Developing a DSL-Based Approach for Event-Based Monitoring of Systems of Systems: Experiences and Lessons Learned

Michael Vierhauser Rick Rabiser
Christian Doppler Laboratory MEVSS
Johannes Kepler University Linz, Austria

Paul Grünbacher Alexander Egyed
Institute for Software Systems Engineering
Johannes Kepler University Linz, Austria

***Abstract*—Complex software-intensive systems are often described as systems of systems (SoS) comprising heterogeneous architectural elements. As SoS behavior fully emerges during operation only, runtime monitoring is needed to detect deviations from requirements. Today, diverse approaches exist to define and check runtime behavior and performance characteristics. However, existing approaches often focus on specific types of systems and address certain kinds of checks, thus impeding their use in industrial SoS. Furthermore, as many SoS need to run continuously for long periods, the dynamic definition and deployment of constraints needs to be supported. In this paper we describe experiences of developing and applying a DSL-based approach for monitoring an SoS in the domain of industrial automation software. We evaluate both the expressiveness of our DSL as well as the scalability of the constraint checker. We also describe lessons learned.**

***Keywords*—Systems of systems, requirements monitoring, constraint checking, domain-specific languages.**

I. Introduction

Many software-intensive systems today are systems of systems comprising heterogeneous and interrelated architectural elements. Common properties of SoS are decentralized control; support for multiple platforms; inherently volatile and conflicting requirements; continuous evolution and deployment; as well as heterogeneous, inconsistent, and changing elements [1]. As the full behavior of SoS emerges during operation only, system testing is not sufficient to determine compliance with requirements. Instead, the behavior of the systems and their interactions need to be continuously monitored and checked during operation to detect and analyze deviations from the expected behavior. Checks include the occurrence and order of runtime events (temporal behavior) [2], the interaction of systems (structural behavior), or properties of runtime data (data checks).

Different research communities have been developing runtime monitoring approaches for various kinds of systems and diverse types of checks. Examples include requirements monitoring [3], [4], monitoring of architectural properties [5], complex event processing [6], or runtime verification [7], [8] to name but a few. The expected runtime behavior often also can be expressed formally using temporal logic [9]–[12]. Furthermore, domain-specific languages are used to facilitate the definition of constraints [13]–[16], which then can be checked based on events and data collected from systems at runtime, e.g., via probes instrumenting systems [17].

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. Furthermore, many SoS are cyber-physical systems [18] that need to run in 24/7 mode for weeks or even months without interruption, which means that constraints need to be defined and deployed dynamically to ensure live and instant feedback on requirements violations to users. Many existing runtime monitoring approaches, however, emphasize particular technologies or types of constraints, or are limited to offline analysis of event traces [19]–[21].

This paper describes experiences of extending an existing incremental consistency checker for design models [22], [23] to support event-based runtime monitoring of SoS [24]. Our work is motivated by an industrial case of monitoring a metallurgical plant automation system, an example of a large-scale industrial SoS (Section II). We describe an industrial scenario and discuss challenges for constraint checking at runtime. To address these challenges, we developed a *domain-specific constraint language aiming at industrial end users*, who often lack deep programming skills, to ease the definition of various types of constraints (Section III). Our DSL-based approach allows to *incrementally check constraints at runtime* and was developed by extending an existing incremental checker (Sections IV and V). This ensures that violations of requirements can be reported instantly to users monitoring an SoS. The approach further supports the *definition and deployment of constraints at runtime*, i.e., constraints can be added or modified without stopping the checker or the monitored systems. We evaluate the expressiveness of the DSL using real constraints from our industrial case and show the scalability of our checker in an industrial monitoring scenario (Section VI). We also discuss lessons learned aimed at researchers and practitioners working on similar challenges (Section VII). We conclude the paper with a discussion of related research and an outlook on future work (Sections VIII and IX).

II. Industrial Case 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 (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 PAS software systems 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 the input to casting solid steel slabs. The PAS SoS is further connected to legacy or third-party systems leading to additional complexity. Furthermore, there are dependencies between components within one particular system. 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 constraints, which cross-cut different systems or components, there are also constraints affecting only 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 specific sequence and within a certain time span to guarantee the uninterrupted and continuous flow of material.

Although such 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. Furthermore, the metallurgical production process is supervised by operators in control rooms and manual intervention by an operator can have unforeseen effects on the automation software and the production process, again suggesting a runtime monitoring approach. For instance, the cut length defined by an operator of the caster system might conflict with required plan characteristics from the production planning system, or the ladle-finished event might not happen in steelmaking, thus affecting the continuous casting process.

