoring language, constraint patterns, trigger, tool support).

(3) *Framework refinement and initial application.* To consolidate these suggestions, we conducted multiple meetings to discuss and iteratively refine the dimensions and elements. This lead to a first prototype comparison framework. We then re-read the 65 publications and tried to categorize and describe the 50 approaches discussed therein using the prototype comparison framework's dimensions and elements. This effort allowed us to further refine our framework, and led to a final set of 4 dimensions and 21 elements (cf. Section 3).

(4) *Population of the framework with approaches covering mandatory elements.* We were not able to assess all 50 approaches against all 21 elements of our framework. This was due to a lack of relevant information in the approaches' publications and web pages. While this may be acceptable for some elements – e.g., an approach does not necessarily have to come with end-user tool support for defining the properties/constraints to be checked – the absence of information about key elements can render it impossible to reasonably compare approaches with each other. Therefore, through discussion, we identified 7 elements that we regard as mandatory for a useful comparison: goal and scope, approach inputs, approach outputs, language, mapping to underlying technology, constraint patterns, and nature of validation. 30 out of the 50 approaches covered all these mandatory main elements and we thus used our framework to analyze and compare these 30. An overview of the results can be found in the appendix of this paper, more details can be found at http://tinyurl.com/moncompfw. Dur- ing a revision of this paper we also contacted the developers of 28 of these 30 approaches (the two remaining approaches were our own) via E-Mail and asked them to check the online spreadsheet regarding the information we had extracted on their approach. We also attached our framework dimensions and elements including a short explanation. We could not reach all approach developers, e.g., because they were deceased or had left academia. Overall, we sent the E-Mail to 84 people, 19 (23%) of which replied. They were involved in the development of 17 different approaches (61%). Most confirmed the extracted information as correct, but some (13 people) additionally suggested minor textual corrections of the information about their approaches. Two also suggested to add their recent approaches. In the revision of the paper we thus made minor textual corrections to the information we had extracted on some of the 30 approaches as requested by the approach developers and also added two additional approaches resulting in a final list of 32 approaches covering the main elements of our framework.

(5) *Application of the framework.* We demonstrate the use of our framework by describing an in-depth comparison of 3 selected approaches regarding their usefulness in different monitoring scenarios (cf. Section 4), to provide the reader with a full understanding of the framework's capabilities. We selected the three approaches from the mentioned 30 through a formal concept analysis (Ganter and Wille, 2012), a technique based on mathematical order theory, which allows to derive a concept hierarchy from a collection of objects and their properties. Each concept in the hierarchy represents the set of objects that share the same values for a certain set of properties; and each sub-concept in the hierarchy contains a subset of the objects in the concepts above it. The end result of a formal concept analysis can be visualized as a concept lattice. We mapped approaches and framework elements to produce such a concept lattice, and used it to understand which approaches provided information on which elements of our comparison framework. This allowed us to select the 3 approaches covering the most elements from our comparison framework.

3. Comparison framework

Table 1 presents our comparison framework; it covers four dimensions – *context, user, content*, and *validation* – and 21 elements (7 of which are mandatory). We now discuss the framework's four main dimensions, using examples from existing monitoring approaches to illustrate the elements in each dimension.

3.1. Context

This section contains the elements that describe how the monitoring approach relates to the application's context. More precisely, these elements tell us the goal and scope of an approach: whether it is tied to a specific domain, architectural style, technical setting, or bound to a specific phase in the application's life-cycle. Further, it covers the inputs and outputs of the approach, and its level of intrusiveness. The goal and scope and the inputs and outputs of the approach are mandatory elements, as for a comparison one needs to at least understand the intention of the approach, the inputs needed to work with it, and the outputs it produces.

A monitoring approach's *specific goal and scope* (mandatory element of our framework) describes the purpose for which it has been developed. For example, ReqMon (Robinson, 2006) supports requirements monitoring, primarily in the domains of enterprise information systems and electronic commerce, while Kieker-1 (Ehlers and Hasselbring, 2011) supports failure diagnosis and performance anomaly detection primarily in Java-based systems. From a high-level perspective, most of the 32 monitoring approaches we analyzed aim at monitoring the runtime compliance of a system to its specification, however, different forms of specifications are used. Some approaches describe their goal and scope in more detail: examples include monitoring and verification of business constraints (Montali et al., 2014) or enforcing program safety and security properties in service-based systems (Aktug et al., 2008).

Table 1
Comparison framework: 4 dimensions and 21 elements to analyze and compare monitoring approaches.

Dimension/Element (* ...mandatory)	Question and examples
1: Context	(see Section 3.1)
1a: Specific goal and scope*	What is the specific goal and scope of the approach?
1b: Life-cycle support	What phases of the software engineering life-cycle does the approach support?
1c: Application domain(s)	What is/are the application domain(s) of the approach?
1d: Architectural style(s)	For what type(s) of architecture(s) is the approach intended?
1e: Approach inputs*	What is the starting point for the approach?
1f: Approach outputs*	What are the results of the approach?
1g: Intrusiveness and overhead	How strong is the approach's impact on the system (e.g., does it run within the system or in parallel)?
2: User	(see Section 3.2)
2a: Target group	Who are the stakeholders addressed by the approach?
2b: Motivation	What are the user's benefits when using the approach?
2c: Needed skills	What capabilities does the user require to apply the approach?
2d: Input guidance	How does the approach guide the user while defining inputs such as constraints?
2e: Output guidance	How does the approach guide the user while managing outputs such as detected violations of constraints?
3: Content	(see Section 3.3)
3a: Language*	What type of language is used to define constraints (e.g., formal language, DSL, high-level representation)?
3b: Reasoning and checking*	What underlying technology supports reasoning on and checking of constraints (e.g., OO language, CEP engine, solver)?What patterns, e.g., by Dwyer et al. (1999) (event ordering, occurrence, data checks, support for fuzzy checks, ...) are covered by the language?
3c: Constraint patterns*	
3d: Data and event manipulation	Does the language support runtime data manipulation (accessing, aggregating, or analyzing data)?
3e: Trigger	How is the instantiation and evaluation of constraints triggered?
3f: Meta-information	Does the approach allow to specify meta-information on constraints (e.g., severity, grouping)?
3g: Variability and evolution	Can constraints be configured in different ways? Is the co-evolution with the monitored system supported?
4: Validation	(see Section 3.4)
4a: Nature of validation*	How has the approach been validated (industrial case studies, formal proof/scientific evaluation, (toy) examples)?
4b: Availability and support	Is the approach (still) available and how is it supported?

The *life-cycle support* element describes the phases of the software engineering life-cycle in which a monitoring approach will be primarily used. For instance, some approaches detect and diagnose deviations from requirements during maintenance and operation (Robinson, 2006; Spanoudakis and Mahbub, 2004), while others focus on the requirements engineering phase (Robinson, 2006).
Yet others aim at performance analysis and code (instrumentation) (Ehlers and Hasselbring, 2011), and thus focus more on the implementation phase.

