Automated Software Engineering (2018) 25:875–915
https://doi.org/10.1007/s10515-018-0241-x

Developing and evolving a DSL-based approach for runtime monitoring of systems of systems

Rick Rabiser[1] · Jürgen Thanhofer-Pilisch[1] · Michael Vierhauser[2] · Paul Grünbacher[3] · Alexander Egyed[3]

Received: 13 December 2017 / Accepted: 27 June 2018 / Published online: 5 July 2018

Abstract
Complex software-intensive systems are often described as systems of systems (SoS) due to their heterogeneous architectural elements. As SoS behavior is often only understandable during operation, runtime monitoring is needed to detect deviations from requirements. Today, while diverse monitoring approaches exist, most do not provide what is needed to monitor SoS, e.g., support for dynamically defining and deploying diverse checks across multiple systems. In this paper we report on our experiences of developing, applying, and evolving an approach for monitoring an SoS in the domain of industrial automation software, that is based on a domain-specific language (DSL). We first describe our initial approach to dynamically define and check constraints in SoS at runtime and then motivate and describe its evolution based on requirements elicited in an industry collaboration project. We furthermore describe solutions we have developed to support the evolution of our approach, i.e., a code generation approach and a framework to automate testing the DSL after changes. We evaluate the expressiveness and scalability of our new DSL-based approach using an industrial SoS. We also discuss lessons we learned. Our results show that while developing a DSL-based approach is a good solution to support industrial users, one must prepare the approach for evolution, by making it extensible and adaptable to future scenarios. Particularly, support for automated (re-)generation of tools and code after changes and automated testing are essential.

✉ Rick Rabiser
rick.rabiser@jku.at

Michael Vierhauser
mvierhau@nd.edu

Paul Grünbacher
paul.gruenbacher@jku.at

1 Christian Doppler Lab. MEVSS, Johannes Kepler University Linz, Linz, Austria

2 Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, USA

3 Institute for Software Systems Engineering, Johannes Kepler University Linz, Linz, Austria

Keywords Systems of systems · Requirements monitoring · Constraint checking · Domain-specific languages · DSL evolution

1 Introduction

Many software-intensive systems today are systems of systems comprising heterogeneous and interrelated architectural elements. Common properties of SoS include decentralized control, support for multiple platforms, inherently volatile and conflicting requirements, and independent and continuous evolution of its heterogeneous parts (Maier 1998; Nielsen et al. 2015). As SoS behavior is often only fully understandable during operation, system testing is not sufficient to determine compliance with requirements. Instead, the behavior of the constituent systems and their interactions need to be continuously monitored and checked during operation to detect and analyze deviations. Checks include the occurrence, order, and timing of runtime events (temporal behavior) (Dwyer et al. 1999), the interaction of systems (structural behavior), or properties of runtime data (data checks) and probabilistic properties (e.g., related to performance) (Autili et al. 2015).

Different research communities have been developing runtime monitoring approaches for various kinds of systems and diverse types of checks. Examples include requirements monitoring (Vierhauser et al. 2016a; Maiden 2013; Robinson 2006), monitoring of architectural properties (Muccini et al. 2007), complex event processing (Völz et al. 2011), runtime verification (Calinescu et al. 2012; Ghezzi et al. 2012), monitoring of probabilistic properties (Cailliau and van Lamsweerde 2017; Autili et al. 2015; Sammapun et al. 2005; Zhang et al. 2011), and resource monitoring (van Hoorn et al. 2012; Eichelberger and Schmid 2014), to name but a few. The expected runtime behavior is often expressed formally using temporal logic (Cailliau and van Lamsweerde 2017; Viswanathan and Kim 2005; Chen et al. 2004; Gunadi and Tiu 2014; Bauer et al. 2006). Furthermore, domain-specific languages are employed to facilitate the definition of constraints (Paschke 2005; Robinson 2008; Baresi and Guinea 2013), which are then checked based on events and data collected from systems at runtime, e.g., via probes instrumenting systems (Mansouri-Samani and Sloman 1993; Eichelberger and Schmid 2014).

Runtime monitoring in SoS, however, is particularly challenging as, e.g., temporal, structural, and data constraints need to be checked for a high number of events collected from heterogeneous systems. Furthermore, many SoS are cyber-physical systems (Bures et al. 2014) that need to run in 24/7-mode for weeks or even months without interruption. Constraints thus need to be defined and deployed dynamically, e.g., in response to issues discovered during operation, and continuously checked to ensure live and instant feedback on requirements violations to users. Many existing runtime monitoring approaches, however, are restricted to particular technologies or types of constraints or are limited to offline analysis of event traces (Vierhauser et al. 2016a).

To address these challenges, we have been developing a *domain-specific constraint language aimed at industrial end users*, who often lack deep programming skills, to ease the definition of various types of constraints in SoS. Our DSL-based

approach supports monitoring the behavior of industrial SoS at runtime, i.e., checking the order, occurrence and timing of events monitored at runtime as well as checking data attached to events, e.g., to ensure certain ranges. We developed the language iteratively to address industrial needs, i.e., to cover all types of constraints elicited in multiple workshops together with industrial users. We described our experiences with the development of the first version of the constraint language in an earlier publication (Vierhauser et al. 2015).

As described in this publication, we decided to only provide features for actual use cases, to keep the language simple and to meet our industry partner's expectations (YAGNI—"you aren't gonna need it"). When continuing work with our industry partner, to support *new use cases*, the language needed to be evolved. For example, it became necessary to express optional events and negations and to allow more complex data analyses. Also, industrial *users requested additional language features*, e.g., event-based constraint evaluation criteria to evaluate a constraint when a certain event occurs, or composite constraints to allow combining different (types of) constraints. Furthermore, we conducted a usability study (Rabiser et al. 2016) with industrial end users that revealed *usability flaws* to be addressed by modifying the constraint DSL and tool support for writing constraints and interpreting constraint violations. A detailed *comparison with other approaches* in the field (Rabiser et al. 2017) also motivated us to further improve our approach.

In this paper we describe our experiences of developing the initial DSL-based approach (v.1) and then evolving it (leading to v.2). We evaluate the expressiveness and scalability of our new approach and discuss DSL evolution in general, i.e., the types of changes and their impacts, and the lessons we learned from evolving our initial approach. Particularly, we discuss how we improved our approach regarding extensibility and maintainability in future evolution scenarios.

Specifically, we claim the following contributions:

- We motivate our work with an industrial case of monitoring a metallurgical plant automation system, an example of a large-scale industrial SoS. We describe an industrial scenario and discuss the challenges for constraint checking in SoS at runtime. We describe the DSL-based approach we developed to address these challenges, which allows to dynamically define and incrementally (Egyed 2006; Vierhauser et al. 2010) check constraints at runtime to support SoS monitoring. We have described parts of this contribution in our earlier publication (Vierhauser et al. 2015).
- We motivate and describe the evolution of this initial approach to address new (user) requirements. We describe the (types of) necessary changes and discuss their impacts.
- We present a new version of our DSL-based approach and also describe a solution for automated testing we developed to better support the (future) evolution of our approach. We present an evaluation based on the use of our approach in an industrial SoS, specifically, we assess its expressiveness and scalability, also in comparison to the first version of our approach.
- We conclude with a general discussion of the evolution of our DSL-based approach. We revisit and extend the lessons we learned from developing our initial approach

to reflect what we learned during the evolution of our approach. We also discuss related work, particularly also regarding DSL evolution.

2 Industrial scenario and challenges

Our industry partner—Primetals Technologies, a joint venture of Siemens and Mitsubishi Heavy Industries—is one of the world's leading engineering and plant-building companies in the iron, steel, and aluminum industries. The company provides machinery, hardware, software, and automation systems for steel producers around the globe. We use the example of a plant automation system of systems (PAS) developed and maintained by Primetals Technologies to illustrate the runtime monitoring and constraint checking challenges. The PAS automates, optimizes, and tracks different stages of the metallurgical production process. It comprises systems for process automation of melting iron ore and raw materials to produce iron, refining liquid iron and other materials to produce steel, and casting liquid steel into solid steel slabs. These independently developed automation systems for iron, steel, and continuous casting (cf. Fig. 1) size up to several million LoC. The systems have heterogeneous architectures, they have been developed using diverse technologies, and they frequently interact, e.g., when exchanging data controlling the metallurgical production process.

Although the different software systems in the PAS are engineered independently, there are manifold dependencies in the metallurgical process that need to be considered when planning their joint operation. For instance, liquid iron is needed for producing liquid steel, which is then input to casting solid steel slabs. The PAS is further connected to legacy and third-party systems leading to additional complexity. Furthermore, there are dependencies between components within particular systems. For instance, a component optimizing the arrangement of steel slabs on a strand in the caster—to minimize scrap and to ensure steel quality—relies on information provided by other components such as material tracking.

Besides such requirements, which cross-cut different systems or components, there are also requirements affecting particular components. For example, the component handling the selection of material from a silo and the subsequent transportation on a conveyor belt, has to ensure that events happen in a specific sequence and within a certain time frame to guarantee the uninterrupted and continuous flow of material. Although PAS requirements and their dependencies are carefully managed during development, it is crucial to monitor them after deployment to detect inaccurate and erroneous behavior at runtime. This is particularly important after upgrading components, a frequent case when modernizing existing plants.

2.1 Industrial runtime monitoring scenario

In the following we describe a typical scenario for runtime monitoring based on an earlier qualitative study with developers and engineers of our industry partner (Vierhauser et al. 2012). The scenario shows that constraints need to be defined and deployed dynamically and checked continuously at runtime. For the scenario we assume that

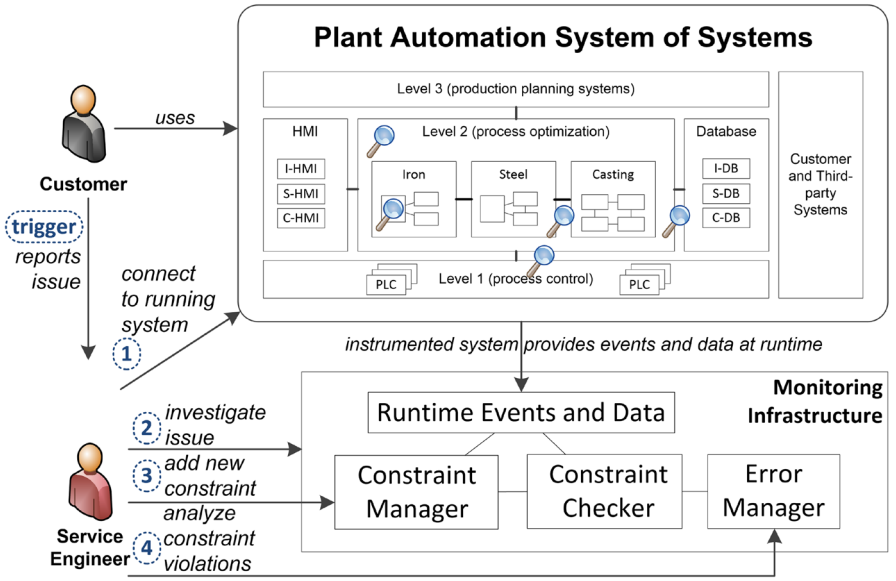

Fig. 1 Industrial scenario for runtime monitoring of SoS

the PAS is running at the customer's site and an infrastructure for runtime monitoring is set up to collect events and data about the running systems. The scenario shown in Fig. 1 starts with a customer report describing a deviation from the expected system behavior: after upgrading several parts, the initialization of the casting system does not complete within the expected time frame, thus delaying the process of casting liquid steel to solid steel slabs. Due to the interplay of several systems there are many possible reasons for this behavior and both hardware and software issues might have caused the problem.

(1) The service engineer reviews the incoming customer report and remotely connects to the customer's PAS to investigate the issue by checking the running systems. (2) The service engineer uses the monitoring infrastructure to retrieve more details about the state of the PAS, e.g., by analyzing recorded event data to reveal the origin of the reported problem or by reviewing violations that might have been recorded in the past. (3) If necessary, the service engineer adds new constraints to the monitoring infrastructure. Depending on the type of problem different types of constraints may be necessary, for instance, to check temporal properties of events or data ranges. In the case of the delayed initialization of the caster the service engineer adds a new constraint checking the execution of the initialization steps within a time frame of 60 s. The constraint is immediately activated in the monitoring infrastructure to detect deviations from the specified behavior in the running PAS. (4) The service engineer is notified by the monitoring infrastructure as soon as the initialization phase is delayed again. Violations of the new constraint are detected and can be instantly reviewed by the engineer.

2.2 Challenges

Several challenges for defining and checking constraints can be derived from our industrial case:

(C1) Constraint diversity Different types of constraints are needed to monitor the industrial SoS. This covers global invariants and range checks of variables across the SoS, temporal constraints on the occurrence and expected order of events, or architectural rules constraining the allowed interactions of components. Constraints are further required to measure properties such as performance or resource consumption. Regarding runtime monitoring, a wide variety of approaches exist that focus on particular types of constraints [see Table 1 and (Rabiser et al. 2017)]. While these approaches focus on specific types of constraints, there is a lack of unified approaches covering multiple types of constraints, as needed in our SoS runtime monitoring context.