A. 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 [25]. The scenario shows that constraints need to be defined and deployed dynamically and checked continuously at runtime. For the scenario we assume that 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

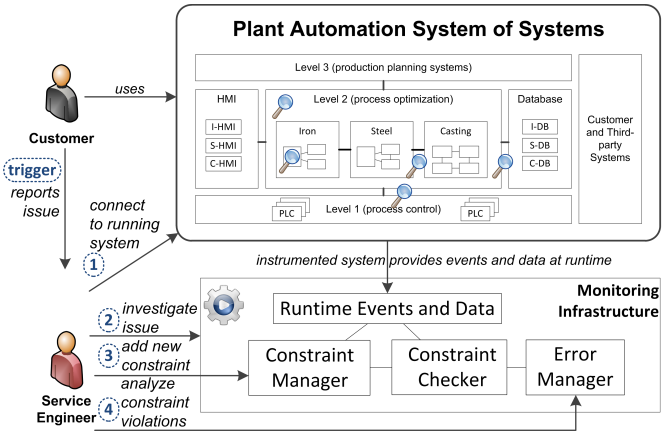


Fig. 1: Industrial scenario for runtime monitoring of SoS.

components 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 potential 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 automation system 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 system, e.g., by analyzing recorded event data to reveal the origin of the reported problem. This task is simplified if constraints related to the issue have already been defined and violations have been recorded. (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 event sequences 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 seconds. The constraint is immediately activated in the monitoring infrastructure to detect deviations from the specified behavior in the running system. (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.

B. Challenges

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

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 Section VIII). 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.

Incremental definition and runtime management of constraints. Constraints are typically not defined once before the system is put into operation but rather incrementally when needed. For instance, as our industrial scenario showed, engineers may need to define additional constraints 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 [26]. Many existing constraint checkers do not support the dynamic definition and management of constraints at runtime (see Section VIII). Specifically, although activating, deactivating, and parameterizing constraints at runtime is supported by some approaches, adding new constraints at runtime typically cannot be done without restarting the monitoring infrastructure.

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 Section VIII), writing new or understanding existing constraints is much harder for users without a deep programming background.

III. A Constraint DSL for SoS Monitoring

Existing constraint languages, e.g., requirements-level methods [14], [27]–[29], UML-based approaches [30], [31], or formal runtime verification techniques [9]–[12], often support specific types of constraints. For example, some approaches focus on monitoring performance properties, 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 [30], [32] or business processes [33]) and are hard to apply in other areas. Furthermore, many existing constraint languages are deemed as inconvenient by industrial end users as they require deep understanding of formal concepts or lack tool support.

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 then developed a DSL allowing engineers to specify temporal, structural, and data constraints on events and data.

We assume a stream of events observed at runtime by a monitoring infrastructure. The events are collected in an event model also managing arbitrary data attached to specific events. For example, events in a material quality optimization system show when the system starts, initializes, optimizes, and stores the optimization results. Each event has a time stamp and data can be attached, e.g., the inputs used for initializing the optimization system, the computed optimization results, or information on the performance of the optimization component.

Listing 1: Grammar of our constraint DSL for specifying past occurrence, future occurrence, and data constraints

```
Constraint:
    trigger = if event 'trigger_event' [with Data] occurs
        (PastOccurrence | FutureOccurrence | DataCheck).

PastOccurrence:
    condition = (event 'event_name' [source='source']
        has occurred [with Data {,Data}])
        |(events 'event_name'{,'event_name'}
        occurred [consecutively])
        previously in the last Time
FutureOccurrence:
    condition = (event 'event_name' [source='source']
        occurs [with Data {,Data}])
        |(events 'event_name'{,'event_name'}
        occur [consecutively]) within Time
DataCheck:
    condition = data Data {,Data}

Data:
    DataItem Operator DataItem | Value
DataItem:
    key('itemname','itempath', [Function])
Function:
    contains | size
Time:
    int milliseconds | seconds | minutes | hours
Operator:
    > | >= | < | <= | == | !=
Value:
    double | int | boolean | String
```

The grammar of our DSL is shown in Listing 1. Each constraint starts with a description of the *trigger* event activating the evaluation of the constraint (cf. Fig. 2), e.g., "optimization requested". To also allow specifying invariants, the event type of the trigger event can be defined as "any". The trigger specification is followed by a *condition* statement. Conditions can be defined to check the *past occurrence* of a (sequence of) event(s) 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*

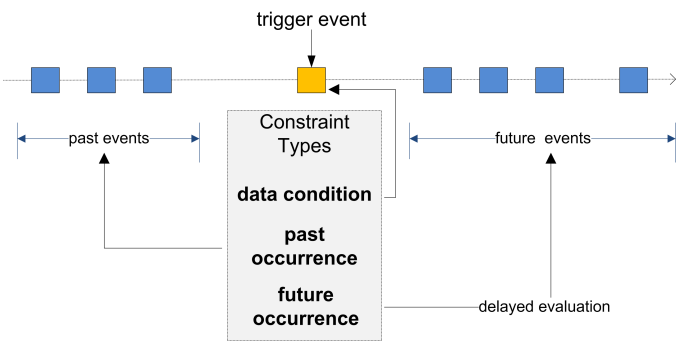


Fig. 2: Event stream and constraint types of our DSL.