Many monitoring approaches have been developed for a particular *application domain*, such as enterprise information systems (Robinson, 2006), service-based systems (Baresi and Guinea, 2013), or systems of systems (Vierhauser et al., 2016b). Others are more generally applicable in different domains (Montali et al., 2014), or are reported as being applicable in multiple domains, although they have been used in only one domain so far, e.g., mobile systems developed in Android (Gunadi and Tiu, 2014). Of the 32 approaches we analyzed most were developed for service-based systems, followed by approaches supporting business process monitoring.

The *architectural style(s)* element defines the type(s) of architecture(s) a monitoring approach can be used with. Depending on how this element is described, it allows us to derive the types of technologies the approach might work for. For example, many approaches we analyzed were developed for service-based systems. However, some are centered on Service-Oriented Architectures (SOA) using the Business Process Execution Language (BPEL), e.g., (Spanoudakis and Mahbub, 2004), while others use the Service Component Architecture (SCA), e.g., (Baresi and Guinea, 2013).

The mandatory element *approach inputs* addresses the kinds of inputs expected by the monitoring approach. Often, the input is a specification, e.g., written in BPEL, which is used to generate monitors for service-based systems (Spanoudakis and Mahbub, 2004). Other approaches require monitors to be specified in their own language, e.g., existing languages such as the Object Constraint Language (OCL) (Robinson, 2006) or custom-made domain-specific languages (Vierhauser et al., 2016b). Recorded event traces (Faymonville et al., 2014) or goal models (Ramirez and Cheng, 2011) can also serve as an input.

The mandatory element *approach outputs* addresses the kinds of outputs the monitoring approach provides to its users. For approaches that focus on runtime verification (Faymonville et al., 2014), the output can be simply TRUE or FALSE. Other approaches (Jeffery et al., 2004; Ehlers and Hasselbring, 2011) generate code that is used to instrument systems for monitoring. For instance, FORMAN (Jeffery et al., 2004) generates C code from behavior specifications (event grammar). Some approaches visualize system behavior (Robinson, 2006), performance characteristics (Ehlers and Hasselbring, 2011), or provide information on the violations of requirements (Vierhauser et al., 2016b).

The element *intrusiveness and overhead* describes the impact of using a monitoring approach on a (running) system. While many approaches run within the system they monitor (Robinson, 2006), others run in parallel (Montali et al., 2014). Some approaches are just listening (Vierhauser et al., 2016b), while others aim to actively adapt the running system (Baresi and Guinea, 2013). Many approaches measure the overhead for the monitored system as a way to demonstrate their feasibility.

3.2. User

This dimension covers the elements that describe the targeted stakeholders as well as their required skills and capabilities for using the approaches. While user focus and guidance are essential to make an approach useful, information about such user elements is not required to understand how approaches work in general. Thus, this dimension of our framework does not contain any mandatory elements.

The *target group* element captures which stakeholder roles, e.g., software developers, testers, software architects, or even end users, are supported by the approach. Unfortunately, the papers we analyzed were typically not very specific in this regard, and often we could only define software engineers or service engineers as the intended users of the approach.

The *motivation* for applying the approach is an important element; it describes the users' benefits when using the approach. Our review revealed a wide range of reported benefits, such as detecting Service-Level Agreement (SLA) violations (Contreras and Mahbub, 2014), detecting compliance issues in service-based systems (Holmes et al., 2011), discovering performance leaks (Ehlers and Hasselbring, 2011), detecting security problems (Gunadi and Tiu, 2014), checking business process conformance (Poppe et al., 2013), checking safety properties (Kim et al., 2001), monitoring constraints on business processes (Montali et al., 2014), checking program behavior (Jeffery et al., 2004), detecting mismatches in service interactions (Baouab et al., 2012), etc. FORMAN (Jeffery et al., 2004) has recently also been extended to support system and software executable architecture modeling and is also the basis for a business process modeling framework called Monterey Phoenix (Auguston et al., 2015).

The *needed skills* element refers to the capabilities that a user needs to possess to properly apply an approach and perform its monitoring tasks. Typically, the approaches we investigated require user skills in several areas including formal background (e.g., LTL, Event Calculus) (Contreras and Mahbub, 2014; Faymonville et al., 2014; Spanoudakis and Mahbub, 2004), experience with different domain-specific languages (Holmes et al., 2011; Paschke, 2005; Aktug et al., 2008), the capability to specify rules (Ehlers and Hasselbring, 2011) or queries (Baouab et al., 2012), programming skills for writing probes (Vierhauser et al., 2016b), or modeling skills (Ramirez and Cheng, 2011).

Our framework also addresses how the approaches guide the users in defining constraints, i.e., through element *input guidance*. Monitoring approaches, for example, may provide text-based editors for defining rules and constraints (Aktug et al., 2008; Ehlers and Hasselbring, 2011; Conforti et al., 2013), or graphical languages for defining the expected behavior (Montali et al., 2014). Few approaches propose or discuss specific tool support (Faymonville et al., 2014; Jeffery et al., 2004; van Hoorn et al., 2012).

Similarly, through element *output guidance*, our framework considers how approaches guide the users in managing the outputs of the runtime monitoring approach, e.g., by visualizing evaluation results or providing diagnosis support. Some approaches provide web-based interfaces or dashboards (Robinson, 2006; Kumar Tripathy and Patra, 2010; van Hoorn et al., 2012). More advanced techniques support adaptations of the system (e.g., by switching services) (Contreras and Mahbub, 2014). Visualizations include calling context trees, responsiveness time series (Ehlers and Hasselbring, 2011), or violations (Montali et al., 2014). Again, the majority of the approaches do not propose or discuss specific tool support.

3.3. Content

The *content* dimension contains elements describing more technical aspects of the monitoring approaches. It contains three mandatory elements one must understand to compare approaches: the used monitoring language, the underlying reasoning and checking mechanism, and the supported constraint patterns.

The *language* an approach uses to define the properties to monitor is an important mandatory element of our framework. Approaches provide a variety of languages, including formal languages (Faymonville et al., 2014), domain-specific constraint languages (Vierhauser et al., 2016b), and model-based representations such as UML sequence diagrams (Simmonds et al., 2009).

Some approaches rely on existing languages or formalisms: ReqMon (Robinson, 2006), for example, uses the Object Constraint Language while MORPED (Contreras and Mahbub, 2014) and Mobucon EC (Montali et al., 2014) are based on the Event Calculus. Of the 32 monitoring approaches we analyzed, almost one third propose their own Domain-Specific Language (DSL) for specifying properties or constraints; typically these are tailored to the needs of the users and the types of constraints and properties (Kumar Tripathy and Patra, 2010; Kim et al., 2001). Our own approaches (Vierhauser et al., 2015; Baresi and Guinea, 2013) also use DSLs.

The *reasoning and checking mechanism* is a mandatory element describing the actual engine used to validate the monitored constraints. This could be a high-level programming language or a formal verification framework, hidden from the user through a constraint definition language. In the domain of service-based systems and business processes, Complex Event Processing (CEP) engines (Luckham, 2011) such as Drools Fusion and Esper are widely used (Holmes et al., 2011; Inzinger et al., 2013). However, there are also many custom-tailored mechanisms for evaluating rules and constraints (Vierhauser et al., 2016b; Gunadi and Tiu, 2014; Jeffery et al., 2004; Faymonville et al., 2014).