(C2) Dynamic definition and runtime management of constraints Constraints are typically not defined once before the system is put into operation but dynamically when needed. For instance, as our industrial scenario showed, engineers may need to define additional constraints on the fly when investigating an issue reported by a customer. Furthermore, SoS evolve continuously and are configured to address specific requirements. The constraints thus do not remain stable but need to co-evolve with the system to adapt to certain monitoring scenarios (Rabiser et al. 2015). Many existing approaches do not support the dynamic definition and management of constraints at runtime (see Table 1).

(C3) End-user definition of constraints End-user support for writing constraints becomes a primary issue, as a runtime monitoring environment and its constraint checking mechanism will be used in practice by both engineers and maintenance personnel. While many existing constraint languages address the needs of software developers (see Table 1), writing new or maintaining existing constraints is much harder for users without a deep programming background.

While diverse constraint languages have been developed, e.g., for requirements-level monitoring techniques (van Lamsweerde 2009; Robinson 2008; Skene and Emmerich 2005; Whittle et al. 2010), UML-based monitoring approaches (Zhang et al. 2008; Kiviluoma et al. 2006), or formal runtime verification techniques (Viswanathan and Kim 2005; Chen et al. 2004; Gunadi and Tiu 2014; Bauer et al. 2006), many do not address the described challenges. Most existing languages support specific types of constraints. For example, some approaches focus on monitoring probabilistic properties (Cailliau and van Lamsweerde 2017; Autili et al. 2015), while others emphasize temporal properties, or support aggregating and checking data. Additionally, most existing approaches have been developed for a particular application domain—e.g., service-based systems (Zhang et al. 2008) or business processes (Luckham 2011)—and would be hard to apply in other areas. Furthermore, many existing constraint languages are deemed inconvenient by industrial end users as they require deep understanding of formal concepts or lack tool support.

As summarized in Table 1, many existing constraint languages thus do not fully support all three challenges in SoS monitoring described above. For instance, Spanoudakis et al. (2009) present SERENITY, a framework for monitoring secu-

Table 1 Comparing monitoring approaches/DSLs regarding their support for three key challenges in SoS monitoring described in Sect. 2

Approach	Description	C1: diversity	C2: runtime mgmt	C3: end-user support
Spanoudakis et al. (2009)	EC-Assertions, a temporal formal language based on the Event Calculus, to detect violations within streams of runtime events	*No* (focus on security and dependability (S&D) properties; event occurrence)	*Yes* (checks on S&D properties can be added at dynamically)	*Partly* (EVEREST tool, but no user study)
Viswanathan and Kim (2005)	Two constraint languages for MaC: PEDL for writing low-level specifications and MEDL for defining safety requirements	*Yes* (can adapt to different implementation languages and specification formalisms)	*No* (statically defined)	*No* (Java prototype of checker, no user study)
Baresi and Guinea (2013)	Custom-developed DSL mlCCL (Multi-layer Collection and Constraint Language) for service-based systems	*Yes* (constraints for analyzing events and capabilities to collect KPIs/aggregate data)	*Yes* (constraints can be added at runtime)	*Partly* (Ecoware tool, but no user studies)
Montali et al. (2014)	Declare, a declarative business process constraint language, based on the Event Calculus	*Partly* (focus on process execution, but diverse types of rules)	*No* (statically specified)	*Partly* (ProM tool, but no user study)
Bertolino et al. (2011)	Custom-developed property meta-model allows the definition of quantitative and qualitative properties	*Yes* (abstract properties, descriptive properties, and perspective properties)	*No* (property model statically specified and then transformed to input for CEP engine)	*Partly* (GLIMPSE framework, but no user study)
Aktug et al. (2008)	Security specification language ConSpec to describe automata for security requirements	*No* (focus on security properties)	*No* (class files need to be annotated)	*No* (formal validation, no user study)

Table 1 continued

Approach	Description	C1: diversity	C2: runtime mgmt	C3: end-user support
Cailliau and van Lamsweerde (2017)	Probabilistic linear temporal logic (PLTL) for specifying goals and obstacles for system monitoring and (self-)adaptation	*Partly* (focus on probabilistic properties, but diverse goals and obstacles)	*Yes* (goal/obstacle model can be updated at runtime)	*No* (benchmarks, no user study)
Zhang et al. (2011)	Formal property specification language Probabilistic Timed Property Sequence Chart (PTPSC)	*Yes* (diverse properties can be expressed, e.g., temporal, probabilistic)	*No* (statically specified)	*Partly* (WS-PSC tool, no user study)
Sammapun et al. (2005)	RT-MaC for quantitative and probabilistic property specs	*Yes* (diverse properties can be expressed)	*No* (statically specified)	*No* (overhead eval., no user study)

rity and dependability properties. Monitoring rules are expressed as EC-Assertions, a temporal formal language based on the Event Calculus. EC-Assertions are used to detect violations within streams of runtime events, which are are provided by different distributed sources. The language is XML-based and provides language support for event occurrences such as *Happens*, *HoldsAt*, or *Terminates*. Viswanathan and Kim (2005) developed two constraint languages for their MaC (Monitoring and Checking) architecture: PEDL (Primitive Event Definition Language) for writing low-level specifications and MEDL (Meta Event Definition Language) for defining safety requirements. This separation allows to adapt to different implementation languages and specification formalisms (e.g., Java-MaC (Kim et al. 2004) for Java programs). Baresi and Guinea (2013) present mlCCL, the Multi-layer Collection and Constraint Language part of the ECoWare framework for monitoring service-based systems. Besides constraints for analyzing events, the language also provides capabilities for defining how to collect messages or key performance indicators and how to aggregate data from multiple objects. Montali et al. (2014) present Declare, a declarative business process constraint language part of the Mobucon EC monitoring framework. The language is based on the Event Calculus and allows defining sets of rules that must be satisfied in order to correctly execute a given process. They distinguish between four different types of constraints: *existence*, *choice*, *relation* and *negation*. Bertolino et al. (2011) present a property-driven approach for runtime monitoring. A property metamodel allows the definition of quantitative and qualitative properties. The approach further distinguishes between abstract properties for generic declarations, descriptive properties describing guaranteed properties, and perspective properties describing system requirements. The approach uses the GLIMPSE framework for monitoring distributed systems and checking the properties at runtime. Aktug et al. (2008) present an approach for monitoring security properties. They use a security specification language called ConSpec to describe automata for security requirements. Existing work also often uses *temporal logic* to support monitoring software systems. Several authors have shown the expressiveness and usefulness of such formalisms (Chen et al. 2004; Gunadi and Tiu 2014; Bauer et al. 2006). Cailliau and van Lamsweerde (2017) present an approach for system monitoring and (self-)adaptation that relies on specifying goals and obstacles using probabilistic linear temporal logic. Based on the model, their approach can decide when system adaptation should be triggered and then adapt the system on the fly. They evaluated their approach using an ambulance dispatching system. Zhang et al. (2011) defined a formal property specification language called Probabilistic Timed Property Sequence Chart (PTPSC), a probabilistic and timed extension of Property Sequence Charts (PSC). The authors describe a formal grammar-based syntax they use to generate a probabilistic monitor combining timed Büchi automata and a sequential statistical hypothesis test process. They also present a tool based on PTPSC, the WS-PSC Monitor, that supports runtime monitoring. Sammapun et al. (2005) present RT-MaC, an extension of MaC (Kim et al. 2004) with the capability to verify timeliness and reliability correctness by providing quantitative and probabilistic property specifications. This is achieved by introducing time-bound temporal operators and probabilistic operators.

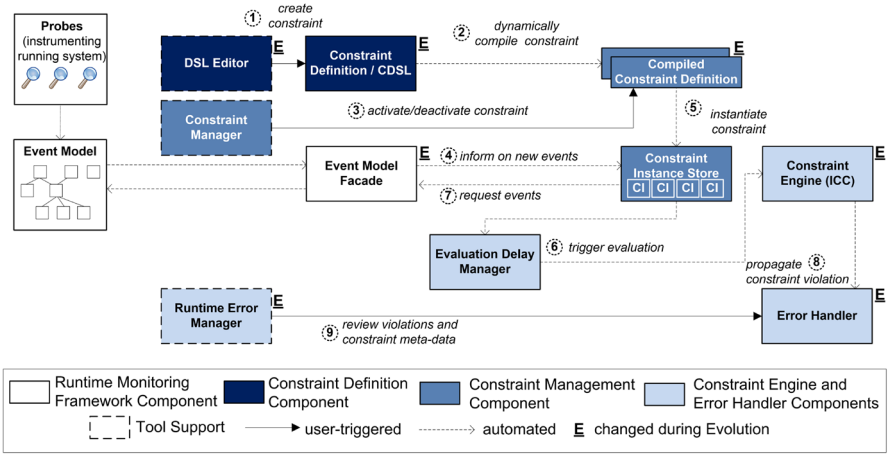

Fig. 2 The main components of our DSL-based approach. The numbers indicate how the different components interact in a typical scenario. The components marked with an 'E' had to be adapted when evolving the DSL-based approach (cf. Sects. 4, 5)

3 DSL-based approach v.1

We developed a DSL-based approach (Vierhauser et al. 2015)—based on an existing incremental checker (Egyed 2006; Vierhauser et al. 2010)—to address the challenges discussed above. We ensure that violations of requirements can be reported instantaneously to users monitoring an SoS. The approach further supports the dynamic definition and deployment of constraints at runtime—i.e., constraints can be added or modified without stopping the checker or the monitored systems—and provides tool support for end-users.

We first describe the key assumptions of our approach and then provide details on its main components (cf. Fig. 2), i.e., our DSL and the editor supporting industrial users in the definition of constraints using this DSL (*Constraint DSL; CDSL*), the components supporting compiling and instantiating constraints at runtime to generate the input for the constraint engine (*Constraint Management*), and the components supporting constraint checking and error handling (*Constraint Engine and Error Handler*). These components of our DSL-based approach are typical components of a DSL-based approach (Van Deursen et al. 2000; Spinellis 2001).

3.1 Key assumptions

We assume a stream of events (cf. Fig. 3) observed at runtime by a monitoring infrastructure. In our monitoring approach these events are collected in an event model (Vierhauser et al. 2016b) also managing arbitrary data attached to specific events. The event model enables checking across system boundaries by linking events provided by probes instrumenting different systems. It also provides the foundation for tools visualizing behavior, persisting event logs, or checking constraints on events.

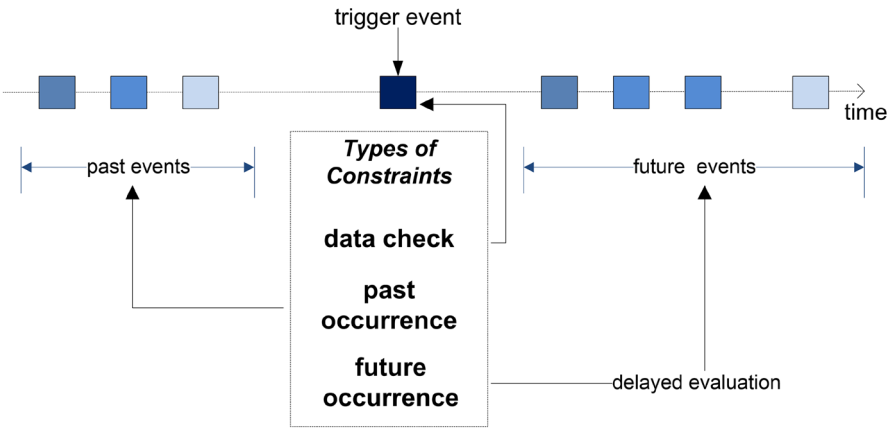

Fig. 3 Event stream and constraint types of our DSL

Events are distinguished by their type. Types can be arranged hierarchically to reflect the location in the system structure. For example, in the PAS Caster system the event type "Optimizer.optimize_START" is a child of the "Optimizer" type, which again is a child of the "Caster" type. Furthermore, events are distinguished by their source, i.e., the probes instrumenting systems or components in the SoS. Each event has a time stamp and arbitrary data can be attached, e.g., primitive data types, objects, as well as arrays and lists of data types or objects. Event data can also carry performance information about the instrumented system.

3.2 Constraint DSL

We conducted a series of workshops and interviews with engineers and project managers of our industry partner to elicit requirements for a constraint language for SoS runtime monitoring, based on concepts from existing constraint languages. Emphasizing usefulness and practical applicability, we developed the CDSL allowing engineers to specify temporal, structural, and data constraints on events and data.

Each constraint written in the CDSL starts with a description of the *trigger* event activating the evaluation of the constraint (cf. Fig. 3). The trigger specification is followed by a *condition* statement. Conditions can be defined to check the *past occurrence* of an event before the trigger event; the *future occurrence* of a (sequence of) event(s) after the trigger event; or *data* attached to the trigger event. Arbitrary or specific orders of sequences can be defined.

Conditions on the past or future occurrence of events are temporal constraints for checking restrictions regarding the occurrence or sequential order of events, i.e., they are pre- or post-conditions on these events. For *simple order restrictions*, an event of a certain type must occur before or after an event of a specific type, e.g., one event of type B must occur after any event of type A (required sequence [A, B]). For *hard time limits*, the occurrence of an event of a certain type is required within a certain time,

Listing 1 Examples of three constraints from the PAS.

```
//future occurrence constraint checking a sequence of events with a hard time limit
trigger = if event "ControlAdapter.requestOptimization" occurs
condition = events
      "Optimizer.optimize_START",
      "Optimizer.fetchData",
      "Optimizer.caculateFINISHED",
      "Optimizer.retrieveOptimizationResult",
      "Cutting.forwardOptimizationResult"
occur consecutively within 10 seconds.

//past occurrence constraint with a hard time limit
trigger = if event "Tundish.carLockedInCastPosition" occurs
condition = event "Tundish.ladleArrived"
has occurred in the last 500 milliseconds.

//data constraint checking the data attached to the event System.discInfo
trigger = if event "System.discInfo" occurs
condition = data("discData","Drives/FreeDiscPercentage")>20.
```

e.g., an event of type B must occur within a maximum time of 5 s after an event of type A has occurred.