Listing 2: 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.
```

limits, the occurrence of an event of a certain type is required within a certain time, e.g., an event of type B must occur within a maximum time of five seconds 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 are frequently used, e.g., to ensure a data item is within a certain range. For character sequences usually only checks for equality are used. 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 2 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%.

Checks on past and future occurrence as well as data checks have been sufficient so far to cover the constraints elicited together with our industry partner (cf. Section VI.A). However, it might be necessary to extend the language in the future to cover additional types of constraints. We have thus developed a DSL-based approach that makes it easy to extend the language.

IV. Extending an Incremental Consistency Checker to support SoS Runtime Monitoring

As runtime monitoring presumes the continuous (re-)evaluation of constraints an incremental evaluation strategy is advisable to ensure fast evaluation feedback to users in case of violations. We therefore decided to use an incremental consistency checker (ICC) [22], [23], [34] developed in our previous work on consistency checking of design models in an IDE. To address the challenges identified in Section II we extended the original ICC to support runtime checking of events and data. We also integrated this new Runtime Incremental Consistency Checker (RICC) in our SoS Runtime Monitoring Framework [24].

More specifically, the RICC (cf. Fig. 3) covers the existing Runtime Monitoring Framework, the extensions necessary to use the ICC for runtime monitoring, and the original ICC. On top of the RICC we added the extensible DSL to support end users defining constraints in a simple and intuitive manner. The constraints are translated to the underlying language of the ICC, which is Java in our current implementation. When developing the RICC we reorganized the original ICC (which was integrated in an IDE) into several components. We now use a client-server architecture allowing multiple users to contribute and modify constraints. Constraints can be added to the checker at any time without restarting, as the DSL code is compiled dynamically to Java. Furthermore, we provide tool support for different monitoring tasks: an editor to write new constraints, a tool for activating and deactivating constraints on the monitoring server, and a tool for reviewing constraint violations of different components and systems of the SoS at runtime.

The RICC relies on events provided by the Runtime Monitoring Framework, which uses an *Event Model* to abstract from different systems and technologies. 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 behaviour, persisting event logs, or checking constraints on events. 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.

An *Event Model Facade* 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, possibly distributed, sources.

The *DSL Editor* provides 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*. As soon as the constraint definition is transmitted to the checking server, it is compiled to Java on the fly by the RICC to a *Compiled Constraint Definition*, thereby making it usable by the ICC. The *Constraint Manager* allows activating, deactivating, and modifying (groups of) constraints.

The *Constraint Instance Store* is a central component 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

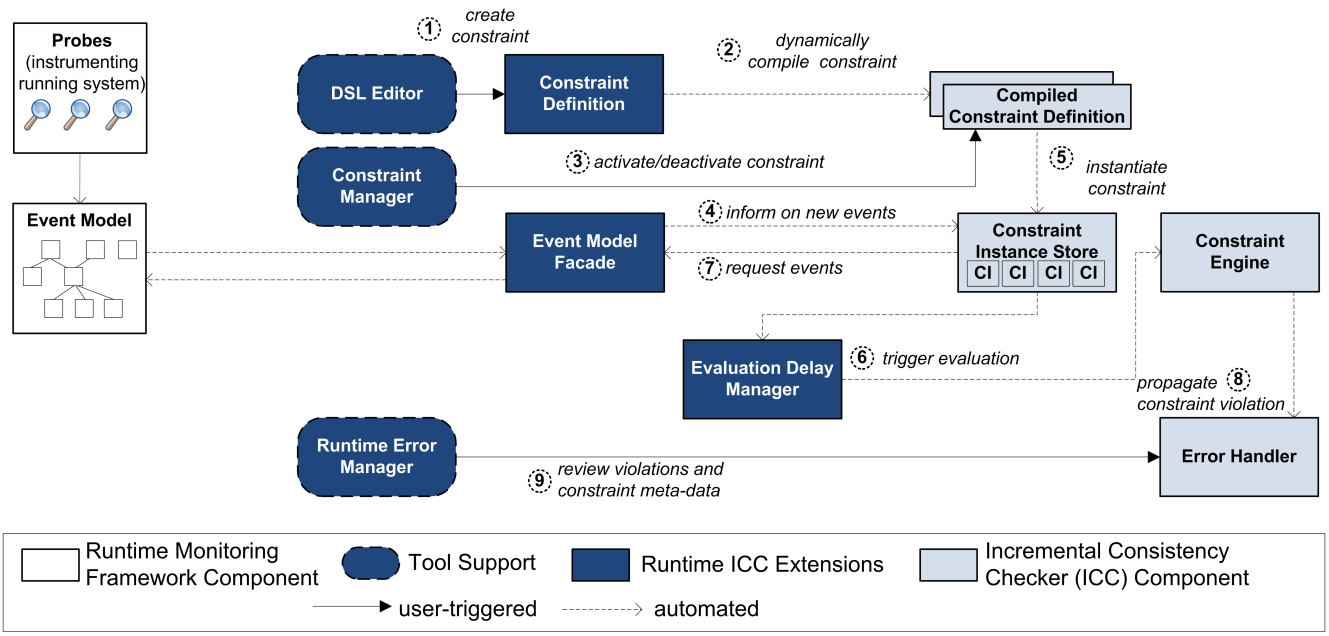


Fig. 3: Our runtime incremental consistency checker (RICC) and its integration with our runtime monitoring framework. The numbers indicate how the different components interact in a typical scenario.

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.

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 *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 checker. If the constraint instance is violated and evaluates to false, further information on the violation is forwarded to the *Error Handler*. This includes the events leading to the violation, missing events, or violated data ranges. Listeners can register to this component to receive this information. For instance, 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.

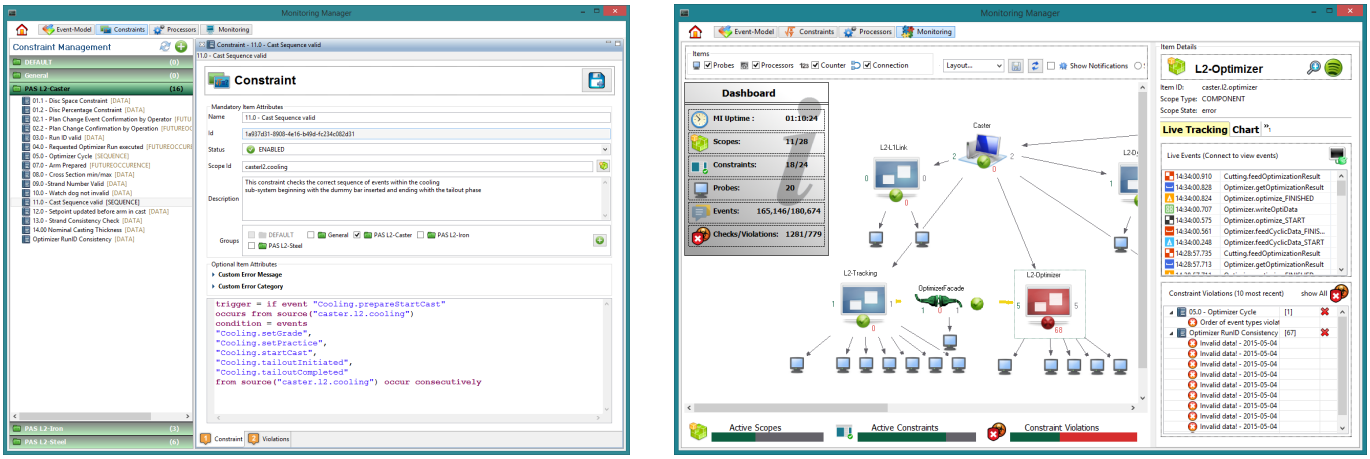
V. Implementation and Tools