Depending on the application area, the target domain, and the designated users of the approach, different *constraint patterns* (Dwyer et al., 1999) are typically supported – a mandatory element of our framework. For instance, approaches check specific sequences of events that have to occur, the presence and absence of a specific event, or data/performance properties. ReqMon (Robinson, 2006), for example, provides support for defining the expected order and occurrence of certain events using real-time temporal operators, while Kieker-1 (Ehlers and Hasselbring, 2011) allows checking performance characteristics. When aiming at system adaptation, some approaches (Inzinger et al., 2013; Contreras and Mahbub, 2014) also use event-condition-action (ECA) rules to directly react to certain events or constraint violations.

Monitoring approaches can also provide support for *data and event manipulation*. This can include capabilities for aggregating data from different events (e.g., calculating sums or average values), defining arbitrary user-specific functions, performing statistical analyses, as well as analyzing complex event data. For example, YAWL (Conforti et al., 2013) and MaC (Kim et al., 2001) provide means for defining additional auxiliary variables. ReqMon (Robinson, 2006) and RBSLA (Paschke, 2005) allow defining filters and data aggregators, which can be used for statistical analysis.

The *trigger* element describes the mechanisms used to initiate constraint evaluation. Most approaches are event-based, i.e., constraint evaluation is triggered when a specific event occurs (Kim et al., 2001; Holmes et al., 2011). Other approaches (Conforti et al., 2013; Ehlers and Hasselbring, 2011) use a periodic trigger, or become active when a monitored component is started (Simmonds et al., 2009).

The *meta-information* element provides additional info on constraints, e.g., describing the severity of a violation or its assignment to a particular component of a system. For instance, MORPED (Contreras and Mahbub, 2014) allows users to define user con- text models in addition to constraints.

The *variability and evolution* element refers to the configurability and adaptability of an approach and its constraints. Co-evolution of the monitoring infrastructure with the monitored system is often necessary to react to newly emerging or changing requirements, or to changes that occur in the structure and architecture of the monitored system. This may include adding new constraints dynamically, activating or deactivating constraints based on certain monitoring scenarios, and modifying or parameterizing constraints to match the system variant to be monitored. Based on our observations, most approaches have not considered this issue so far. Exceptions are, e.g., Kieker-1 (Ehlers and Hasselbring, 2011), Kieker-2 (van Hoorn et al., 2012) and REMINDS (Vierhauser et al., 2016b), which allow activating and deactivating probes dynamically.

3.4. Validation

This framework dimension contains information about how a monitoring approach has been validated, and to what extent it is publicly available. This dimension is essential to understand the potential impact of an approach on practice. At least the nature of the validation must be understood in order to compare monitoring approaches, which is why this element is mandatory.

The mandatory *nature of validation* element assesses the degree and rigor of the scientific validation of the monitoring approach. Most tools provided by monitoring approaches are research-oriented prototypes and thus their validation must be framed in this context. Many approaches have only been assessed through simple examples or through systems that are "friendly enough". Only in a few cases did cooperations with industry (Montali et al., 2014) and/or EU-sponsored projects (Robinson, 2006; Paschke, 2005) help create proper sandboxes for the proposed monitoring solutions.

Regarding the *availability and support* element, most of the tools and frameworks discussed in the papers we analyzed are not available (anymore) to the public. Many papers do not provide information about the availability of the artifacts in the paper itself, requiring a web search or to contat authors. In some cases, the tools are no longer available, since they are no longer maintained. In a few exceptional cases, the tools are still available, or at least the code can be retrieved from personal web pages (Poppe et al., 2013), open repositories like GitHub or sourceforge (Robinson, 2006; Ehlers and Hasselbring, 2011), or they require some special-purpose license (Montali et al., 2014; Vierhauser et al., 2016b).

4. Evaluation: applying the comparison framework

One of our framework's main goals is to enable the comparison of existing approaches, with the objective of establishing which can be deemed applicable, and/or better suited, given a specific monitoring scenario. In this section we shall demonstrate how this can be achieved, with regard to two specific scenarios. In the first scenario, we assume that maintenance personnel are interested in monitoring event sequences in service-based systems to investigate a reported issue. This scenario requires an approach through which they can (i) instrument service-based systems, (ii) specify constraints on event sequences, and (iii) receive valid end-user support. In the second scenario, we assume that engineers are interested in monitoring the performance of a Java-based application to identify performance leaks. This requires an approach through which they can (i) instrument Java-based systems, (ii) specify performance checks, and (iii) visualize the monitored performance.

From the 32 monitoring approaches we analyzed (cf. appendix), we selected 3 approaches to demonstrate the use of our comparison framework. As previously mentioned, we performed a formal concept analysis (Ganter and Wille, 2012) to identify the approaches that covered the most elements of our framework. To avoid bias, we excluded our own approaches and the two additional approaches suggested by the approach authors we contacted (cf. research method). We then selected three approaches with different goals and application areas. Specifically, we selected the Mobucon-EC approach (Montali et al., 2014) for the runtime verification and monitoring of business processes; the ReqMon approach (Robinson, 2006), which focuses on requirements monitoring of enterprise information systems; and the Kieker-1 approach (Ehlers and Hasselbring, 2011), which supports application performance monitoring.

Regarding their *context* and scope – i.e., runtime verification, requirements monitoring, and performance monitoring – both ReqMon (monitoring of enterprise information systems) and Mobucon-EC (process monitoring) have a clear focus, while Kieker-1 is more generally applicable. ReqMon and Mobucon-EC would both support distributed systems with a service-oriented architecture. The three approaches expect very different inputs, i.e., OCL monitoring rules (Kieker-1), constraints (Mobucon-EC), and requirements specifications (ReqMon). The outputs also differ significantly and include violations (Mobucon-EC), performance anomalies (Kieker-1), and generated monitors (ReqMon). Regarding intrusiveness, both Kieker-1 and Mobucon-EC run mostly in parallel to the monitored system using existing event streams, while ReqMon can have a more significant impact depending on the instrumentation technology applied (instrumentation is generated but then potentially runs within the monitored system).

All three approaches are intended to support engineers as their primary *user* group. Regarding the required skills both Mobucon-EC and ReqMon require the user to learn a domain-specific language, while Kieker-1 utilizes OCL. All three approaches provide tool support to ease writing constraints. Kieker-1 and ReqMon also provide rather sophisticated means for visualizing the monitoring results, i.e., monitored performance information and detected constraint violations.

Regarding the *content* dimension, Kieker-1 links its OCL monitoring rules to an underlying (aspect-oriented) instrumentation, while ReqMon maps it to a domain-specific event sink provided by the Microsoft Instrumentation Framework. While both approaches are extensible, existing implementations mainly focus on a particular technology, i.e., Java (AspectJ) for Kieker-1 and .NET for ReqMon. Mobucon-EC is more generic and maps its DSL to Event Calculus, allowing the use of existing solvers. A key distinguishing element are the types and patterns of constraints supported by the approaches. For example, Kieker-1 mainly focuses on performance calculations (even though OCL can be used to express more general patterns), while ReqMon supports checks on event ordering and occurrence as well as data checks and ECA rules. Constraint evaluation is triggered by events in Mobucon-EC and ReqMon, and periodically in Kieker-1. Kieker-1 and ReqMon provide support to activate and deactivate probes based on monitored information, e.g., to reduce the monitoring overhead.