Data constraints check certain items contained in runtime data objects attached to an event. For numeric values boundary checks can be defined, e.g., to ensure a data item is within a certain range. For character sequences checks for equality can be defined. Data conditions in our DSL can also contain functions, e.g., to count the number of elements in a list, to check whether a list of data objects contains a certain item, or to calculate the maximum, minimum, or average of a set of values. It is also possible to combine past and future occurrence checks with data checks, e.g., to determine if a certain event occurred with the attached data fulfilling a particular condition.

Listing 1 shows three examples from the PAS: a constraint checking the future occurrence of an event sequence in a particular order including a hard time limit for the optimization system cycle; a constraint on the past occurrence of a particular event with a hard time limit checking that the ladle has arrived before starting casting; and a data constraint checking that free disc space is larger than 20%.

We also developed a *DSL Editor* that provides user support for writing constraints in our DSL, including meta-data such as a description, a custom error message, or a severity class. We refer to a constraint defined in the DSL Editor as a *Constraint Definition* (cf. Fig. 2). We employed the Java-based frameworks Xtext and Xtend (http://www.eclipse.org/Xtext) for developing the CDSL and end-user guidance in the editor. These frameworks allow adding new language constructs in a rather straightforward manner as the DSL and the transformation steps are treated separately and automation is provided, e.g., for supporting syntax highlighting and auto completion in the editor as shown in Fig. 4.

3.3 Constraint management

The *Constraint Manager* tool (cf. Fig. 4) allows activating, deactivating, and modifying (groups of) constraints.

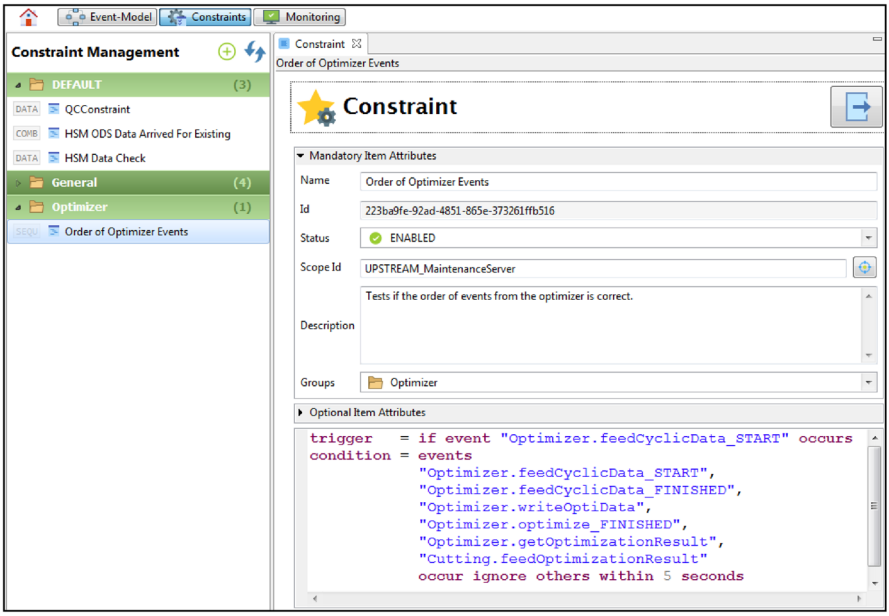

Fig. 4 Tool support for defining and managing constraints (constraint shown in screenshot already based on v.2 of our DSL-based approach)

As runtime monitoring presumes the continuous (re-)evaluation of constraints, dynamic constraint management and an incremental evaluation strategy are advisable to ensure fast feedback to users in case of violations. We therefore decided to use an incremental consistency checker (ICC) developed in our previous work on consistency checking of design models (Egyed 2006; Vierhauser et al. 2010, 2012).

As soon as the Constraint Definition is transmitted to the checking server, it is thus on the fly compiled to executable Java code, i.e., to a *Compiled Constraint Definition* to make it usable by the ICC. The *Constraint Instance Store* (cf. Fig. 2) is responsible for maintaining and instantiating compiled constraints. An active constraint is not instantiated permanently but only if an event occurs that matches the trigger event defined for this constraint. Each created constraint instance is completely self-contained and can be evaluated independently. This ensures that even for a high number of incoming events only selected constraints are instantiated and checked.

Please note that while the constraints written in our DSL are currently translated to the underlying language of the ICC, using Xtext and Xtend we could also translate the constraints to a different target language and thus use a different engine for checking constraints. This also supports the "source-to-source" transformation pattern introduced by Spinellis (2001).

3.4 Constraint engine and error handler

Constraint checking relies on the events provided by the REMINDS Runtime Monitoring Framework (Vierhauser et al. 2016b), which uses an *Event Model* to abstract

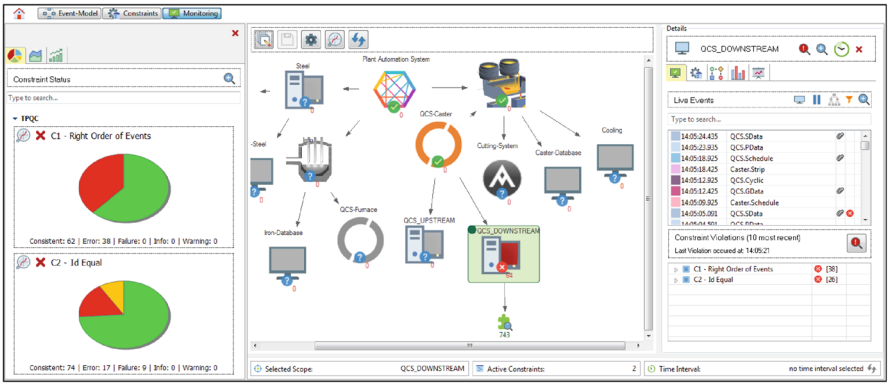

Fig. 5 Tool support for monitoring and reviewing constraint violations

from different systems and technologies as described under key assumptions above. An *Event Model Facade* (cf. Fig. 2) allows the ICC—typically running on a separate server—to register to the event model as listener and to connect with the runtime monitoring infrastructure (i.e., the event model). The facade is informed about new incoming events of certain types from specific sources.

Depending on the type of constraint, a constraint instance may need to be evaluated immediately after instantiation (e.g., when checking data attached to the triggering event or the past occurrence of events), or it may need to be postponed until future events arrive. This task is handled by the *Evaluation Delay Manager*, which extends the original ICC and adds capabilities for intercepting constraint evaluation requests. It delays their evaluation and executes them only when required events arrive or as soon as a specified timeout occurs.

If a constraint can be evaluated it is passed to the ICC's actual *Constraint Engine*. The constraint is evaluated by executing its code, thereby accessing the event model via the Event Model Facade to retrieve events or data if necessary. If the constraint instance evaluates without errors (the constraint condition evaluates to true), the instance is immediately destroyed and removed from the engine. If the constraint instance is violated and evaluates to false, further information on the violation is forwarded to the *Error Handler*, i.e., a text explaining the violation.

The *Runtime Error Manager* tool displays this information as soon as it becomes available to allow users reviewing occurring violations immediately. Violations can also be persisted for later inspection. The Runtime Error Manager also provides a graphical overview of all components of the SoS and their current state (cf. Fig. 5). It allows reviewing constraint violations related to a specific component, events responsible for constraint violations, and additional information provided by the constraints.

4 Evolving the approach

In our earlier work (Vierhauser et al. 2015), we demonstrated the expressiveness and scalability of our approach for a set of constraints relevant to cover the requirements of

one of our industry partner's systems. However, when monitoring other systems, new requirements emerged. For example, it became necessary to express optional events and negations and to allow more complex data analyses. Furthermore, users requested additional language features and tool support such as event-based evaluation criteria and composite constraints to combine different (types of) constraints. In addition, we conducted a usability study (Rabiser et al. 2016) with industrial end users that revealed some usability flaws to be addressed by modifying the CDSL and tool support for writing constraints and interpreting constraint violations.

We thus decided to evolve our approach. We aimed to address new requirements and usability flaws. We also wanted to increase the *maintainability and extensibility*—e.g., reengineer the code generation component to better support future changes—and facilitate better *testability and reliability*, i.e., introduce an automated testing framework and enable regression testing.

Extending our approach to address these issues required changes of different components on different levels of granularity (cf. Fig. 2) with different impacts. While some changes only required to perform minor modifications of the DSL's grammar, other changes required to change the code generation part or even adapting the underlying constraint engine.

4.1 Levels of changes

We distinguish six levels of changes, which comprise the affected component and the type and impact of changes, to ease discussing the requirements. The impacts on the affected components and the necessary changes increase with each level. Please note that in our discussion of the different levels of changes we distinguish between *adaptation*, i.e., adding a new capability or simple renaming activities, and *reengineering*, i.e., major changes that do also affect the components' internal structure. If a change to the DSL-based approach requires changes at different levels, the highest level defines the overall level of the change. Therefore, a change categorized as reengineering can also include adaptations. We define the impacts of the different change levels as a result of the estimated effort for requirements analysis plus the estimated effort for development plus the estimated effort for testing. We mapped this result to a four-point impact scale: low, medium, high, and very high.

Table 2 summarizes the six change levels. If the DSL's grammar, the code generation, and the constraint engine are properly separated and their interfaces are clearly defined, then a higher level change does not necessarily involve all lower level changes. For instance, if a level 4 (code generation reengineering) change is required, it might be the case that level 1 and 2 (syntax) changes are not necessary assuming that there is a clear separation between code generation and the DSL grammar itself. If the grammar or the generation code is not well structured, however, most lower level changes will eventually result in higher level changes as they then require substantial structural changes in addition to adaptation.

Table 2 Levels of changes for evolving our DSL-based approach and their impact

#: Name	Explanation	Impact
L1: DSL syntax adaptation	The DSL grammar is adapted, e.g., rename an existing (non-)terminal symbol or add a new symbol	*Minor*—renames might require adapting existing constraints; new symbols are not used so far
L2: DSL syntax reengineering	The DSL grammar is reengineered, i.e., non-terminal symbols are removed or restructured	*Medium*—the original behavior of existing constraints might be affected and dependent components need to be reengineered
L3: Code generation adaptation	The code readable by the engine, generated from the compiled constraints, is adapted internally, e.g., by adding new code in reaction to DSL changes	*Minor to Medium*—as the interfaces to other components are clearly defined, adaptations mostly affect only the result of the constraint evaluation rather than other components. Therefore, they will usually not require further adaptations
L4: Code generation reengineering	Here, the interfaces of the code generation part are modified, e.g., in reaction to a bigger DSL syntax refactoring	*High*—due to the high complexity of the code generation part, reengineering can be rather expensive, particularly if interfaces are modified. Can also lead to reengineering of the constraint engine
L5: Constraint engine adaptation	The constraint engine providing an evaluation result based on the code generated from the compiled constraints is adapted, e.g., to add new constraint types or additional error information	*Medium to High*—adding new evaluation capabilities based on existing ones can be very complex if a completely new type of behavior/constraint has been introduced
L6: Constraint engine reengineering	The constraint engine is reengineered, e.g., to improve evaluation performance. Adaptations made on levels 1–5 can eventually lead to such a reengineering, if, e.g., adding a new capability to the DSL required adapting the engine and this adaptation led to a decrease in performance	*Very High*—due to the high complexity of reengineering the code processing the constraint evaluations, which are executed in parallel, also the effort for testing this component is very high, particularly because some errors might only occur in very specific test setups

4.2 Evolution requirements (required changes)

Based on the identified shortcomings of the first version of our DSL-based approach and user requirements resulting from the usefulness study (Rabiser et al. 2016), we derived requirements (required changes) for evolving the approach, which are summarized in Table 3. For each requirement we provide a description, a rationale, and list the impact (cf. Table 3).