The RICC has been developed using different components and technologies. The event model has been implemented in Java as part of the runtime monitoring infrastructure [24]. The model provides interfaces for retrieving events of a certain type or from a specific source. It further allows registering change listeners informing the RICC about new events. The DSL editor and the constraint manager tool are both implemented in Java using the Eclipse RCP framework.

We employed the Java-based frameworks XText and Xtend (http://www.eclipse.org/Xtext) for developing the constraint DSL, end-user guidance in the editor, and support for the dynamic compilation of DSL code to Java code. The frameworks allow adding new language constructs in a rather simple 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(a).

The implementation of the RICC is based on an earlier implementation for checking the consistency of software product lines [34]. We split the original ICC implementation into a client and a server part and use the Java Compiler API to dynamically compile and load constraints on demand. The evaluation delay manager has been implemented as a separate Java component intercepting evaluation requests from the ICC and postponing them in case of future occurrence constraints. The error manager provides interfaces retrieving information on violated constraints and for registering listeners to be notified as soon as constraint violations are detected.

At runtime, a graphical overview is provided of all components of the SoS and their current state as shown in Fig. 4(b). The runtime error manager shown in this view allows reviewing constraint violations related to a specific component, (missing) events responsible for constraint violations, and additional information provided by the constraints.



(a) DSL Editor and Constraint Manager.

(b) Runtime System Overview and Runtime Error Manager.

Fig. 4: Tools for defining and managing constraints (a) and for monitoring and reviewing constraint violations (b).

VI. Evaluation

Our cooperation with Primetals Technologies allowed us to evaluate our approach by applying the runtime monitoring infrastructure and the constraint checking approach to several systems in a realistic setting. We explore two research questions regarding the general applicability of our approach. Specifically, we investigate the expressiveness of our DSL and the scalability of the RICC in a typical monitoring scenario based on Primetals Technologies' SoS.

RQ1 – Is the DSL sufficiently expressive to allow its use in a real-world industrial SoS? We discussed monitoring needs together with architects and engineers in several workshops and conducted a series of interviews to assess the current practices at Primetals Technologies for developing, commissioning, and operating their directed SoS. Furthermore, we studied documents and analyzed different systems part of the PAS. For the evaluation we selected requirements covering the different systems of the PAS and formalized them as constraints.

RQ2 – Does the constraint checking approach scale to industrial needs in the context of a real-world SoS? The company gave us access to their PAS simulation environment, which is used to test the automation software before and during commissioning. We experimented with parts of the PAS in a virtual environment simulating machinery and production planning components. This allowed us to use our approach in an SoS environment without interfering with real production systems in a metallurgical plant. We describe the results of simulations to address RQ2. Specifically, in a ten-hour simulation run we measured the number of constraints instantiated and checks performed, the memory consumption of the constraint checking engine, and the average time necessary to evaluate single constraint instances. We also dynamically added and removed constraints during the ten-hour run to test our capabilities for dynamic constraint definition and checking.

A. RQ1 – DSL Expressiveness

We performed five two-hour workshops and several follow-up meetings with seven system experts to capture requirements, which have to be monitored at runtime, from different parts of the PAS. For example, at SoS level, various non-functional requirements concern monitoring the performance of the systems' hardware. Log and trace files, dumps, and stack traces are continuously created, which can lead to low disk space, requiring to monitor the remaining hard disc capacity. Other requirements cover the interaction of different parts of the system (e.g., user interface and database) or the checking of (the duration of) certain sequences of events. For example, one requirement regards the maximum duration of a metallurgical calculation. We also monitor the optimization component, which calculates the optimal distribution of steel slabs on a casting strand and relies on input data from various other components. Overall, we identified 40 requirements from several different systems of the PAS to be monitored.