All three approaches have been *validated*; however, the reported case studies and examples focus mainly on demonstrating the general feasibility and on assessing the monitoring overhead. Other aspects, e.g., the usefulness of their constraint language and the provided tool support have not been scientifically evaluated. In terms of availability, Kieker-1 has the biggest community and is frequently updated, which eventually led to the new version Kieker-2 (van Hoorn et al., 2012).

Our comparison summarized in the appendix of this paper reveals that ReqMon could be useful in the first scenario, to support maintenance personnel in monitoring event sequences in service-based systems, even though it might not appear so from a first look at the context dimension. While ReqMon's *goal and scope* (requirements monitoring) is more general than the goal of the first scenario (monitoring event-sequences in service-based systems), and its *application domain* (enterprise information systems) is not necessarily service-based systems, a closer look at the *architectural style* (SOA) and *approach inputs* (policies) elements reveals that it could be actually useful for service-based systems. In the first scenario maintenance personnel are the *target user group*. While ReqMon is more targeted towards (requirements) engineers, the provided *input guidance* (probes are generated from templates and a RequisitePro tool extension allows specifying monitoring goals) and *output guidance* (ReqMon's Presenter provides a digital dashboard presenting monitoring results in a way suitable for requirements analysts) would allow maintenance personnel to work with it. Furthermore, the *constraint patterns* supported by ReqMon include occurrence and ordering of events as required in the first scenario, and the *trigger* of ReqMon's constraint engine are

events. While ReqMon's *evaluation* demonstrates its low overhead and principle feasibility, a detailed, scientific evaluation has not been conducted, particularly not in the domain of service-based systems.

Mobucon-EC could also be used in the first scenario. Its main *goal and scope* is runtime verification and monitoring of business processes during operation, which matches with the need to check constraints on event sequences. Its *application domain* (process monitoring) and *architectural style* (sets of systems enacting processes) would be suitable for the first scenario, because business processes are often realized as services and service-based systems are indeed sets of (distributed) systems. Mobucon-EC provides *input guidance* in the form of a graphical notation for defining constraints in Event Calculus; it is also capable of visualizing violations (*output guidance*). However, the approach still requires experience to specify constraints, and output guidance is not as sophisticated as the one provided by ReqMon. In terms of supported *constraint patterns* Mobucon-EC supports quantitative time constraints and non-atomic, durative activities, i.e., it can be used to monitor event sequences. The *trigger* of its underlying Event-Calculus-based reasoning engine are also events. Mobucon-EC has been *evaluated* in a case study about maritime safety and security. Mobucon-EC could also be used in the first scenario. Its main *goal and scope* is runtime verification and monitoring of business processes during operation, which matches with the need to check constraints on event sequences. Its *application domain* (process monitoring) and *architectural style* (sets of systems enacting processes) would be suitable for the first scenario, because business processes are often realized as services and service-based systems are indeed sets of (distributed) systems. Mobucon-EC provides *input guidance* in the form of a graphical notation for defining constraints in Event Calculus; it is also capable of visualizing violations (*output guidance*). However, the approach still requires experience to specify constraints, and output guidance is not as sophisticated as the one provided by ReqMon. In terms of supported *constraint patterns* Mobucon-EC supports quantitative time constraints and non-atomic, durative activities, i.e., it can be used to monitor event sequences. The *trigger* of its underlying Event-Calculus-based reasoning engine are also events. Mobucon-EC has been *evaluated* in a case study about maritime safety and security.

Kieker-1 would definitely be the most suitable of the three approaches when it comes to the second scenario. For instance, its *goal and scope* are performance anomaly detection and failure diagnosis during operation. While no particular *application domain* is emphasized by Kieker-1, component-based architectures (implemented in Java) are the mainly supported *architectural style*. Engineers are the *target group* and are supported with a rule editor (*input guidance*). Visualizations that can be provided by Kieker-1, such as calling context trees and responsiveness time series, would definitely help engineers to monitor the performance of their Java-based applications. Kieker-1's support can easily be integrated into an engineer's development environment. Furthermore, Kieker-1 also automates instrumenting Java-based applications through aspect-orientation and it requires writing (and generating) OCL monitoring rules, which suits our second scenario very well. Supported *constraint patterns* include performance calculations. The approach has been *evaluated* in multiple case studies.

5. Discussion and perspectives

The application of our framework to 50 monitoring approaches – 32 of which we analyzed in detail and 3 of which we compared regarding their usefulness for different scenarios – demonstrates that our framework can be adopted by researchers and practitioners. The framework made it easier for us to extract relevant information from the papers we were analyzing, by emphasizing the important aspects of runtime monitoring and by clearly distinguishing the context, user, content, and validation dimensions.

While our framework is an informal and qualitative approach, its dimensions and elements provide a structure to the knowledge that is hidden within the papers, making them more accessible and more easily comparable. The framework especially shines when the user is interested in finding an approach that is suitable for her/his domain, architectural style, or technical setting, as we demonstrated in Section 4.

The framework also helped us to uncover several issues that we believe should be known by the monitoring community at large. Specifically, we noticed that *most approaches do not provide a proper validation, but have only been assessed through simple examples or through systems that are "friendly enough"*. Clearly, most research prototypes cannot be evaluated on complex, real cases already at the very beginning of a research project; nevertheless, we believe that more attention should become customary when evaluating a produced solution. When it comes to practitioners seeking for a runtime monitoring approach for their system or domain, it becomes hard for them to estimate the actual value or applicability of an approach without proper knowledge of its limitations regarding performance and/or scalability.

Most of the tools and frameworks discussed in the papers we analyzed are not available (anymore) to the public/community. This sounds quite odd. We believe that the reader should be granted access to the artifacts that are discussed within a paper; if not, there should be a credible reason (e.g., non-disclosure agreements). In the long run, by making the tools open source, and by publicly providing its artifacts via some form of repository, we can ensure that valuable knowledge about the approach is preserved. This is true even if the approach is, in itself, no longer maintained. If this is not done, many excellent approaches and lessons that were once learned will be lost, often leading the community to have to "reinvent the wheel" over and over again.

Another observation is that *only few approaches we analyzed discussed variability and evolution*. In practice, supporting the coevolution of the monitoring solution and the monitored system is essential, as modern systems are often highly variable and frequently change as requirements and context parameters change. Practical use of monitoring solutions requires adding new constraints dynamically, activating or deactivating constraints at runtime to support particular monitoring tasks or users, and modifying or parameterizing constraints according to the system variant that is being monitored.

While most approaches require that one learns some kind of language (BPEL, XPath, a custom-developed DSL, etc.), *many approaches do not provide much end-user tool support for working with the languages* and generally only very few provide fully-fledged tools with visualizations and guidance for users. The target user group seems to be mainly (experienced) engineers. This again hinders the adoption of the proposed monitoring approaches, and also makes it difficult for other researchers to experiment with existing approaches.