Table 3 Requirements for the evolution of our DSL-based approach

#: Name	Description	Rationale	Impact
R1: Composite constraints	The approach should support combining and aggregating the evaluation of multiple constraints	Monitoring the execution of a single process in a system can require several independent, yet related constraints, e.g., if there are several valid event sequences	L5 (2+4+5)
R2: Multiple data checks	It should be possible to perform data checks on multiple events in a constraint	This is frequently required in practice, e.g., if in some event sequences all events must have the same id stored in their data objects	L4 (2+4)
R3: Cross-event data access	Data should be accessible across events in a constraint	It can be necessary to check if all events of a constraint belong to the same action, which can be checked, e.g., with an id in the events' data.	L4 (2+4)
R4: Multiple data checks for one event	The approach should allow multiple data checks on one event	It is sometimes necessary to perform multiple checks on one event, e.g., to check both, the type and the value of an analysis event	L4 (1+4)
R5: Event-based evaluation criteria	It should be possible to evaluate constraints based on the occurrence of events	Practical examples demonstrated that it should be possible to force a sequence constraint to be evaluated no later than when the start event of the next event sequence occurs	L3 (1+3)
R6: Multiple evaluation criteria	The approach should support defining multiple evaluation criteria in one constraint	For instance, sometimes constraints shall be evaluated within a certain time span and until another event occurs	L3 (1+3)
R7: Flexible event sequences	The CDSL should provide additional capabilities for defining event sequences such as negation, optional events and arbitrary orders	It is not always possible or desired to check an event sequence just through a list of events that have to occur as sometimes multiple valid event sequences are possible	L4 (1+4)
R8: Violation details	Constraint violations should provide more detailed information on the violation cause	Violation causes should not only be available as strings, but together with details such as what (missing) events caused the violation (Vierhauser et al. 2017)	L5 (4+5)
R9: Grammar consistency	The DSL's grammar should be consistent	The grammar of the first version of the CDSL allows inconsistent event declarations as identified in our usability study (Rabiser et al. 2016), which can have a negative effect on the ease of writing constraints	L3 (2+3)

5 DSL-based approach v.2

Several of the requirements described in Table 3 required major changes to the first version of the DSL syntax and the code generation components [cf. components and steps (1) and (2) in Fig. 2]. For the constraint engine and the UI parts it was fortunately sufficient to limit the changes to minor adaptations. We, however, also improved the capabilities of our DSL-based approach to support future evolution. This included the development of a testing framework to automate and parallelize testing of constraints, which we then utilized for regression testing throughout the evolution of our DSL-based approach.

In this section we first describe the changes we made to the constraint DSL to address the requirements described above. Then we describe the changes to the code generation approach and present the testing framework. Implementation was done by one Master's student, assisted by one PhD student, over the time of one and a half years. Frequent meetings with senior researchers (co-authors of this paper) and the industry partner Primetals Technologies helped to guide the implementation process.

5.1 Changes made in the constraint DSL

As a starting point for designing the new DSL grammar, we analyzed the first version of the grammar for possible weaknesses regarding the introduction of the new required functionality and potential future evolution. Based on these findings, we derived all necessary grammatical changes and adapted the grammar. As it was essential to preserve the original intentions of the first version, i.e., making the language comprehensible and easy to use for industrial users, we decided to limit the reengineering of the grammar mainly to restructuring its non-terminal symbols to increase the grammar's modularity and extensibility. The changes visible to the user were kept as minimal as possible. The new version of the grammar, especially its new structure, focuses on supporting future changes without requiring structural changes or complex refactorings. The full grammar is listed in the "Appendix", examples for constraints in the new grammar are presented in Listings 2 and 3 as well as in Table 4.

Specifically, we performed the following changes to the grammar:

Composite Constraints (cf. R1): we introduced a new high-level concept that allows to combine different (types of) constraints. A constraint can now either be a composite constraint combining several constraints or a single constraint. Constraints in a composite constraint can be combined with AND or OR and their individual evaluation contributes to the overall evaluation. It is also possible to nest them, meaning to create composite constraints of composite constraints.

Multiple data checks (cf. R2): a data check can now be added to every event defined in a constraint, which is particularly relevant for events in sequences (allowing to check different data items attached to different events in a sequence).

Cross-event data access (cf. R3): events can now be mapped to a user-defined key (expressed by: *event_type as "key"*). The data of these events can then be accessed via this shortcut ("key".data("item-type","item-name")).

Listing 2 Example constraint checking a sequence of events in a cyclic optimization process.

```
trigger = if event "Optimization_Start" as "trigger" occurs
condition = events
"Fetch_Data"
    where data("meta–data", "run–id") == "trigger".data("meta–data", "run–id")
    AND data("meta–data", "cut–point") == "trigger".data("meta–data", "cut–point"),
? "Optimize_Run"
    where data("meta–data", "run–id") == "trigger".data("meta–data", "run–id")
    AND data("meta–data", "cut–point") == "trigger".data("meta–data", "cut–point"),
"Optimization_End"
    where data("meta–data", "run–id") == "trigger".data("meta–data", "run–id")
    AND data("meta–data", "cut–point") == "trigger".data("meta–data", "cut–point"),
occur
ignore others
until "Optimization_Start" occurs
```

Multiple data checks for one event (cf. R4): multiple data checks for one event can now be combined using Boolean operators.

Event-based evaluation criteria (cf. R5): in the first version only time-based evaluation criteria were possible. With the new version we have introduced event-based evaluation criteria, i.e., using "until 'Event' occurs" and "since 'Event' occurred" the user can define that a specified event (sequence) has to occur before or after this event.

Multiple evaluation criteria (cf. R5 and R6): we replaced the evaluation criterion "consecutively" (the defined sequence events must occur one after the other without interruption by other events) from the first version with the term "ignore others" (the defined sequence may be interrupted by other events) as the standard behavior of a (sequence) constraint in our industrial use cases has been consecutively. Additionally to "ignore others" an event sequence can now also be defined to have to occur "in any order".

Flexible event sequences (cf. R7): events (in sequences) can now also be negated (must not occur) or made optional (may or may not occur).

Violation details (cf. R8): this requirement did not require any changes in the DSL. Instead, to realize this requirement we adapted the internal structure of the Constraint Engine component so that the violation cause (e.g., event missing, additional event, wrong event) together with other information is now captured and forwarded as an object to the Error Handler component. Further details regarding the diagnosis of violations are discussed in another publication (Vierhauser et al. 2017).

Grammar consistency (cf. R9): We unified the way and the order of defining events, data checks and scope checks throughout the language (cf. the grammar in the "Appendix" vs. the grammar in our earlier publication Vierhauser et al. (2015)).

The examples in Listings 2 and 3 show constraints that use the new language features. The examples are taken from the plant automation system of systems described in Sect. 2.1. Further examples can be found in the evaluation section in Table 4.

Listing 2 shows a constraint from the Casting system that checks the correct order of events in a cyclic optimization process. After the triggering event that indicates the start of an optimization run, the events "Fetch_Data", "Optimize_Run" and "Optimization_End" have to occur until the next optimization cycle starts. The event "Optimize_Run" (indicated by '?') is optional since it may not occur if no optimiza-

Listing 3 Example constraint checking multiple data items of an iron tap analysis.

```
trigger = if event "Tap_End" occurs
condition = events
"Tap_Analysis" where data("analysis", "type") == "Metal" AND data("analysis","temperature") > 1690,
"Tap_Analysis" where data("analysis", "type") == "Slag" AND within( data("analysis","thickness"), 10%,
        2.5)
occur in any order within 5 minutes
```

tion is necessary. Furthermore, other events that may occur in between the defined sequence will not be considered (indicated by "ignore others") and the constraint also checks that the "run-id" and the "cut-point" data values of all events are equal (multiple data checks).

The second example constraint (cf. Listing 3) is from the Iron system. It expects two analyses to occur after the end of an iron tap from a blast furnace. The events can occur in any order, however, one of them must be a metal analysis measuring a temperature higher than 1690 degrees Celsius and the other one must report a slag thickness that is around (10%) of 2.5 centimeters. Also, both events need to occur within a time frame of five minutes.

5.2 New code generation approach

We did not redevelop the actual constraint engine, but we redesigned the code generation component to support the new language features described above and particularly to ease future evolution. The code generation component takes a constraint written in our DSL as input and produces the output required by the constraint engine, i.e., Java code for the ICC.

Our new approach has a modular architecture—Fig. 6 compares version 1 and 2 of the code generation architectures. In the first version, we used the Xtext framework to parse the constraints and generate the Java code needed for the constraint evaluation by combining several predefined code snippets (cf. all the snippets shown in Fig. 6 in the Code Generation (Xtext/Xtend) part of version 1). This approach has two major drawbacks regarding maintainability and extensibility: (i) generating code by assembling code snippets is error-prone due to the high number of possible variations of how the code snippets can be combined. Furthermore, it is very difficult to understand and analyze and makes extending the code generation part with new functionality difficult. (ii) The generated code is just code that is inserted into one monolithic method and compiled at runtime by the constraint engine. Debugging this dynamically compiled code is cumbersome because it is scattered across multiple files.

In our second version of the DSL-based approach, we thus developed a hybrid approach aiming at getting the best trade-off between directly processing the constraint information in the Xtext parser and using external helper classes. We use the code-snippet assembly approach for the first parsing steps, i.e., until the type of the constraint (e.g., a future constraint) is defined (cf. the remaining snippets shown in Fig. 6 in the Code Generation (Xtext/Xtend) part of version 2). Then, a model (bottom right in the figure) is instantiated with the rest of the variable information contained in the

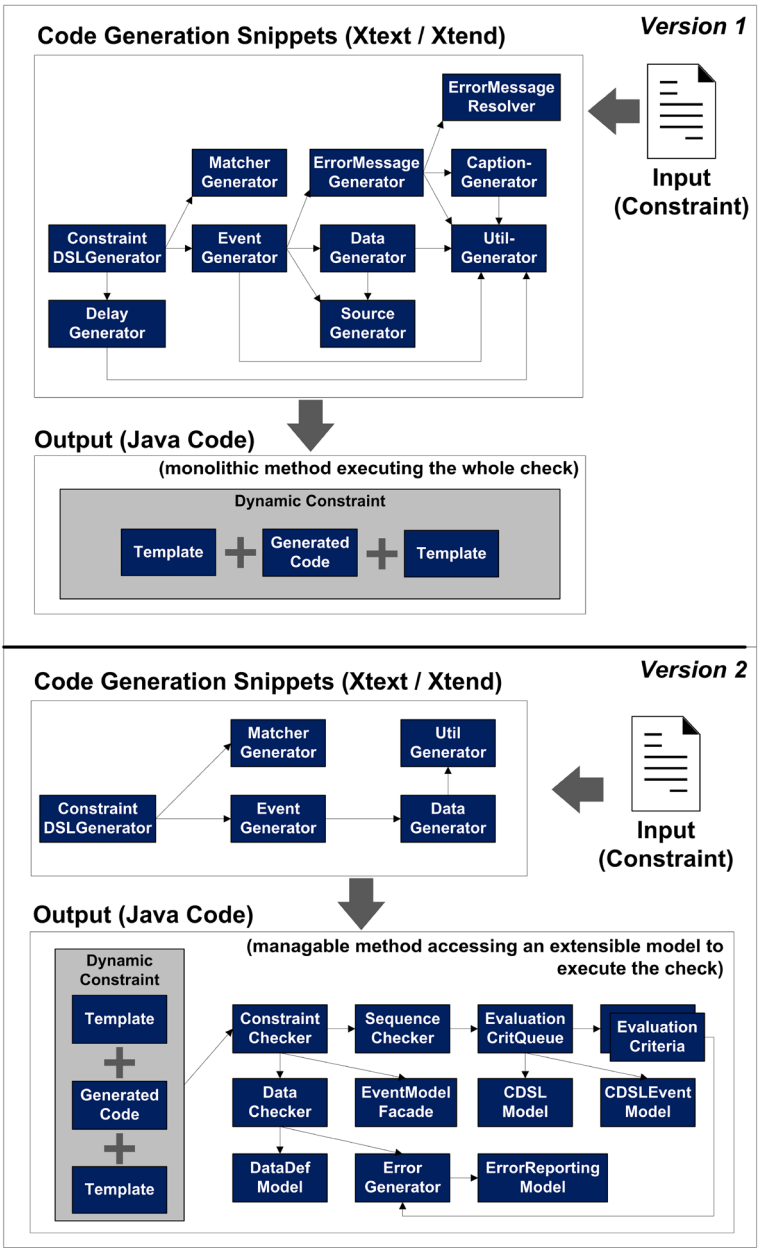

Fig. 6 Comparison of the code generation architectures of the first and the second version of our DSL-based approach

constraint (e.g., the defined events to occur or the defined evaluation criteria), which is then handed over to a helper class that processes them and evaluates the constraint accordingly.

5.3 Testing framework

Every change to the DSL requires thorough testing to ensure that new functionality is implemented correctly and to ensure that the change does not affect the existing functionality and constraints. During the evolution of our DSL-based approach, we quickly reached a point where manual testing became infeasible: testing just one of the several hundred possible constraint variations took us about two to five minutes due to the need to (re-)start several components of our architecture, e.g., run the monitored software and probes, run the monitoring server, and then manually check whether a particular constraint evaluates correctly. We thus decided to automate and parallelize the testing of our DSL-based approach.

Specifically, we developed a testing framework that allows defining test cases for different (types of) constraints and automates repetitive tasks involved in testing our DSL-based approach such as producing events simulating monitored software, deploying and checking constraints, and comparing the expected behavior as defined in the test case with the actually observed behavior. The framework we developed is based on JUnit and executes test cases in parallel wherever possible. We use a model of the CDSL for creating constraints in the test cases, which allows changing the grammar without the need to also adapt the test cases (if not removing features).

A CDSL test case (Listing 4 shows an example) is written in Java and comprises a unique ID, a constraint to be checked, an event sequence to be produced and an expected result, which can be a successful evaluation of the constraint or an expected violation. This data is then used to deploy the specific constraint, to evaluate it against the defined event sequence, and to compare the result of the evaluation with the defined expected result.

The example test case in Listing 4 defines a constraint that checks if the events `Iron.SlagStart` and `Iron.TapEnd` follow within a time period of 3 s after a triggering event `Iron.TapStart`. The expected event sequence thus is `Iron.TapStart`, `Iron.SlagStart` and `Iron.TapEnd`. The event generator component is used to create a predefined event sequence `IRON_TS_TE_SS_2sec`, which fires the events `Iron.TapStart`, `Iron.TapEnd` and `Iron.SlagStart` within 2 s in exactly this order. Therefore, the expected sequence should be violated as the second event to occur has a different event type as specified in the constraint. Thus, the test case defines an expected violation of type `Wrong Event Type`, containing information about the events which caused the violation (cf. requirement 8). When the test is executed, the defined constraint is activated and instantiated. Afterwards, the defined event sequence is produced, the constraint engine recognizes the triggering event, evaluates the constraint and throws the violation, which is then compared to the expected violation. If this comparison succeeds, the test case is reported as passed on the command line interface.