For each of these 40 requirements, we discussed with engineers from Primetals Technologies how constraints could be defined for checking the expected behavior at runtime. For example, one requirement on the timely calculation of optimizations led to a future occurrence constraint. This constraint checks that after an optimization run is triggered, either by the operator in his user interface or by another system, the optimization result has to be available within 5 seconds and then fed to the subsequent system. Another requirement on avoiding incorrect or missing quality calculations led to a past occurrence constraint checking that when a robotic arm holding a ladle full of liquid steel is starting to move, the data about the material in the ladle must already have been updated. Yet another requirement on avoiding incorrect optimization results led to a data constraint checking that the "cross section optimization" value is positive and in a certain range when optimizations are being calculated.

During these workshops we defined at least one constraint for each requirement leading to over 40 different constraints. So far the types of constraints our DSL allows to express were sufficient. However, future workshops might reveal new types of constraints requiring the extension of our DSL. For now, we can claim our DSL is expressive enough to define constraints for monitoring the industrial SoS.

TABLE I: Overview of the 13 constraints used for the evaluation, describing the number of checks performed (# checks) and the median evaluation time (MET) for each constraint during the ten-hour evaluation run.

Const.	Type	Name	# checks	MET [ms]
CST-01	FUTURE	PlanChangeUserCheck	128	1.38
CST-02	FUTURE	CastSequenceValidityCheck	2	10.55
CST-03	DATA	CheckCrossSectionRange	747	54.06
CST-04	DATA	CheckOptiRunId	753	22.00
CST-05	PAST	CheckCastingArmProcedure	1	9.74
CST-06	DATA	CheckAvailableDiskSpace	16	0.66
CST-07	DATA	CheckAvailableStorage	16	0.60
CST-08	DATA	CheckCastWatchdogStates	9,700	0.37
CST-09	FUTURE	CheckOptiCalcRun	687	1.20
CST-10	FUTURE	CheckOptiRunConsistency	584	1.37
CST-11	DATA	CheckStrandNumbersValid	548	14.64
CST-12	FUTURE	CheckOptiRunCycle	584	9.71
CST-13	PAST	CheckStrandSpeedLength	55,029	1.55

B. RQ2 – RICC Scalability

For evaluating the scalability and applicability of the checking approach to a real-world SoS we used our monitoring infrastructure and the PAS simulation environment to perform a ten-hour evaluation run. Not all constraints we defined can be monitored in the provided simulator, because it cannot run all required systems. We thus selected 13 constraints (cf. Table I; details are not shown due to non-disclosure agreements) to evaluate the scalability of the RICC as described below. While this might seem like a small number, the number of checks (almost 70k) performed on instantiated constraints based on almost 500k events still allows us to demonstrate scalability.

The goal of our evaluation was to investigate whether the constraint checker can handle a realistic number of events without significant increases in execution time and memory usage. We also assessed the capabilities for dynamically adding and removing constraints by incrementally adding constraints and later again deactivating them.

We started with five active constraints (CST-01–CST-05) and added added four constraints (CST-06–CST-09) after one hour, followed by four more constraints (CST-10–CST-13) after another hour. The evaluation then continued for six hours with the full set of constraints active. After eight respectively nine hours we again reduced the number of active constraints by removing CST-01–CST-05 and CST-06–CST-09. The simulation environment and the monitoring infrastructure were set up on a standard Desktop machine with an Intel(R) Core(TM) i5 CPU @2.60GHz 16GB RAM running Windows 7 64-Bit.

We measured the number of events that occurred, the number of constraints instantiated from the 13 constraint definitions (i.e., the checks performed), the (median) evaluation time for each constraint (in ms), the maximum number of constraint instances alive at a certain point in time, and the memory consumption of the constraint checker (in MB). During the ten hours of the simulation run, 482,841 events and their related data were captured and 68,795 constraints checks were performed resulting in an average of 115 checks per minute.