We also learned from our analysis that *service-based systems and business processes are the application domains of most monitoring approaches*. One could argue that research on service-based systems has lost momentum in the last few years, and that smart environments (e.g., spaces, buildings, and even cities) have become the current trend. Given the complexity of these environments, it is obvious that they will also require suitable monitoring and runtime adaptation solutions. Curiously, however, the actual software backbones for these smart environments often employ service-oriented architectures and technology. This means that, even if research on monitoring and adapting service-based systems has lost momentum, the adoption and contextualization of existing solutions can still be considered in its infancy. We advocate that

our framework can provide valuable insights on alternatives when looking for an appropriate solution, or at least shed light on what the appropriate ingredients for a new tailored approach might be.

Overall, our key observation from analyzing 50 monitoring approaches is that a lot of useful work has been done over the years, and that many interesting approaches have been developed. To preserve the knowledge and artifacts that have been developed, and to not render the researchers' efforts meaningless, we believe that the community should spend some additional effort. The community needs to go beyond publishing papers. Our comparison framework is a first step in this direction; it allows researchers to extract and preserve information about existing approaches in a structured way.

In our future work, we aim to continue our work and include more approaches. Of course we are interested in the feedback of the monitoring community. We already contacted the developers of the approaches and asked them to validate the information we have extracted. We also plan to involve additional people from the community in populating and maintaining a living library of compared approaches. For this purpose, we are interested in creating a virtual meeting place for the community, independent of editorials, books, and universities. Our on-line spreadsheet (http: //tinyurl.com/moncompfw) is a good starting point. It will make our framework widely available, and allow us to collect and share materials and tools in a more meaningful way. Additionally, dedicated web communities in social networks such as LinkedIn, ResearchGate, or Mendeley could be created. To make our framework more easily applicable by others, we also plan to develop guidelines for extracting information from papers. Our framework could even be used to create a tool for selecting an existing approach, e.g., by creating a variability model (Czarnecki et al., 2012) from the populated spreadsheet and using existing product line configuration tools (Rabiser et al., 2012).

Acknowledgments

This work has been partially supported by project EEB - Edifici A Zero Consumo Energetico In Distretti Urbani Intelligenti (Italian Technology Cluster For Smart Communities) - CTN01_00034_594053, the Christian Doppler Forschungsgesellschaft Austria, and Primetals Technologies.

Appendix. Comparison framework: application to 32 selected approaches

Tables 2–8 present an overview of the results of applying our comparison framework to 32 selected approaches. Please refer to http://tinyurl.com/moncompfw for more details. Please note that for each approach, in the tables, we only link one representative publication (a click on the approach name leads to this publication). Please also note that for 4b (availability), for some approaches, we integrated hyperlinks. Three approaches are highlighted because we used them in Section 4 to demonstrate the use of our framework in an in-depth comparison of approaches regarding their usefulness in different monitoring scenarios. Please also note that two of the authors of this paper are developers of the EcoWare approach (Baresi and Guinea, 2013) and the three other authors of this paper are developers of the REMINDS approach (Vierhauser et al., 2016b).

Table 2
Comparison framework: application to 32 approaches (cf. Table 1): 1–5.

#	Agent monitoring	aPro	athena	Backmann_CEP	BPEL4WS
1a*	behavior analysis	assess compliance to goals based on KPIs	requirements monitoring	business process monitoring	correctness of BPEL processes
1b	testing & operation	business process lifecycle	requirements & operation	operation	requirements & operation
1c	multi-agent systems	business process models	adaptive systems	business processes	services
1d	multi-agent architectures	event-based architectures	unclear	SOA/CEP	SOA/BPEL
1e*	action language	BPMN model & KPIs	goal model	extended/annotated BPMN	BPEL & external annotations
1f*	info whether system behaves as planned	CEP results & div. generated (probes, rules, etc.)	values of utility functions	CEP results	notifications to user
1g	runs within the system	runs within the system	unspecified	unspecified	runs in parallel
2a	testers	service engineers & business users	engineer	engineers	service engineers
2b	understand system behavior	alarms on goal violations	check satisfaction of rqts at runtime	monitor KPIs	find violations
2c	action languages	BPMN & aPro notation	goal modeling & fuzzy logic	KPIs	Event calculus & fluents
2d	agents & action languages	formula lang. editor	none	none	unclear
2e	unclear	dashboard, anal. comp. & data warehouse	historical values	unclear	unclear
3a*	action language	models based on BPMN & annotations	RELAXed KAOS goal models	Annotated BPMN models	proprietary language based on event calculus & fluents
3b*	MAS	Petri nets & CEP	fuzzy & conventional logics	CEP engine	integrity constraints
3c*	behavior checks of agents	timing sequences	event occurrence, event order, data checks & fuzzy logic	event occurrence, event order, & data checks	event occurrence, event order, & data checks
3d	no	through CEP	through KAOS	through CEP	unclear
3e	messages exchanged among agents	events	unclear	events	events
3f	no	no	no	no	no
3g	no	regenerate rules	RELAX goals are adaptable	no	no
4a*	example	research and ind. examples	example	example	example, used in div. projects
4b	N/A	comm. tool	N/A	Unicorn	N/A

Table 3
Comparison framework: application to 32 approaches (cf. Table 1): 6–10.

#	Choreographies Structure Trees	CHOReOS	ConSpec	ECE Rules	EcoWare
1a*	service composition monitoring	service composition monitoring	program safety & security monitoring	process property monitoring	collecting, aggregating, & analyzing runtime data
1b	operation	operation	maintenance & policy specification	maintenance	operation
1c	business processes & services	services	services & program verification	processes	services
1d	Event-driven arch.	SOA	SOA	SOA & Event-driven arch.	SOA
1e*	service composition & events	external rules written in DSL	annotated programs transformed to automata	expectations and properties	event bus (Siena P/S Bus)
1f*	CEP queries & violations	violations	alerts	violations & recovery actions	KPIs & aggregated/analyzed data
1g	unspecified	runs within the system	unspecified	unspecified	runs in parallel
2a	service engineers	engineers	(service) engineers	engineers	(service) engineers
2b	detect mismatches in service interactions	monitor service compositions	detect security violations	detect violations & repair system	analyze behavior of multi-layered systems
2c	no specific skill needed	Drools	programming language skills	engineering skills & Drools rules	DSL (mlCCL)
2d	no constraints are defined manually	unclear	ConSpec Editor & Monitoring Pattern Repository	none	templates for generating adapters & aggregators
2e	dashboard	none	notification module	none	dashboard
3a*	CEP queries	DSL	Policy Specification Language	ECE Rules/Drools	CEP-based DSL (mlCCL)
3b*	CEP engine	CEP engine	automata & Drools rules	Drools fusion & CEP	CEP engine
3c*	occurrence/ordering of events	occurrence/ordering of events & data checks	occurrence/ordering of events	occurrence/ordering, custom functions	event occurrence & data checks
3d	unclear	unclear	unclear	Drools (data access, functions, ...)	aggregation & analysis of arbitrary data
3e	events	events	defined by 'scope'-method or variable	events	events
3f	no	no	no	expectation meta-model	no
3g	no	no	no	no	no
4a*	example	example	information services case study, implementation	use case of clinical support system	examples
4b	POC	developed within EU project	EU project, tools & code in github	N/A	researchers still active

Table 4
Comparison framework: application to 32 approaches (cf. Table 1): 11–15.