Listing 4 Example of a CDSL test case.

```
@Test
public void MF_within3sec_SeqOrderViol() throws Exception {
    String constrID = "MF_within3sec_SeqOrderViol";
    IConstraintGenerator cGen=getConstraintGenerator(constrID);

    //create constraint (equal to):
    //trigger = if event "Iron.TapStart" occurs
    //condition = events "Iron.SlagStart", "Iron.TapEnd" occur within 3 seconds
    ITestConstraint constraint=cGen.generateConstraint(ITestConstraint.CONSTRAINT_TYPE.FUTURE)
        .setTriggerEvent(cGen.generateConstraintEvent("Iron.TapStart"))
        .setEventsToCheck(cGen.generateConstraintEventSequence("Iron.SlagStart", "Iron.TapEnd"))
        .setWithinTime("3_seconds");

    // IMEEvents for simulation run
    List<IMEEvent> fireEventSequence;
    synchronized (ParallelViolationTestRunner.SEQUENCE_CREATION_LOCK) {
        fireEventSequence = cGen.getIMEEventGenerator().createEventSequence(IEventSequenceGenerator
                .Sequence_IDs.IRON_TS_TE_SS_2SEC);
    }

    //add expected errors
    Set<IExpectedError> expected=new HashSet<>();
    expected.add(ExpectedViolationFactory.createViolation(fireEventSequence.get(0),
            fireEventSequence.get(1), ViolationErrorType.WRONG_TYPE));
    addExpectedViolations(constrID, expected);

    //execute the test
    executeTestCase(constrID,constraint.getConstraintCode(),fireEventSequence);
}
```

6 Evaluation

Based on the evaluation of the first version of our DSL-based approach (Vierhauser et al. 2015) we aim to show that the new version of our approach still fulfills the existing requirements and also supports the new ones. Specifically, our evaluation investigates the expressiveness and the scalability and performance of the new version. A side goal also was to demonstrate that the changes between version 1 and 2 did not introduce any performance issues. To ensure a realistic and comparable evaluation, we again monitored a simulated version of Primetals Technologies' SoS.

Specifically, we investigate the following research questions in our evaluation:

RQ1—Is the DSL sufficiently expressive to allow its use in a real-world industrial SoS? In Vierhauser et al. (2015) we showed that the first version of the approach is sufficiently expressive for a real-world use case and constraints could be written representing all requirements to be monitored at that point in time. Since no functionality was removed during the development of version 2, we can safely conclude, that this is still true. With our evaluation of RQ1 we thus here want to show that the new version of the CDSL is expressive enough to represent the new requirements (cf. Sect. 4).

RQ2—Does the constraint checking approach scale to industrial needs in the context of a real-world SoS? To evaluate the scalability of our approach, we monitored a simulated version of Primetals Technologies' PAS to measure the performance of our approach. We updated the constraints used in the previous evaluation to the newer version of the CDSL and added additional constraints containing new functionality

to show the impact of the new features on the performance of constraint evaluation. The previous evaluation of RQ2 (Vierhauser et al. 2015) started only with a limited set of active constraints and activated two separate sets of constraints afterwards to show the impact of dynamically adding constraints at runtime. The main part of the evaluation however, was a 6-h run with all constraints activated. Since there were no changes made to the components responsible for (dynamically) adding, modifying and removing constraints at runtime, we here focus on evaluating and comparing the 6-h run with all constraints set active ("old" ones and new ones). We also discuss the relation of constraint complexity and measured evaluation (execution) time.

6.1 RQ1: DSL expressiveness

To evaluate the expressiveness of our CDSL, we conducted five 2-h workshops as well as several additional meetings with seven system experts to identify requirements that have to be monitored at runtime in different systems or among system borders of Primetals Technologies' PAS. Altogether, about 40 requirements were found. At least one constraint for each of them was developed. These requirements motivated the implementation of the three constraint types (past occurrence, future occurrence and data constraints) used in CDSL version 1. Since CDSL version 1 was capable of representing all of these constraints, we demonstrated in Vierhauser et al. (2015) that its expressiveness is sufficient for using it in a real-world industrial SoS.

Future meetings and workshops, after the release of the first version of our approach, however, revealed that additional capabilities were necessary for the DSL-based approach which we implemented as described above (cf. Sect. 5). Specifically, there were six requirements regarding new DSL features and one requirement demanding a new constraint type (composite constraints). Table 4 shows seven constraints from the PAS demonstrating these requirements. Please note that we have simplified and/or obfuscated some of the real events and data item names due to non-disclosure agreements. Based on these positive experiences and the feedback we received from our industry partner, we conclude that the new version is sufficiently expressive to use it with a real-world SoS.

6.2 RQ2: scalability

In Vierhauser et al. (2015), we have shown that our approach is sufficiently scalable to be applied to a real-world SoS. This evaluation was done by performing a 10-h run with 13 constraints. To show the effect of dynamically adding and removing constraints, the evaluation started with only five active constraints. After an hour, four additional constraints were added and finally, after another hour, four more constraints were added. These 13 constraints were then active for a total of 6-h. Afterwards, constraints were deactivated again stepwise in the last 2 h of the 10-h run.

In this evaluation we focus on the part affected by the evolution of our approach. Since no changes were made to the component capable of dynamically adding and removing constraints at runtime, we omitted this part in our evaluation and repeated the 6-h run with all constraints being active to try to compare it against the first evaluation.

Table 4 Seven constraint examples demonstrating the expressiveness of our DSL

Req.	Constraint Example
R1	Composite constraints

```
//at least one of the constraints "A", "B" or "C" must not lead to a
      violation
composite constraint = constraint "A" or constraint "B" or constraint "C"
```

R2	Multiple data checks

```
//sequence check with data checks on multiple events in the sequence
trigger = if event "Start_Analysis" occurs
condition = events
      "Temp_Analysis" where data("general", "temperature") > 1250,
      "Quality_Analysis" where data("general", "qualityRating") > 0.98,
      "End_Analysis"
occur within 10 seconds
```

R3	Cross-event data access

```
//checks data values against other, previously occurred, events
trigger = if event "LabAnalysis1" as "Point1" occurs
condition = events
      "LabAnalysis2" as "Point2" where data("general", "temperature") < "
            Point1".data("general", "temperature"),
      "LabAnalysis3" where data("general", "temperature") < "Point2".data(
            "general", "temperature")
occur ignore others within 1 hour
```

R4	Multiple data checks for one event

```
//checks that more than five percent and more than two gigabytes free
      storage are on the disk
trigger = if event "Disc_Analyzed" occurs
condition = data("disk","freePercentage") > 5 and data("disk","freeSpace"
      ) > 2048
```

R5	Event-based evaluation criteria

```
//checks that the defined sequence occurs before the next start event of
      this sequence occurs again
trigger = if event "Start_Analysis" occurs
condition = events
      "Temp_Analysis",
      "Quality_Analysis",
      "End_Analysis"
occur until event "Start_Analysis" occurs
```

R6	Multiple evaluation criteria

```
//checks that the defined sequence occurs within eight hours and before
      the next start event of this sequence occurs again
trigger = if event "ProductionStarted" occurs
condition = events
      "QualityAnalyzed",
      "ProductionEnded"
occur until event "ProductionStarted" occurs within 8 hours
```

R7	Flexible event sequences

```
//defines an event sequence that can occur in any order, but the second
      event must not occur and the third event may occur
trigger = if event "Tap_Start" occurs
condition = events
     !"Slag_Start",
     ?"Tap_Analysis",
      "Tap_End"
occur in any order within 3 minutes
```

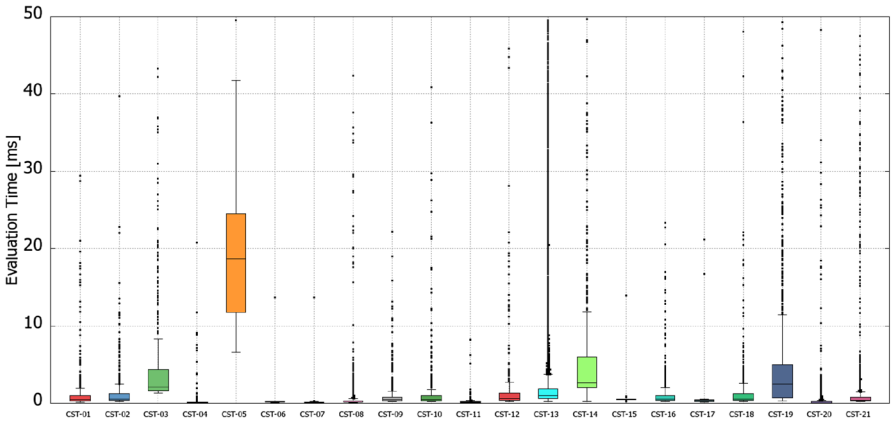

Fig. 7 Boxplots of median evaluation times (in ms) for all constraints

We used a simulator of a new version of the PAS for our current evaluation since some of the requirements for version 2 of the DSL-based approach were the result of newly introduced features and functionality in the PAS.

Another difference to the previous evaluation is the used system to execute the evaluation which now is a standard Desktop machine with an Intel(R) Core(TM) i5 CPU @3.20 GHz 16 GB RAM running Windows 10 64-Bit. In order to retain comparability with the previous evaluation, which used a PC with slightly lower computational power, we re-ran the 6-h run using only the existing CDSL version 1 constraints (without recompiling them to CDSL version 2) and then compared these results to our new ones.

To show the scalability and applicability of our new DSL features, we developed eight new constraints (CST-14–CST-21, cf. Table 5). Our evaluation thus in total comprises 8 new constraints and the 13 constraints of the previous evaluation, which were recompiled using the new version of our approach. The semantics of the 13 constraints is identical, however, the code that is executed by the constraints is now the reengineered version 2 code of our approach. We measured the number of constraint checks and the (median) evaluation time, i.e., the time required for executing the method evaluating the constraint.

Table 5 shows the used constraints and the newly evaluated median evaluation time of the 13 previously existing CDSL version 1 constraints wherever a re-evaluation was possible with the new PAS simulator. It was not possible to re-evaluate seven of the 13 old constraints since some events, their data and some systems in the new version of the PAS have changed which made it impossible to re-evaluate them without adapting and recompiling them. Thus we only present comparable MET v1 values for constraints CST-3, CST-5, and CST-8-11. For constraints CST-1, CST-2, CST-4, CST-6, CST-7, CST-12, and CST-13 we present the original results from Vierhauser et al. (2015) in square brackets as they cannot directly be compared due to the changed machine. Please note that details about the constraints are not shown due to non-disclosure agreements. We, however, explicitly indicate their complexity in Table 5 using the following levels 1 to 5, with 5 being the highest complexity:

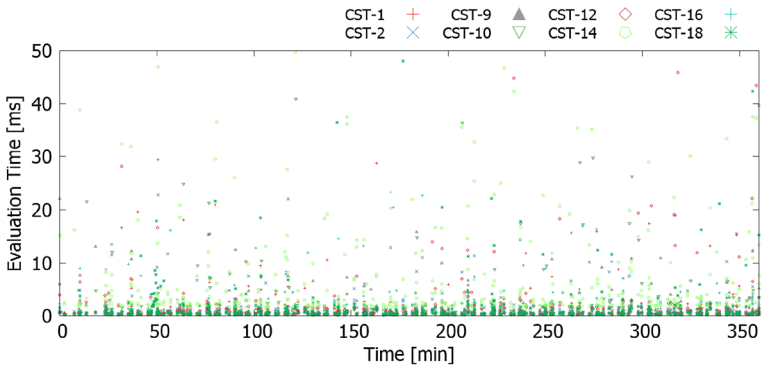

Fig. 8 Distribution of median evaluation times for future occurrence constraints (CST-01, CST-02, CST-09, CST-10, CST-12, CST-14, CST-16, CST-18)

1. A single data check in the constraint's condition (cf. Listing 1 (third constraint)).
2. Multiple data checks in the constraint's condition or simple event occurrence/timing check (cf. Listing 1 (second constraint)).
3. Complex event occurrence/order/timing with multiple events in the constraint's condition or complex data check (cf. R2 in Table 4).
4. Combined event occurrence/order/timing and data check (cf. R3 in Table 4).
5. Multiple combined types of constraints (cf. R1 in Table 4).

To reduce the impact of the operating system and other programs and services running in the background, we performed three evaluation runs and present their median results in this section. During the 6-h simulation run of the PAS, 103,761 events were monitored and 73,705 constraint checks were performed resulting in an average of 205 constraint checks per minute. Table 5 shows the median evaluation time of each constraint (in ms) and the number of checks performed per constraint. The memory consumption remains between 18.09 MB (lower quartile) and 27.7 MB (upper quartile).