For each constraint instance we measured the time required for executing the method, which checks the constraint and generates violations in case of errors. The goal was to assess whether the number of active constraints negatively influences the evaluation time.

The median evaluation time for a constraint instance ranges from 0.37 ms for CST-08 up to 54 ms for CST-03. Fig. 5 provides an overview of the evaluation times during the simulation grouped by the different constraint types. The comparably high median evaluation times for data constraints CST-03, CST-04, and CST-11 can be explained by the size and complexity of the data items that need to be checked for these constraints. Also there are a few outliers for constraints CST-05, CST-09, and CST-13 (details cf. Fig. 5(a)). Nevertheless, the evaluation time was always less than one second, which still allows to provide instant feedback to users. Also, the values indicate that for future occurrence constraints (Fig. 5(c)), past occurrence constraints (Fig. 5(b)), and data constraints (Fig. 5(d)) the number of active constraints does not at all influence the evaluation time.

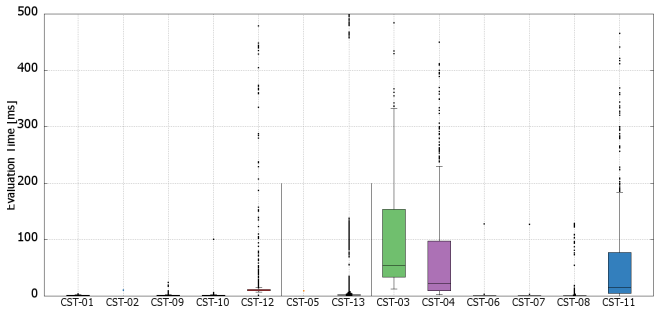
The maximum number of active constraint instances for a measure point ranges from 0 (for most constraints) to 137 for constraint CST-13. As described, a constraint instance is created only when needed and immediately destroyed after its evaluation. This is also confirmed by the memory usage of the Java Virtual machine running the constraint checker. In case of CST-13, due to its type, i.e., past occurrence, and the rather high number of instances that have to be evaluated, constraint instances are queued before they are destroyed. Still, despite this queuing the memory usage does not increase noticeably, remaining between 330 MB (lower quartile) and 412 MB (upper quartile) during the simulation run.

C. Discussion of Results and Threats to Validity

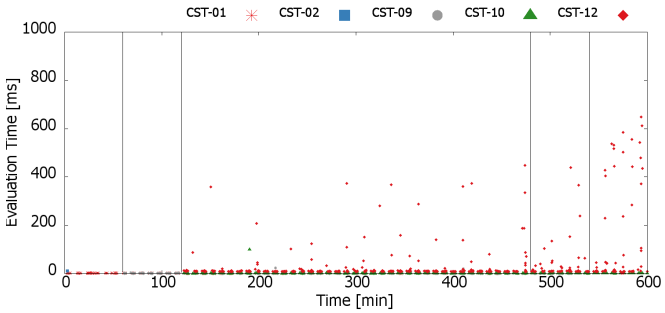
Our evaluation confirms that we could express all constraints elicited for the PAS SoS so far (RQ1) and that the underlying constraint checker was able to handle the high number of incoming events and perform all constraint checks fast enough to provide instant feedback (RQ2).

In terms of external validity, the results and findings are based on a single directed SoS in 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 constraint checker it is possible to consider additional constraint types if needed and adapt the constraint checker accordingly. Also, the requirements and constraints selected for the evaluation might not cover the full range of requirements existing in the industrial SoS. However, given the scale and complexity of the PAS we consider our evaluation a good starting point representing a realistic case. We plan to conduct further evaluations with different systems in the future.

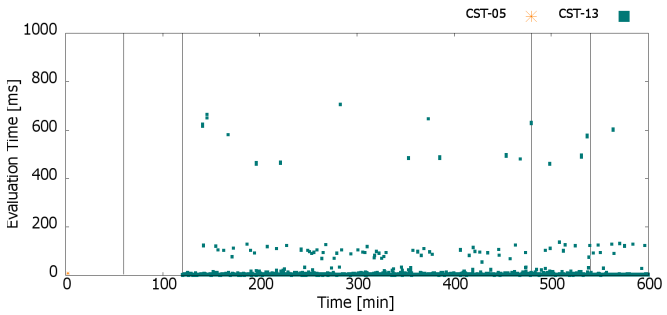
The evaluation focuses on the expressiveness of the DSL (RQ1) and on the scalability of the constraint checker (RQ2). We deliberately did not discuss end-user tools in detail as this is part of a separate study assessing the usefulness of the different tools and editors for writing and managing constraints. However, the constraints and the DSL



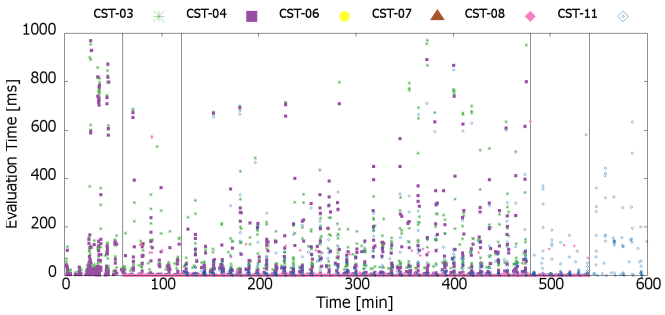
(a) Boxplots of evaluation times (in ms) for all constraints.



(b) Distribution of evaluation times for future occurrence constraints (CST-01, CST-02, CST-09, CST-10, CST-12)



(c) Distribution of evaluation times for past occurrence constraints (CST-05, CST-13)



(d) Distribution of evaluation times for data constraints (CST-03, CST-04, CST-06, CST-07, CST-08, CST-11)

Fig. 5: Overview of median evaluation times for all constraints (a) and distributions of evaluation times separately for future occurrence (b), past occurrence (c), and data constraints (d) over the ten-hour simulation run. We activated constraints after one and after two hours and deactivated them again after eight and nine hours as shown by the vertical lines.

were developed together with engineers of our industry partner leading to rapid feedback, which resulted in several adaptations and improvements during the development process.