#	HIT	Kieker-1	Kieker-2	MaC	Mobucon EC
1a*	workflow monitoring	failure diagnosis & performance anomaly detection	generic monitoring	execution sequence correctness monitoring	runtime verification & business process monitoring
1b	maintenance & operation	operation	operation, profiling & re(verse) engineering	unspecified	operation
1c	workflows & business processes	not discussed	generic	reactive systems	processes
1d	stream processing applications	component-based systems	component-based and OO systems	reactive systems	systems enacting processes
1e*	event streams	measuring points	Code instr. or models to generate inst.	requirement specifications	constraints defined in Declare
1f*	constraint violations	performance anomalies	measurements & arbitrary analyses	deviations from expected event sequences	violations
1g	runs in parallel	runs in parallel	dep. on instr. approach	runs in parallel	runs in parallel
2a	engineers	engineers	engineers and sys. ops.	engineers	engineers & business process designers
2b	detect workflow deviations	detect performance problems	div. analyses	check safety properties	check business constraints
2c	DSL	OCL	programming skills	MEDL/PEDL language	Declare language
2d	none	editor to write rules	Eclipse IDE Plug-In	none	graphical notation
2e	none	call context trees & responsiveness time series	GUI for Trace Analysis/Diagnosis & WebGUI	none	visualization of violations & functions
3a*	DSL to query event stream	OCL monitoring rules	arbitrary via analysis plug-ins	DSL with events & conditions	(graphical) Declare notation
3b*	custom impl.	inference engine	arb. via analysis plug-ins	formal	Event Calculus
3c*	complex temporal checks, causality & data checks	performance checks	arb. via analysis plug-ins	event sequences, safety properties & alarms & data checks	business constraints, quantitative time constraints & non-atomic, durative activities
3d	no	no	arb. via analysis plug-ins	auxiliary variables	Prolog version implemented in Java
3e	events	time intervals	events or time intervals	events	events
3f	no	no	no	no	no
3g	no	probes can be (de)activated	probes can be (de)activated	no	no
4a*	running example; formal semantics	case study on performance overhead	lab experiments & ind. case studies	formal discussion & case studies	case study about maritime safety & security; benchmark
4b	CEA	kieker-monitoring.net	kieker-monitoring.net	author can provide download	personal homepage

Table 5
Comparison framework: application to 32 approaches (cf. Table 1): 16–20.

#	Monina	MOP	MORPED	MORSE	MSL
1a*	monitoring & adaptation	monitoring-oriented prog.	error detection & adaptation	compliance monitoring	compliance monitoring
1b	deployment	development & operation	maintenance	development & maintenance	maintenance
1c	virtual service platforms	arbitrary	SOA	SOA	SOA
1d	virtual services	Java-based systems	SOA	SOA	SOA
1e*	monitoring queries & adapt. rules in Monina	formal specs & logical statements	BPEL specifications, annotations & event calculus	models of services and processes	process execution engine
1f*	deployment considers av. resources	monitors	EC formulae & adaptations	info on compliance	violations
1g	runs in parallel	runs within the system	runs within the system	runs within the system	runs in parallel
2a	engineers	programmers	engineers	engineers	not specified
2b	specify adaptation rules	improve reliability	detect SLA violations/adaptation	check compliance	check compliance
2c	Monina language	annotation language	formal background	EPL, XPath & (WS)-BPEL	MSL
2d	Eclipse Plugin to create Monina rules	logic plugins	none	Runtime Client	Web page
2e	none	none	sys. adaptation	root cause analysis	web page
3a*	Monina DSL	JASS or JML annotations	Event Calculus	EPL/Xpath	Monitor Specification Language (MSL)
3b*	CEP engine	arbitrary via logic plug-in modules	Java	CEP engine	MSL compiler
3c*	facts & events, actions, components, monitoring queries & adaptation rules	depends on logic plugin	timing sequences, occurrence & data aggregations	timing sequences & occurrence	Boolean, temporal & numerical/statistical formulae
3d	aggregation, basic stat. anal. & event generation	can be done in host language	functions & event generation	depends on the CEP engine	statistical formulae
3e	events	logic statements in PL	events	events	events
3f	no	no	yes, in separate files though	compliance meta-data model	no
3g	no	no	no	variability in service models, VCS	no
4a*	example from FP7 project	benchmarks	(scientific) experiment using simulators	(fictitious) case study	examples
4b	GitHub.io	GitHub	N/A; researchers still around	download	N/A

Table 6
Comparison framework: application to 32 approaches (cf. Table 1): 21–25.

#	MuloEtAl_DSL	PLTL	RBSLA	REMINDS	ReqMon
1a*	compliance monitoring	runtime verification	contract performance monitoring	monitoring systems of systems	requirements monitoring
1b	development & maintenance	maintenance & testing	maintenance	operation, requirements specification & evolution	requirements specification & maintenance
1c	SOA	not specified	service-based systems	distributed, component-based systems	enterprise information systems
1d	SOA	not specified	SOA	systems of systems	SOA
1e*	event patterns/DSL	event traces	event streams	requirements, event & scope model	requirements specs, policies & ebXML
1f*	info on compliance	verification result	adaptation	evaluation results	monitors
1g	runs within the system	runs in parallel	unspecified	runs in parallel	runs within the system
2a	engineers	engineers	engineers	engineers & service staff	(requirements) engineers
2b	check compliance	verification	monit SLAs & adaptation	detect deviations from requirements	detect req. violations/perf. leaks
2c	DSL	formal background	RuleML/RBSLA languages	programming skills for probes	programming skills & DSLs
2d	MDD tool	none	none	monitoring IDE	several
2e	reports	none	none	monitoring IDE	dashboard
3a*	DSL	formal language (PLTL)	DSL based on RuleML	DSL	MSL based on KAOS & OCL
3b*	Esper engine	custom impl.	RuleML engines	custom impl.	domain-specific
3c*	event ordering, occurrence & data checks	event occurrence & ordering	event ordering, event occurrence & data checks	event ordering, event occurrence & data checks	event ordering, event occurrence, data checks & ECA rules
3d	filters	no	data in attachments, new events can be generated	data attached to events, aggregated via processors	data aggregations, functions & statistical analyses
3e	events	events	events	events	events
3f	activities & filters	LTL	monitoring schedule	requirements & constraint attribs	scopes, groups & responsibilities
3g	no	no	no	via decision models	ReqMon Configuration API
4a*	experiment with example programs & ind. case studies	experiments	examples	industrial use case & performance evaluation	example & experiment
4b	N/A; researchers still around	N/A	N/A	research license	researcher still active

Table 7
Comparison framework: application to 32 approaches (cf. Table 1): 26–30.