Figure 7 depicts box plots of the median evaluation time for all 21 constraints. Figures 8, 9 and 10 depict the distribution of median evaluation times for the three different types of constraints (future occurrence, past occurrence and data constraints). The constraints with the highest MET all have a complexity level of 3 or higher. Figure 11 shows box plots in which the 21 constraints' median evaluation times have been grouped by complexity level. One can nicely see the increase of the median evaluation times with complexity level, while the overall performance still stays very good (the highest medians are just slightly above 3 ms). Complexity level 5 is an exception, which can be explained by the fact that in our evaluation on this level two composite constraints combine sub-constraints with a very good performance. We discuss details on evaluating the performance of the composite constraints below.

To sum up the results of the evaluation, the MET of all constraints ranges from 0.08 ms (CST-04) to 16.7 ms (CST-05). Every CDSL version 2 constraint that could be compared to its version 1 pendant, except CST-05, became slightly faster. CST-05 performed worse than in CDSL version 1 because in the reengineering we decided to

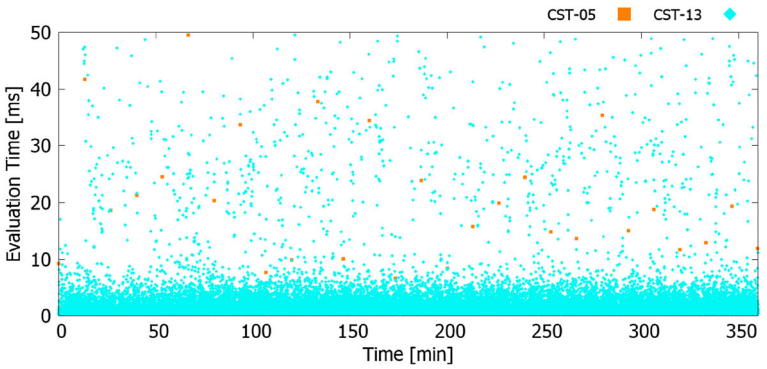

Fig. 9 Distribution of median evaluation times for past occurrence constraints (CST-05, CST-13)

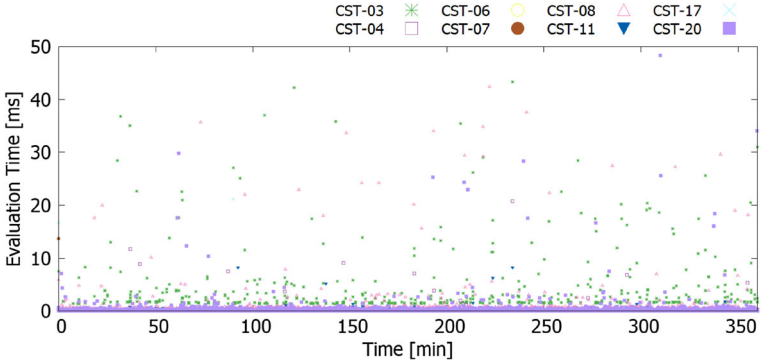

Fig. 10 Distribution of median evaluation times for data constraints (CST-03, CST-04, CST-06, CST-07, CST-08, CST-11, CST-17, CST-20)

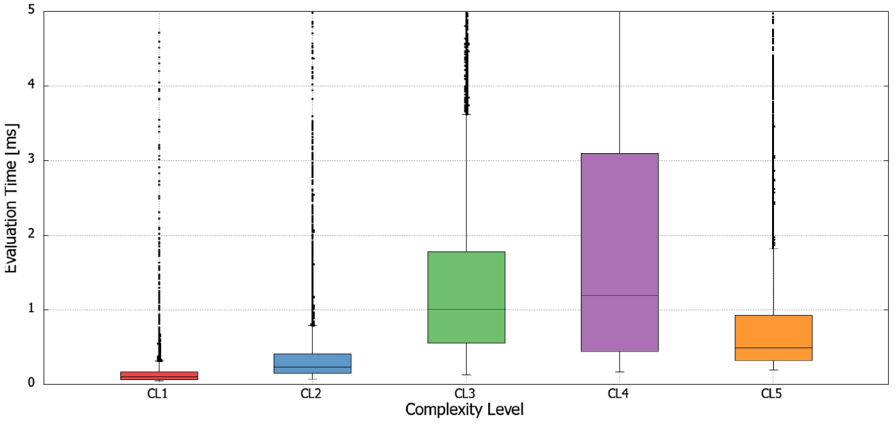

Fig. 11 Boxplots of median evaluation times (in ms) for all constraints, grouped by complexity level (cf. Table 5)

Table 5 Overview of the 21 constraints used for the evaluation, showing their types (T) and indicating their complexity level (CL), and describing the number of checks performed (# checks) and the median evaluation time (MET) for each constraint during the 6-h evaluation run for CDSL version 1 and 2

Const.	T	CL[a]	Name	# Checks	MET v1 [ms][b]	MET v2 [ms]
CST-01	F	2	PlanChangeUserCheck	495	*[1.38]*	0.44
CST-02	F	3	CastSequenceValidityCheck	496	*[10.55]*	0.57
CST-03	D	3	CheckCrossSectionRange	495	8.78	2.40
CST-04	D	1	CheckOptiRunId	498	*[22.00]*	0.08
CST-05	P	4	CheckCastingArmProcedure	28	14.24	16.70
CST-06	D	1	CheckAvailableDiskSpace	13	*[0.66]*	0.15
CST-07	D	1	CheckAvailableStorage	13	*[0.60]*	0.14
CST-08	D	2	CheckCastWatchdogStates	7539	0.36	0.20
CST-09	F	3	CheckOptiCalcRun	436	0.76	0.43
CST-10	F	3	CheckOptiRunConsistency	496	1.05	0.50
CST-11	D	2	CheckStrandNumbersValid	497	4.66	0.16
CST-12	F	3	CheckOptiRunCycle	505	*[9.71]*	0.61
CST-13	P	3	CheckStrandSpeedLength	44,370	*[1.55]*	1.04
CST-14	F	4	CheckOptiRunCycleInclData	504		3.20
CST-15	C	2	DiskChk (CST-06 & CST-07)	13		0.50
CST-16	F	4	OptiPrcCrossEventDataCheck	499		0.61
CST-17	D	2	CheckDiskSpaceAndStorage	13		0.36
CST-18	F	4	ExtendedOptiRunCycleCheck	530		0.57
CST-19	C	5	OptimizerStandardConfCheck (CST-14 & CST-16 & CST-18)	1027		2.60
CST-20	D	1	CheckIfSimulationRun	7587		0.11
CST-21	C	5	OmitWatchdogIfSimulation (CST-20 & CST-08)	7651		0.45

T type of constraint, *F* future, *D* data, *P* past, *C* composite

[a] CL, complexity level: 1 (lowest)–5 (highest); details described in text

[b] The MET v1 values in square-brackets are the original results from Vierhauser et al. (2015), which cannot directly be compared since this evaluation had to be performed on another PC with different hardware. The other MET v1 values are the results of the original constraints from the former evaluation, that could be executed on the new machine to make them comparable

retrieve more events for past occurrence constraints that do not use the "previously" keyword in CDSL version 2. The processing of those additional events requires more time than before. The benefit of this decision is that it allows us to extract more, and more accurate information about constraint violations. The drawback is that it increases the evaluation time.

We designed some of the new constraints based on existing ones, however, extended with new functionality of CDSL version 2. This allows us to directly compare these constraints to identify the impact of the new functionality on the performance of the constraint. For instance, CST-18 is the same as CST-12, but includes a negation and

an optional event. Our evaluation shows that the new capabilities have no significant impact on this constraint's performance [e.g., 0.04 ms (CST-12 vs. CST-18)].

To analyze the performance of composite constraints and to identify their overhead, which results from additional trigger checks and additional code to combine the constraints' results, we compared their execution time with the execution times of the constituent constraints. To compare the execution times of a composite constraint and its sub-constraints, we thus summed up the evaluation times of all sub-constraints multiplied by their evaluation count and compared it to the evaluation time of the composite constraint multiplied with its evaluation count. For instance, for CST-19: composite constraint: 1027*2.60 = 2670.20 ms; sub-constraints: 504*3.20 + 499*0.61 + 530*0.57 = 2219.29 ms. The overhead of the composite constraint therefore is 0.44 ms per evaluation in this case. The overheads of the other composite constraints are 0.21 ms (CST-15) and 0.14 ms (CST-21). Also, while the complexity of composite constraints is rather high (as they combine multiple sub-constraints; cf. Table 5), their overall median evaluation time remains quite low. We thus consider the performance of composite constraints as sufficient (also see Fig. 11).

Although we did not observe as many events in this evaluation compared to the previous 10-h run reported in Vierhauser et al. (2015), the total number of constraint checks, as well as the number of constraint checks per minute is higher in this evaluation. Also, we have evaluated more complex constraints, e.g., combining sequences of multiple events with multiple data checks or even combining multiple constraints. Our evaluation demonstrated that there is no increase in the evaluation time over the whole run (cf. Figs. 7, 8, 9, 10) and we showed that the MET of nearly all constraints improved in comparison to the first version of the DSL-based approach, especially also for more complex constraints. Therefore, we claim that our second version of the DSL-based approach is still scalable and applicable to a real-world SoS and we could even improve scalability compared to the first version.

7 Discussion, lessons learned and threats to validity

Our evaluation demonstrates that the new DSL-based approach is expressive and scales to a real-world SoS. Here we also discuss improvements regarding the extensibility of our new approach compared to the first version. We further revisit the lessons we reported in Vierhauser et al. (2015) and report new lessons we learned when developing version 2. Further open issues and future work are discussed in the conclusions.

7.1 Extensibility

Due to the close coupling between DSLs and the systems they have been designed for and due to the YAGNI principle, it can be expected that the DSL will (again) have to be adapted at some point in the future. One of the major goals of this work thus was to address this issue by designing the DSL to make it more extensible.

The most important changes with respect to extensibility from version 1 to version 2 were unifying parts of the grammar, improving the modularization of the whole

grammar, redesigning of the code generation and the code executed by the constraints, and the testing framework (cf. Sect. 5). Maintainability and extensibility strongly benefit from reusing existing software components. Therefore, creating a highly modular and consistent DSL is very likely to reduce the costs for future changes. This is what the redesign of the grammar and the code generation mainly focused on. Furthermore, every change made to the DSL has to be tested thoroughly. Since most of our DSL's functionality is interdependent, side effects of code changes are very likely to occur. Therefore changes require testing the whole functionality of the DSL rather than testing just the components in which the changes were actually implemented. As mentioned in Sect. 5.3 only the implementation of the automated testing framework allowed sufficient testing, as well as regression tests with reasonable costs.

To show the effects of our improvements, we compare the evolution process using an example requirement: assume in the future it becomes necessary to support writing a constraint that checks whether temperature sensor data works with certain temperature models. For simplicity's sake, we imagine the temperature model in this example as an arbitrary function *f(x)* representing a required temperature value for a given progress *x* of the current process. The required extensions can thus be broken down into two main actions: changing the grammar to express functions and implementing the functions.

In version 1 of our DSL-based approach, implementing this requirement would have worked as follows: to represent this new kind of data check in the grammar, one has to add a new NTS containing the grammar for the check. This NTS is then added to the available data items using a disjunction. This may look like "`Model: 'model(' path=STRING ',' progress=DOUBLE ')'`". Afterwards, one has to create a method for the Xtext parser which has to be executed if a model data item was defined. This method has to create the code for loading and representing the model, as well as creating a new variable in which the retrieved value for the given progress from the model has to be stored for the following comparison. It is crucial that this code snippet does not interfere with the rest of the generated code and that variables aren't defined twice (e.g., if multiple data checks were defined). As this code actually is Xtend code instead of Java code, there are no Java syntax checks at development time, which increases the need for extensive testing of this component. Finally, the developer has to add missing imports for the new data check in the constraint template. As testing has to be done manually in the first version of our approach, the developer has to create mockup probes to simulate a real system, as well as several sample constraints to test the various constraint checking possibilities. The results of the tests also have to be checked manually. Referring to Sect. 4.1, the required grammatical changes would be of Level 1 (DSL syntax adaptation), the changes to the code generation would be of Level 3 (Code generation adaptation) and the changes in the constraint template would be of Level 5 (Constraint Engine adaptation). The overall level for these changes would therefore be Level 5, having a medium to high impact.

In comparison to this, we consider the implementation in the second version of our approach as more easy and straightforward: the implementation of the data check in the grammar works similar to version 1, since this part of the grammar was already sufficiently modularized and did not require any further improvement. The required Xtend code—which has to be called if a model is defined—can, however, be limited

to a simple object instantiation like "*new ModelDataItem*("*<< model.path >>*","*<< model.progress >>*)". To implement the code for loading and representing the model, a Java class *ModelDataItem*, which derives from *DataItem* has to be created. Afterwards, the developer has to define additional test cases in the testing framework for the new constraints, which can then be executed and evaluated automatically by running the testing framework. Additionally, every previously existing test case will be checked also to ensure that no other functionality of the approach was affected by this change. With regard to the levels of change (cf. Sect. 4.1), the impacts for the changes to the grammar would still be of Level 1 and the changes to the code generation would also be of Level 3. However, the constraint engine does not need to be adapted. The overall change level therefore is Level 3 with estimated impacts rated as minor to medium.

The major improvements that can be seen from this sample scenario are the impact of the reengineering of the architecture—particularly the complex and error-prone Xtend code generation part—and the possibility to easily create test cases and perform automated regression testing. Based on this discussion and the effects of our improvements shown in the sample evolution scenario, we claim that we have significantly improved our DSL's extensibility and maintainability.

7.2 Lessons learned

In this section we reassess the lessons we had learned earlier and report new lessons we learned when evolving our approach.