Regarding the rather small number of constraints used in our scalability evaluation (13), the number of constraint instances created at runtime and the number of constraint checks performed have a much higher impact on the scalability of the checker. We demonstrated in our evaluation that even for constraints leading to many instances and checks (cf. CST-13) our approach ensures immediate feedback.

Our evaluation did not measure the overhead caused by the probes instrumenting the system. We also did not consider the performance of the monitoring infrastructure, which is not directly part of the constraint checking mechanism and running on a separate server. However, earlier evaluations [24] confirm that the underlying infrastructure is capable of handling a high amount of events. Events used for checking constraints in our evaluation are provided by probes using AspectJ (http://eclipse.org/aspectj/). A separate evaluation we conducted shows 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 that are later checked in a constraint. However, the probes used in our evaluation are small, atomic code fragments only collecting specific data in the SoS, and serializing complex objects was rather the exception. Furthermore, all additional processing and analysis tasks are performed independently on a separate machine to keep the impact on the monitored systems low.

VII. Lessons Learned

We summarize experiences and lessons learned of developing our DSL-based checker, which may be useful for researchers and practitioners working on similar challenges.

Systems of systems require an iterative language design. The heterogeneous systems in an SoS and the diverse teams involved in their development and maintenance make it difficult to develop a language addressing the different needs. When developing our constraint DSL we started with interviewing different teams responsible for different systems to identify common requirements for the language. We showed each version of the DSL to these teams and refined it according to their comments. This iterative language design helped to come up with a solution that is acceptable for its users.

Keep the YAGNI ("you aren't gonna need it") principle in mind when developing a DSL. When starting our collaboration with Primetals Technologies we analyzed diverse existing constraint languages and formal notations used for defining the expected behavior of a running system. Although we found several of these approaches quite appealing the reaction of our industry partner was different. This was mainly because the languages provided too many features not needed for their context and defining constraints was regarded as difficult using formal notations. They wanted to define the constraints as close as possible to natural language and liked the structured prose technique we proposed to them. A key lesson thus is to keep a language as simple as possible and to cover only what is necessary for the concrete context. The constraint language

should be oriented towards end users and hide the complexities of the underlying constraint checker.

Simplify and automate extending the DSL. While keeping the constraint language simple definitely makes sense for practical use, new types of constraints and checks might still be needed. For example, while we started with support for simple range checks for data constraints with primitive data objects, later the need arose for more complex checks of data objects and their relations. It is thus essential to allow extending the language at any time. We thus employed technologies such as XText and XTend, which make it easy to compile to a target language like Java from a DSL and to generate end-user support for a DSL based on a grammar.

Keep the mapping of the constraint DSL to the constraint checker flexible. During our project we also learned that it might be required to exchange the underlying checker, e.g., when the number of events to be monitored becomes too high impacting the checker's performance too much. So far our Java-based incremental checker worked fine as demonstrated in the evaluation. However, future needs might require switching to a checker providing higher performance for certain types of constraints. It is thus advisable to define a clear interface between the language and the checker to allow replacing both. For instance, while keeping our simple DSL, one could replace our checker with another checker that supports checking temporal and data constraints. Only the compilation of our constraints to the target language of the checker would then have to be changed.