#	RMTL	Serenity	Simmonds1	Simmonds2	Spass-Meter
1a*	security policy monitoring	security & dependability checking	safety & liveness checking	behavioral analysis	performance/resource monitoring & runtime adaptation
1b	maintenance	maintenance	operation	operation	design, impl., testing & op.
1c	mobile apps	distributed systems	Web services	business processes	domain & platform independent
1d	mobile (Android) OS	distributed systems	SOA	BPEL processes	any
1e*	instrumentation of Android OS kernel	instrumentation using SERENITY framework	sequence diagrams	BPEL process def., service conditions & correctness properties	instrumentation
1f*	detected security policy violations	detected security or dependability violations	evaluation results	formal model/monitors	summary, snapshots & API
1g	runs within the system	runs within the system	runs in parallel	runs within the system	runs within the system
2a	Android users	engineers	engineers	engineers	engineers & ops
2b	detect security violations	detect security & dependability violations	unclear	behavioral monitoring	analyze and improve performance
2c	formal background	formal background	modeling skills	formal background	prog. exp. & using annotations or XML
2d	none	templates for XML-based EC-Assertion rules	Graphical sequence diagram tool	monitoring automata	EASy-Producer editor
2e	none	unclear	faulty execution traces	unclear	textual report/API
3a*	RMTL, extension of MTL	XML-based EC-Assertion language based on event calculus	Temporal Logic Property Patterns	WSCoL & deterministic finite automata	IVML, inspired by OCL in EASy-Producer
3b*	custom impl.	custom impl.	WebSphere	WebSphere	custom impl.
3c*	event ordering & occurrence	event occurrence, event order & data checks	event occurrence & order	event occurrence & order	data checks
3d	blocking of application calls	no	no	WSCoL: pre- & post-conditions	via annotations & API
3e	call from Android app intercepted	events	start-up of a process instance	service invocations	via API or via EASy-Producer
3f	severity	no	no	no	typed annotations
3g	via policy spec. and hooks	no	no	no	via EASy-Producer
4a*	examples	performance experiments	examples	examples	experiments, scientific eval. with industry Spass-Meter, EASy
4b	LogicDroid	no download but researchers around	unclear	unclear	

Table 8
Comparison framework: application to 32 approaches (cf. Table 1): 31–32.

#	UFO/FORMAN	YAWL
1a*	software monitoring & runtime verification	task performance checking
1b	maintenance	design & execution
1c	reactive systems / real-time	general-purpose
1d	component-based systems	business processes
1e*	behavioral properties	annotated YAWL model
1f*	monitoring code	runtime evaluation results
1g	unspecified	runs in parallel
2a	engineers	process analysts/owners
2b	check program behavior	assess risks of faults
2c	writing assertions	YAWL
2d	none	YAWL editor extension with Sensor editor & risk templates
2e	unclear	unclear
3a*	assertion language	DSL inspired by Java (syntax) & SQL (semantics)
3b*	generated C code	custom implementation
3c*	event occurrence, event order & data checks	event occurrence, event order, data checks & fuzzy checks
3d	access to system properties	simple arithmetic + functions
3e	events or time-based	periodic or event-based
3f	no	no
3g	no	no
4a*	examples	performance analysis & usability evaluation
4b	N/A	in next release of YAWL

References

Aktug, I., Dam, M., Gurov, D., 2008. Provably Correct Runtime Monitoring. In: Cuellar, J., Maibaum, T., Sere, K. (Eds.), FM 2008: Formal Methods. In: LNCS, vol.5014. Springer, pp. 262–277.

Auguston, M., Giammarco, K., Baldwin, W.C., Crump, J., Farah-Stapleton, M., 2015. Modeling and verifying business processes with monterey phoenix. Procedia Comput. Sci. 44, 345–353.

Baouab, A., Perrin, O., Godart, C., 2012. An Optimized Derivation of Event Queries to Monitor Choreography Violations. In: Liu, C., Ludwig, H., Toumani, F., Yu, Q.(Eds.), Service-Oriented Computing. In: LNCS, vol.7636. Springer, pp. 222–236.

Baresi, L., Guinea, S., 2013. Event-based multi-level service monitoring. In: 20th IEEE Int'l Conf. on Web Services. IEEE, pp. 83–90.

Bauer, A., Leucker, M., Schallhart, C., 2006. Monitoring of Real-time Properties. In: FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science. Springer, pp. 260–272.

Calinescu, R., Ghezzi, C., Kwiatkowska, M.Z., Mirandola, R., 2012. Self-adaptive software needs quantitative verification at runtime. Commun. ACM 55 (9), 69–77.

Conforti, R., Rosa, M.L., Fortino, G., ter Hofstede, A.H., Recker, J., Adams, M., 2013. Real-time risk monitoring in business processes: a sensor-based approach. J. Syst. Softw. 86 (11), 2939–2965.

Contreras, A., Mahbub, K., 2014. MORPED: Monitor rules for proactive error detection based on run-time and historical data. In: Fifth Int'l Conf. on the Applications of Digital Information and Web Technologies. IEEE, pp. 28–35.

Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., Wasowski, A., 2012. Cool fea-tures and tough decisions: A comparison of variability modeling approaches. In: 6th Int'l Workshop on Variability Modelling of Software-Intensive Systems. ACM, pp. 173–182.

Delgado, N., Gates, A.Q., Roach, S., 2004. A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Trans. Softw. Eng. 30 (12), 859–872.

Dwyer, M., Avrunin, G., Corbett, J., 1999. Patterns in property specifications for finite-state verification. In: Int'l Conf. on Software Engineering. IEEE, pp. 411–420.

Ehlers, J., Hasselbring, W., 2011. A Self-adaptive Monitoring Framework for Component-based Software Systems. In: Crnkovic, I., Gruhn, V., Book, M. (Eds.), Software Architecture. In: LNCS, vol.6903. Springer, pp. 278–286.

Faymonville, P., Finkbeiner, B., Peled, D., 2014. Monitoring Parametric Temporal Logic. In: McMillan, K.L., Rival, X. (Eds.), Verification, Model Checking, and Abstract Interpretation. In: LNCS, vol.8318. Springer, pp. 357–375.

Ganter, B., Wille, R., 2012. Formal concept analysis: Mathematical foundations. Springer.

Ghezzi, C., Mocci, A., Sangiorgio, M., 2012. Runtime monitoring of component changes with Spy@Runtime. In: 34th Int'l Conf. on Software Engineering. IEEE, pp. 1403–1406.

Gunadi, H., Tiu, A., 2014. Efficient Runtime Monitoring with Metric Temporal Logic: A Case Study in the Android Operating System. In: Jones, C., Pihlajasaari, P., Sun, J. (Eds.), FM 2014: Formal Methods. In: LNCS, vol.8442. Springer, pp. 296–311.

Holmes, T., Mulo, E., Zdun, U., Dustdar, S., 2011. Model-aware Monitoring of SOAs for Compliance. In: Service Engineering. Springer, pp. 117–136.

Inzinger, C., Satzger, B., Hummer, W., Dustdar, S. , 2013. Specification and Deployment of Distributed Monitoring and Adaptation Infrastructures. In: Ghose, A., Zhu, H., Yu, Q., Delis, A., Sheng, Q., Perrin, O., Wang, J., Wang, Y. (Eds.), Service-Oriented Computing - ICSOC 2012 Workshops. In: LNCS, vol.7759. Springer, pp. 167–178.

Jeffery, C., Auguston, M., Underwood, S., 2004. Towards Fully Automatic Execution Monitoring. In: Wirsing, M., Knapp, A., Balsamo, S. (Eds.), Radical Innovations of Software and Systems Engineering in the Future. In: LNCS, vol.2941. Springer, pp. 204–218.