Systems of systems require an iterative language design For both cases, when developing the first version of the DSL and when developing the second version, several people were involved in the evolution of the language. To meet their requirements, we had to apply an iterative approach to design the new language and its features, i.e., we frequently presented and iteratively refined our solution.

Keep the YAGNI ("you aren't gonna need it") principle in mind when developing a DSL When starting to collect the requirements for the first version of our DSL, many different alternatives and ideas on what could be monitored using the constraint-based approach were identified. To keep the language simple and to meet our industry partner's needs, we decided to stick only to those functionality with an actual use case. When evolving the language, we also had several ideas for possible language features, which could be improved or extended. However, most of them turned out to be not needed by our industry partner and thus would make the language unnecessary complicated. We conclude that the YAGNI principle was very beneficial for the DSL's evolution since it avoids possibly unnecessary functionality in the DSL, which might have to be removed or adapted in future versions.

Simplify and automate extending the DSL When developing the first version, technologies were chosen that allow extending the DSL easily, since it was already expected that it will have to be extended in the near future. When the CDSL had to be evolved, we noticed that using such technologies is indeed essential, however, just relying on these technologies' capabilities is not enough. Xtext and Xtend make it easy to develop a DSL, to compile DSL code to a target language like Java, and to provide end-user

support for developing DSL code. However, to successfully extend a DSL, the DSL itself as well as the code generated by the constraint compiler have to be extensible too. This was the crucial reason why we re-developed the grammar and code generation to further increase the extensibility and adaptability of our approach.

Keep the mapping of constraint DSL to constraint engine flexible Since we did not have to cope with dramatic increases of the required performance of our constraint engine, there was no necessity yet to actually exchange the engine. We still recommend to keep the mapping between the used components flexible to avoid possibly upcoming performance problems and allow the future replacement of the engine.

Runtime monitoring requires dynamic constraint management The capabilities for adding, removing and modifying constraints at runtime were excessively used since the first release of our approach. Also the study on the usefulness of the REMINDS Framework and the CDSL (Rabiser et al. 2016) showed that support for dynamic constraint management is crucial for runtime monitoring in SoS which confirms our lesson.

We conclude that our previous lessons learned [cf. (Vierhauser et al. 2015)] still hold after the evolution of the DSL performed over the course of about one and a half years. Additionally, we report new lessons we learned during the evolution, which might be useful for other researchers and practitioners working on/evolving DSLs:

Design the DSL to keep the effort for adaptations minimal As mentioned before, relying on the capabilities of technologies such as Xtext and Xtend in terms of extensibility is insufficient. To successfully evolve a DSL with minimal required effort, all components have to be designed to be extensible. This requires to think about possible future evolution scenarios when developing or evolving a DSL and to adapt its architecture and interfaces based on this knowledge.

Consider backward compatibility When a DSL is evolved, typically many different instances (i.e., constraints) written with the DSL do already exist. Porting these instances to the DSL's new version often causes a huge effort, both, on the developers' and on the users' side. Removing functionality from the DSL might even make it impossible to port existing instances. We suggest to omit these problems by avoiding unnecessary functionality from the beginning (YAGNI), only making changes that extend/improve the DSL, and keeping backward compatibility as far as possible.

Automated testing is crucial As mentioned in Sect. 5.3, manual testing of the DSL-based approach became infeasible in our case quickly. Due to the variety of different DSL instances (i.e., constraints), testing DSLs is especially challenging. To cope with this ever-increasing complexity when evolving a DSL, support for automated testing of the DSL is crucial. In our case, we developed a testing framework which automatically compiles, executes and evaluates predefined test cases. This testing framework can also be used for regression testing on every update of the DSL to check for possible errors resulting from side effects of changes.

7.3 Threats to validity

In terms of external validity, our results and findings are based on a single directed SoS (in two versions) from the domain of industrial automation. We thus cannot claim

that the DSL is capable of covering all possible types of constraints in other systems. However, our knowledge of other systems suggests similar constraint patterns. Due to the flexibility and extensibility of both, the DSL and the underlying architecture it is possible to consider additional constraint types if needed. Also, the requirements and constraints selected for the evaluation might not cover the full range of requirements existing in the PAS. However, we consider our evaluation a good starting point representing a realistic case.

The evaluation focuses on the expressiveness of the DSL (RQ1) and on the scalability of the constraint engine (RQ2). We deliberately did not discuss end-user tools in detail as this is part of a separate study (Rabiser et al. 2016) assessing the usefulness of the different tools and editors for writing and managing constraints. However, the constraints and the DSL were developed together with engineers of our industry partner leading to rapid feedback, which resulted in several adaptations and improvements during the development process, for both versions of our approach.

Regarding the rather small number of constraints used in our scalability evaluation (21), the number of constraint instances created at runtime and the number of constraint checks performed have a much higher impact on the scalability of our approach. We demonstrated in our evaluation that even for complex constraints leading to many instances and checks our approach does provide immediate feedback.

Our evaluation of the scalability of our constraint-based approach in Sect. 6.2 demonstrated that it is indeed scalable and applicable to a real-world SoS and that we could even improve scalability when compared to the first version, especially also for complex constraints. In this paper, we did not focus on the monitoring infrastructure (probes and event and data processing) but on the constraint evaluation. As constraints are evaluated on a separate machine, the constraint checking has no influence on the monitored software (no overhead in this regard). Also, we already have evaluated the overhead of the monitoring infrastructure in earlier work (Vierhauser et al. 2016b) and could confirm that the underlying infrastructure is capable of handling a high amount of events. The earlier evaluation, for example, showed that the overhead for our probes ranges from 1 to 13% for typical instrumentations, but can go up to 70% when serializing complex data structures. As probes are meant to be small, atomic code fragments only collecting specific data in the SoS, and serializing complex objects is rather the exception, this has not been an issue in our industrial context so far.

8 Related work

Different research communities have been developing *approaches for monitoring systems, e.g., to detect violations of requirements of certain properties at runtime*. Examples include requirements monitoring (Vierhauser et al. 2016a; Maiden 2013; Robinson 2006; Fickas and Feather 1995), resource and event monitoring (van Hoorn et al. 2012; Bubak et al. 2004; Ludwig et al. 1997), complex event processing (Völz et al. 2011; Luckham 2011), runtime verification (Calinescu et al. 2012; Ghezzi et al. 2012), and monitoring of probabilistic properties (Cailliau and van Lamsweerde 2017; Zhang et al. 2011; Sammapun et al. 2005). Our own systematic literature review provides a detailed overview (Vierhauser et al. 2016a). Other researchers have also focused

on providing an overview of existing monitoring approaches, e.g., Dwyer et al. (1999) discuss patterns in property specifications for finite-state verification, Delgado et al. (2004) provide a taxonomy and catalog of runtime software fault monitoring tools, Kanstrén (2011) presents a systematic review and taxonomy of runtime invariance in software behavior, and Autili et al. (2015) align qualitative, real-time, and probabilistic property specification patterns. What can be learned from all these reviews and surveys is that a wide variety of different *constraint languages* exist for defining requirements, system properties, or desired event sequences. We have presented nine examples of constraint languages in Sect. 2 and analyzed their support for three challenges in SoS monitoring (see Table 1). We demonstrated that they do not fully support all three challenges. This is why we developed our own DSL in the first place. As we have described in detail above, we support coping with the diversity of constraints in large-scale systems, support the incremental definition and runtime management of constraints, and provide end-user support for constraint definition (Rabiser et al. 2016).

Developing domain-specific languages to support (industrial) end users in complex tasks has been discussed in detail in related work, e.g., by Hermans et al. (2009) in the area of Web services and by Voelter and Visser (2011) in the area of product line engineering. Hermans et al. (2009) have identified several success factors for the use of DSLs in an industrial context: reliability, usability, productivity, learnability, expressiveness, and reusability. Our experiences confirm these success factors (Rabiser et al. 2016).

As also confirmed by our case, due to volatile user requirements and new technologies DSLs, similar to the software systems they describe or produce, are subject to continuous evolution. A lot of research has already been published on *DSLs and their evolution*. We have conducted a systematic mapping study to structure the research field of DSL evolution (Thanhofer-Pilisch et al. 2017). As we described in this mapping study, since 2005 an *increase of papers focusing on DSLs* can be observed, showing the growing relevance of this topic over the last years. *Papers are, however, mostly solution proposals and experience papers* (Wieringa et al. 2006). This might be caused by the still young topic of DSL evolution, i.e., in early phases many new solutions and corresponding experiences are typically published, while validation and evaluation of already existing solutions are less frequent. However, the fact that *only few evaluation research exists so far* is still an indicator that more studies should be conducted. With this paper, in which we provide an evaluation of our evolved DSL, we contribute to this lack.

Our systematic mapping study (Thanhofer-Pilisch et al. 2017) also revealed three DSL evolution trends in existing work, i.e., seven approaches focus on *(i) automating DSL evolution* (often with a special focus on automating the migration of old DSL instances to the current grammar) (Nikolov et al. 2015; Juergens and Pizka 2006; Meyers and Vangheluwe 2011; Vermolen and Visser 2008; Schmidt et al. 2013; Pizka and Jurgens 2007; De Geest et al. 2008), four approaches put their focus on *(ii) efficient DSL creation and maintainability* (Albuquerque et al. 2015; Cazzola and Poletti 2010; van den Bos and van der Storm 2013; Izquierdo and Cabot 2012), and two approaches emphasize the *(iii) reuse of (parts of) existing DSLs* (Degueule et al. 2015; Zschaler

et al. 2009). Our own work also follows this trend and focuses on (ii) and (iii) as discussed above.

Another conclusion of the systematic mapping study (Thanhofer-Pilisch et al. 2017) was that more publications reporting lessons learned from practical DSL evolution are needed. This is also something we provide with this work.

9 Conclusions and future work

In this work we described how we evolved a DSL-based approach for defining and supporting checking constraints when monitoring systems of systems at runtime. When evolving the DSL-based approach to its second version, we re-developed its grammar and the code generation components which parse the constraint and generate the required Java code to execute the defined checks. The aim of re-developing the grammar was to add additional functionality, to increase its modularization to support reuse and to ease future changes, as well as to remove usability flaws. Re-developing the code generation was necessary to increase the maintainability and extensibility of the approach to support future evolution scenarios. Additionally, we developed a testing framework to automate testing the approach since manual testing became infeasible.

In our evaluation, we showed that the new version of the CDSL is more expressive than the previous one and sufficiently expressive to allow its use in a real-world SoS. Furthermore, we assessed the performance of our approach and showed that it is scalable enough to apply it to a SoS with a realistic number of occurring events and corresponding constraint checks. Lastly, we discussed the performed changes and their impact on the extensibility of our approach. We revisited the lessons learned when developing the first version of our approach and presented additional lessons learned when evolving the approach to its second version, i.e., that it is essential to design the approach in a way that keeps the effort for adaptations minimal, to enable backward compatibility as far as possible, and to support automated testing.

While we could improve our DSL-based approach significantly, some issues still remain motivating future work. Currently, data checks on events are evaluated after the rest of the constraint has passed the evaluation. Therefore, the evaluation process might wait until future events occur to verify a specified event sequence, even though the data check of the first event might already violate, which makes the rest of the evaluation obsolete. The new architecture for constraint evaluations would allow to change this behavior to evaluate data checks as soon as possible, which we assume would result in an even lower constraint evaluation time. Automated constraint migration is another feature we plan to develop. We are also currently exploring constraint mining (Krismayer et al. 2017), i.e., analyzing recorded event streams, e.g., for recurring patterns, to generate constraints. We also do expect further changes to come up in the future due to changing requirements and new use cases. When making further changes to the DSL, we could then also evaluate the effort for their implementation to further assess the effects of the increased extensibility of our approach.

Acknowledgements This work has been supported by the Christian Doppler Forschungsgesellschaft (Grant No. CDL MEVSS), Austria and Primetals Technologies. Michael Vierhauser's work was funded by the

Austrian Science Fund (FWF) under Grant No. J3998-N31. We want to thank Thomas Krismayer for his feedback and his help in setting up the evaluation environment.