Kim, M., Kannan, S., Lee, I., Sokolsky, O., Viswanathan, M., 2001. Java-Mac: a runtime assurance tool for java programs. Electron Notes Theor. Comput. Sci. 55 (2), 218–235.

Kitchenham, B.A., 2007. Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical Report. Version 2.3, EBSE Technical Report, UK.

Kumar Tripathy, A., Patra, M., 2010. An event based, non-intrusive monitoringframework for web service based systems. In: Int'l Conf. on Computer Information Systems and Industrial Management Applications. IEEE, pp. 547–552.

Luckham, D.C., 2011. Event processing for business: Organizing the real-time enterprise. Wiley.

Maiden, N., 2013. Monitoring our requirements. IEEE Software 30 (1), 16–17.

Mansouri-Samani, M., Sloman, M., 1993. Monitoring distributed systems. IEEE Netw. 7 (6), 20–30.

Matinlassi, M., 2004. Comparison of software product line architecture design methods: COPA, FAST, FORM, KobrA and QADA. In: 26th Int'l Conf. on Software Engineering. IEEE, pp. 127–136.

Medvidovic, N., Taylor, R., 20 0 0. A classificiation and comparison framework for software architecture description languages. IEEE Trans. on Softw. Eng. 26 (1), 70–93.

Montali, M., Maggi, F.M., Chesani, F., Mello, P., van der Aalst, W.M.P., 2014. Monitoring business constraints with the event calculus. ACM Trans. Intell. Syst. Technol. 5 (1), 17:1–17:30.

Muccini, H., Polini, A., Ricci, F., Bertolino, A., 2007. Monitoring Architectural Properties in Dynamic Component-based Systems. In: Component-Based Software Engineering, LNCS 4608. Springer, pp. 124–139.

Paschke, A., 2005. RBSLA a declarative rule-based service level agreement language based on RuleML. In: Int'l Conf. on Computational Intelligence for Modelling, Control and Automation and Intelligent Agents, Web Technologies and Internet Commerce. IEEE, pp. 308–314.

Phan, H., Avrunin, G.S., Clarke, L.A., 2008. Considering the Exceptional: Incorporating Exceptions into Property Specifications, 1003. Department of CS, Univ. of Massachusetts, Amherst, MA.

Poppe, O., Giessl, S., Rundensteiner, E.A., Bry, F., 2013. The HIT Model: Workflow-aware Event Stream Monitoring. In: Hameurlain, A., Küng, J., Wagner, R., Amann, B., Lamarre, P. (Eds.), Trans. on Large-Scale Data- and Knowledge-Cen- tered Systems XI. In: LNCS, vol.8290. Springer, pp. 26–50.

Rabiser, R., Grünbacher, P., Lehofer, M., 2012. A qualitative study on user guidance capabilities in product configuration tools. In: 27th IEEE/ACM Int'l Conf. on Au- tomated Software Engineering. ACM, pp. 110–119.

Ramirez, A.J., Cheng, B.H., 2011. Automatic Derivation of Utility Functions for Monitoring Software Requirements. In: Whittle, J., Clark, T., Kühne, T. (Eds.), Model Driven Engineering Languages and Systems. In: LNCS, vol.6981. Springer, pp. 501–516.

Robinson, W.N., 2006. A requirements monitoring framework for enterprise systems. Requirements Eng. 11 (1), 17–41.

Seracini, F., Menarini, M., Krueger, I., Baresi, L., Guinea, S., Quattrocchi, G., 2014. A comprehensive resource management solution for web-based systems. In: 11th Int'l Conf. on Autonomic Computing. USENIX, pp. 233–239.

Simmonds, J., Gan, Y., Chechik, M., Nejati, S., O'Farrell, B., Litani, E., Waterhouse, J., 2009. Runtime monitoring of web service conversations. IEEE Trans. Serv. Comp. 2 (3), 223–244.

Spanoudakis, G., Mahbub, K., 2004. Requirements monitoring for service-based systems: towards a framework based on event calculus. In: 19th Int'l Conf. on Automated Software Engineering. ACM, pp. 379–384.

van Hoorn, A., Waller, J., Hasselbring, W., 2012. Kieker: A framework for application performance monitoring and dynamic software analysis. In: 3rd ACM/SPEC Int'l Conf. on Performance Engineering. ACM, pp. 247–248.

Vierhauser, M., Rabiser, R., Grünbacher, P., 2016. Requirements monitoring frameworks: a systematic review. Inf. Softw. Technol. 80, 89–106.

Vierhauser, M., Rabiser, R., Grünbacher, P., Egyed, A., 2015. Developing a DSL-based approach for event-based monitoring of systems of systems: Experiences and lessons learned. In: 30th IEEE/ACM Int'l Conf. on Automated Software Engineering. ACM, pp. 715–725.

Vierhauser, M., Rabiser, R., Grünbacher, P., Seyerlehner, K., Wallner, S., Zeisel, H., 2016. Reminds: a flexible runtime monitoring framework for systems of systems. J. Syst. Softw. 112, 123–136.

Viswanathan, M., Kim, M., 2005. Foundations for the Run-time Monitoring of Reactive Systems: Fundamentals of the MaC Language. In: Theoretical Aspects of Computing. Springer, pp. 543–556.

Völz, M., Koldehofe, B., Rothermel, K., 2011. Supporting strong reliability for distributed complex event processing systems. In: 13th Int'l Conf. on High Performance Computing & Communication. IEEE, pp. 477–486.

Rick Rabiser is a Senior Researcher at the Christian Doppler Lab for Monitoring and Evolution of Very-Large-Scale Software Systems at Johannes Kepler University Linz, Austria. He holds a Ph.D. in Business Informatics and the Habilitation in Practical Computer Science, both from Johannes Kepler University Linz. Refer to http://mevss.jku.at/rabiser for more detailed information.

Sam Guinea is an Assistant Professor at the DEpendable Evolvable Pervasive Software Engineering (DEEP-SE) group at Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy. He studied Computer Engineering and holds a Ph.D. in Information Engineering from Politecnico di Milano. Refer to http://home.deib.polimi.it/guinea/ for more detailed information.

Michael Vierhauser is a Researcher at the Christian Doppler Laboratory for Monitoring and Evolution of Very-Large-Scale Software Systems at Johannes Kepler University Linz, Austria. He received a Ph.D. in Computer Science in 2016 from the Johannes Kepler University Linz. Refer to http://mevss.jku.at for more detailed information.

Luciano Baresi is a Full Professor at the DEpendable Evolvable Pervasive Software Engineering (DEEP-SE) group at Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy. He holds a Ph.D. in Information Engineering from Politecnico di Milano. Refer to http://home.deib.polimi.it/baresi/ for more detailed information.

Paul Grünbacher is an Associate Professor at Johannes Kepler University Linz, Austria and the head of the Christian Doppler Laboratory for Monitoring and Evolution of Very-Large-Scale Software Systems. He studied Business Informatics and holds a Ph.D. from the Johannes Kepler University Linz. Refer to http://mevss.jku.at for more detailed information.