Appendix: grammar of the CDSL v.2

Listing 5 Grammar of our constraint DSL for specifying past occurrence, future occurrence, and data constraints as well as composite constraints combining these types.

```
Constraint: ConstraintDefinition | CombinedConstraintDefinition

CombinedConstraintDefinition: combined constraint = constraint 'constraint_name' {and constraint '
    constraint_name'} | {or constraint 'constraint_name'}

ConstraintDefinition: TriggerCondition [DataCheckCondition] (PastCondition | FutureCondition)

TriggerCondition: trigger = if event EventDefinition occurs

EventDefinition: 'event_type' [EventScope] [as 'event_shortcut_key'] [where DataCheck]

DataCheckCondition: condition = DataCheck

PastCondition: condition = [!] event EventDefinition has occurred PastEvalCriteria

FutureCondition: condition = (SingleFuture | SequenceFuture) FutureEvalCriteria [StopCriteria]

DataCheck: DataCheckType {and DataCheckType} | {or DataCheckType}

DataCheckType: ValueCondition | RangeCondition

RangeCondition: within '(' DataItem , 'within_percentage' % , DataItem ')'

ValueCondition: DataItem RelationalOperator DataItem

DataItem: DataKey | DataValue | ConfigurationParameter | ExternalDataItem

ExternalDataItem: external '(' event EventDefinition , [EventSelection ,] DataKey ')'

EventSelection: select '(' ('latest'|'next') ')'

DataKey: ['event_shortcut_key' '.'] data '(' 'data_item_name' [, 'data_key_path'] [, ('amount' |
    Function)] ')'

ConfigurationParameter: config '(' [(static | runtime) ,] [EventScope ,] 'config_key' ')'

EventScope: from scope '(' ('scope_id' | same) ')'

FutureEvalCriteria: [in any order] [ignore others] [until EventDefinition occurs] [within Time]

StopCriteria: evaluation after Time

SingleFuture: [!] event EventDefinition occurs

SequenceFuture: events SequenceEvent {, SequenceEvent} [EventScope] occur

SequenceEvent: [? | !] EventDefinition

PastEvalCriteria: [previously] [since EventDefinition occurred] [in the last Time]

Function: size | unique | contains

DataValue: DOUBLE | STRING

Time: INT TimeUnit
```

TimeUnit: **seconds** | **minutes** | **hours** | **days**

BoolOperator: **and** | **or**

RelationalOperator: > | >= | < | <= | != | ==

References

Aktug, I., Dam, M., Gurov, D.: Provably correct runtime monitoring. In: Formal Methods (FM'08), pp. 262–277. Springer (2008)

Albuquerque, D., Cafeo, B., Garcia, A., Barbosa, S., Abrahão, S., Ribeiro, A.: Quantifying usability of domain-specific languages: an empirical study on software maintenance. J. Syst. Softw. **101**, 245–259 (2015)

Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A.: Aligning qualitative, real-time, and probabilistic property specification patterns using a structured English grammar. IEEE Trans. Softw. Eng. **41**(7), 620–638 (2015)

Baresi, L., Guinea, S.: Event-based multi-level service monitoring. In: Proceedings of the 20th International Conference on Web Services, pp. 83–90. IEEE (2013)

Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: Foundations of Software Technology and Theoretical Computer Science, pp. 260–272. Springer (2006)

Bertolino, A., Calabrò, A., Lonetti, F., Di Marco, A., Sabetta, A.: Towards a model-driven infrastructure for runtime monitoring. In: Software Engineering for Resilient Systems, pp. 130–144. Springer (2011)

Bubak, M., Funika, W., Smetek, M., Kiliański, Z., Wismüller, R.: Event handling in the J-OCM monitoring system. In: Parallel Processing and Applied Mathematics, pp. 344–351. Springer (2004)

Bures, T., Hnetynka, P., Plasil, F.: Strengthening architectures of smart CPS by modeling them as runtime product-lines. In: Proceedings of the 17th International ACM SIGSOFT Symposium on Component-Based Software Engineering, pp. 91–96. ACM (2014)

Cailliau, A., van Lamsweerde, A.: Runtime monitoring and resolution of probabilistic obstacles to system goals. In: 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 1–11. IEEE (2017)

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

Cazzola, W., Poletti, D.: DSL evolution through composition. In: Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution, pp. 6:1–6:6. ACM (2010)

Chen, F., d'Amorim, M., Roşu, G.: A formal monitoring-based framework for software development and analysis. In: Formal Methods and Software Engineering, pp. 357–372. Springer (2004)

De Geest, G., Vermolen, S., Van Deursen, A., Visser, E.: Generating version convertors for domain-specific languages. In: Proceedings of the 15th Working Conference on Reverse Engineering, pp. 197–201. IEEE (2008)

Degueule, T., Combemale, B., Blouin, A., Barais, O.: Reusing legacy DSLs with melange. In: Proceedings of the Workshop on Domain-Specific Modeling, pp. 45–46. ACM (2015)

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

Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the International Conference on Software Engineering, pp. 411–420. IEEE (1999)

Egyed, A.: Instant consistency checking for the UML. In: Proceedings of the 28th International Conference on Software Engineering, pp. 381–390. ACM (2006)

Eichelberger, H., Schmid, K.: Flexible resource monitoring of Java programs. J. Syst. Softw. **93**, 163–186 (2014)

Fickas, S., Feather, M.S.: Requirements monitoring in dynamic environments. In: Proceedings of the 2nd IEEE International Symposium on Requirements Engineering, pp. 140–147. IEEE (1995)

Ghezzi, C., Mocci, A., Sangiorgio, M.: Runtime monitoring of component changes with Spy@Runtime. In: 34th International Conference on Software Engineering, pp. 1403–1406. IEEE (2012)

Gunadi, H., Tiu, A.: Efficient runtime monitoring with metric temporal logic: a case study in the Android operating system. In: Proceedings Formal Methods 2014, pp. 296–311. Springer (2014)

Hermans, F., Pinzger, M., van Deursen, A.: Domain-specific languages in practice: a user study on the success factors. In: Schürr, A., Selic, B. (eds.) Model Driven Engineering Languages and Systems, pp. 423–437. Springer, Berlin (2009)

Izquierdo, J.L.C., Cabot, J.: Community-driven language development. In: Proceedings of the 2012 4th International Workshop on Modeling in Software Engineering, pp. 29–35 (2012)

Juergens, E., Pizka, M.: The language evolver lever—tool demonstration. Electr. Notes Theor. Comput. Sci. **164**(2), 55–60 (2006)

Kanstrén, T.: A systematic review and taxonomy of runtime invariance in software behaviour. Int. J. Adv. Softw. **4**(3 and 4), 256–274 (2011)

Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-MaC: a run-time assurance approach for Java programs. Form. Methods Syst. Des. **24**(2), 129–155 (2004)

Kiviluoma, K., Koskinen, J., Mikkonen, T.: Run-time monitoring of architecturally significant behaviors using behavioral profiles and aspects. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis, pp. 181–190. ACM (2006)

Krismayer, T., Rabiser, R., Grünbacher, P.: Mining constraints for event-based monitoring in systems of systems. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pp. 826–831. IEEE (2017)

Luckham, D.C.: Event Processing for Business: Organizing the Real-Time Enterprise. Wiley, New York (2011)

Ludwig, T., Wismueller, R., Sundcram, V., Bode, A.: OMIS: On-line Monitoring Interface Specification (v. 2.0). Technical report TUM-I9733, Technische Universität München (1997)

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

Maier, M.W.: Architecting principles for systems-of-systems. Syst. Eng. **1**(4), 267–284 (1998)

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

Meyers, B., Vangheluwe, H.: A framework for evolution of modelling languages. Sci. Comput. Program. **76**(12), 1223–1246 (2011)

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

Muccini, H., Polini, A., Ricci, F., Bertolino, A.: Monitoring architectural properties in dynamic component-based systems. In: Component-Based Software Engineering, LNCS 4608, pp. 124–139. Springer (2007)

Nielsen, C.B., Larsen, P.G., Fitzgerald, J., Woodcock, J., Peleska, J.: Systems of systems engineering: basic concepts, model-based techniques, and research directions. ACM Comput. Surv. **48**(2), 18:1–18:41 (2015)

Nikolov, N., Rossini, A., Kritikos, K.: Integration of DSLs and migration of models: a case study in the cloud computing domain. Procedia Comput. Sci. **68**, 53–66 (2015)

Paschke, A.: RBSLA—a declarative rule-based service level agreement language based on RuleML. In: International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce, vol. 2, pp. 308–314. IEEE (2005)

Pizka, M., Jurgens, E.: Automating language evolution. In: Proceedings of the 1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, pp. 305–315. IEEE (2007)

Rabiser, R., Vierhauser, M., Grünbacher, P.: Assessing the usefulness of a requirements monitoring tool: a study involving industrial software engineers. In: Proceedings of the 38th International Conference on Software Engineering, Companion, pp. 122–131. ACM (2016)

Rabiser, R., Vierhauser, M., Grünbacher, P.: Variability management for a runtime monitoring infrastructure. In: Proceedings of the 9th International Workshop on Variability Modelling of Software-Intensive Systems, pp. 35–42. ACM (2015)

Rabiser, R., Guinea, S., Vierhauser, M., Baresi, L., Grünbacher, P.: A comparison framework for runtime monitoring approaches. J. Syst. Softw. **125**(March), 309–321 (2017)

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

Robinson, W.N.: Extended OCL for goal monitoring. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. **9**(1), 1–12 (2008)

Sammapun, U., Lee, I., Sokolsky, O.: RT-MaC: runtime monitoring and checking of quantitative and probabilistic properties. In: Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 147–153. IEEE (2005)

Schmidt, M., Wider, A., Scheidgen, M., Fischer, J., von Klinski, S.: Refactorings in language development with asymmetric bidirectional model transformations. In: Proceedings of the 16th International SDL Forum on Model-Driven Dependability Engineering, pp. 222–238. Springer (2013)

Skene, J., Emmerich, W.: Engineering runtime requirements-monitoring systems using MDA technologies. In: Nicola, R.D., Sangiorgi, D. (eds.) Trustworthy Global Computing, pp. 319–333. Springer (2005). https://doi.org/10.1007/11580850

Spanoudakis, G., Kloukinas, C., Mahbub, K.: The SERENITY runtime monitoring framework. In: Kokolakis, S., Gómez, A.M., Spanoudakis, G. (eds.) Security and Dependability for Ambient Intelligence, pp. 213–237. Springer (2009). https://doi.org/10.1007/978-0-387-88775-3

Spinellis, D.: Notable design patterns for domain-specific languages. J. Syst. Softw. **56**(1), 91–99 (2001)

Thanhofer-Pilisch, J., Lang, A., Vierhauser, M., Rabiser, R.: A systematic mapping study on DSL evolution. In: Proceedings of the 43rd Euromicro Conference on Software Engineering and Advanced Applications, pp. 149–156. IEEE (2017)

van den Bos, J., van der Storm, T.: A case study in evidence-based DSL evolution. In: Proceedings of the 9th European Conference on Modelling Foundations and Applications, pp. 207–219. Springer (2013)

Van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. ACM SIGPLAN Not. **35**(6), 26–36 (2000)

van Hoorn, A., Waller, J., Hasselbring, W.: Kieker: A framework for application performance monitoring and dynamic software analysis. In: Proceedings of the 3rd Joint International Conference on Performance Engineering, pp. 247–248. ACM (2012)

van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley, New York (2009)

Vermolen, S., Visser, E.: Heterogeneous coupled evolution of software languages. In: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems, pp. 630–644. Springer (2008)

Vierhauser, M., Grünbacher, P., Egyed, A., Rabiser, R., Heider, W.: Flexible and scalable consistency checking on product line variability models. In: Proceedings of the International Conference on Automated Software Engineering, pp. 63–72. ACM (2010)

Vierhauser, M., Grünbacher, P., Heider, W., Holl, G., Lettner, D.: Applying a consistency checking framework for heterogeneous models and artifacts in industrial product lines. In: Model Driven Engineering Languages and Systems, pp. 531–545. Springer (2012)

Vierhauser, M., Rabiser, R., Cleland-Huang, J.: From requirements monitoring to diagnosis support in system of systems. In: Proceedings of the 23rd International Working Conference on Requirements Engineering: Foundation for Software Quality, pp. 181–187. Springer (2017)

Vierhauser, M., Rabiser, R., Grünbacher, P., Egyed, A.: Developing a DSL-based approach for event-based monitoring of systems of systems: experiences and lessons learned. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, pp. 715–725. IEEE (2015)

Vierhauser, M., Rabiser, R., Grünbacher, P.: A case study on testing, commissioning, and operation of very-large-scale software systems. In: Proceedings of the 36th International Conference on Software Engineering, Companion, pp. 125–134. ACM (2012)

Vierhauser, M., Rabiser, R., Grünbacher, P.: Requirements monitoring frameworks: a systematic review. Inf. Softw. Technol. **80**, 89–109 (2016a)

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

Viswanathan, M., Kim, M.: Foundations for the run-time monitoring of reactive systems—fundamentals of the MaC language. In: Liu, Z., Araki, K. (eds.) Theoretical Aspects of Computing, 1st International ICTAC Colloquium, (Revised Selected Papers), Lecture Notes in Computer Science 3407, pp. 543–556. Springer (2005)

Voelter, M., Visser, E.: Product line engineering using domain-specific languages. In: Proceedings of the 15th International Software Product Line Conference, IEEE CS, pp. 70–79 (2011)

Völz, M., Koldehofe, B., Rothermel, K.: Supporting strong reliability for distributed complex event processing systems. In: Proceedings of the 13th International Conference on High Performance Computing & Communication, pp. 477–486. IEEE (2011)

Whittle, J., Sawyer, P., Bencomo, N., Cheng, B., Bruel, J.M.: RELAX: a language to address uncertainty in self-adaptive systems requirement. Requir. Eng. **15**(2), 177–196 (2010)

Wieringa, R., Maiden, N., Mead, N., Rolland, C.: Requirements engineering paper classification and evaluation criteria: a proposal and a discussion. Requir. Eng. **11**(1), 102–107 (2006)

Zhang, P., Li, B., Muccini, H., Sun, M.: An approach to monitor scenario-based temporal properties in Web service compositions. In: Advanced Web and Network Technologies, and Applications, pp. 144–154. Springer (2008)

Zhang, P., Li, W., Wan, D., Grunske, L.: Monitoring of probabilistic timed property sequence charts. Softw. Pract. Exp. **41**(7), 841–866 (2011)

Zschaler, S., Kolovos, D.S., Drivalos, N., Paige, R.F., Rashid, A.: Domain-specific metamodelling languages for software language engineering. In: Proceedings of the 2nd International Conference on Software Language Engineering, pp. 334–353. Springer (2009)