Support dynamic constraint management in runtime monitoring. Industrial scenarios demonstrate the need to add new or modify existing constraints even while the system and the monitoring infrastructure are running, e.g., to investigate an unforeseen issue. Providing a dynamic approach is thus needed. A positive side-effect is also the performance of the constraint checker, which dynamically instantiates constraints and immediately destroys constraints after their evaluation results have been stored.

VIII. Related Work

Different research communities have been developing approaches for monitoring systems to detect violations of requirements at runtime. Examples include requirements monitoring [3], [4], performance monitoring, complex event processing [6], or runtime verification [7], [8].

Requirements monitoring approaches aim at determining the compliance of a system with its requirements during runtime [3], [35]. Monitors are used to detect requirements violations and serve as a starting point for revealing the root cause of problems. For example, Robinson [4] has proposed the ReqMon framework including a language for defining requirements and tools supporting different user roles during monitoring. ReqMon supports formalizing high-level goals, requirements, and their monitors. It automates generating and deploying monitors and provides traceability between high-level descriptions and lower-level runtime events.

Performance and event monitoring approaches focus on monitoring specific aspects of a running system, e.g., related with performance. Bubak et al. [36], for instance, propose the J-OCM approach supporting programmers in developing monitoring tools for distributed Java applications at the virtual machine level by providing uniform and extensible monitoring facilities for communication between different components. Their work is based on the Online Monitoring Interface Specification (OMIS) [37] providing a standardized interface between monitoring tools and the systems to be monitored. Van Hoorn et al. [21] describe Kieker, an extensible framework for event and application performance monitoring. The framework provides capabilities for system instrumentation, event recording, event processing, and basic visualization.

Requirements monitoring and event monitoring approaches provide a solid basis for monitoring specific aspects of single systems – e.g., performance, data-flow, or system communication – but are limited regarding the diversity of constraints and support for constraint checking in SoS architectures.

Complex event processing (CEP) [33] is an approach for monitoring arbitrary business processes. It aims at combining event streams gathered from multiple sources to infer events or patterns of events. Event patterns are typically described by implementing rules in some higher programming language or in an Event Description Language. While we are not focusing on monitoring business processes, CEP makes use of some common concepts that are related with our work, i.e., event description languages can be seen as DSLs useful to describe constraints to be monitored.

Furthermore, in the domain of *runtime verification* various approaches have been proposed to support monitoring and verifying system properties. For instance, Calinescu et al. [7] propose a three-staged process of monitoring, analysis, and planning. A system model is verified to detect violations of requirements. Ghezzi et al. [8] present the SPY@RUNTIME approach that relies on behavior models which are represented by finite state automata. An initial model is inferred in a setup phase and then used at runtime to detect changes.

In these and other domains, a wide variety of different *constraint languages* exist for defining requirements, system properties, or desired event sequences. For example, Spanoudakis et al. [38] present SERENITY, a framework for monitoring security 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 et al. [9] 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 [39] for Java programs). Baresi et al. [15] 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. [40] 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. [41] present a property-driven approach for runtime monitoring. A property meta-model 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. [42] present an approach for monitoring security properties. They use a security specification language called ConSpec to describe automata for security requirements.

Existing constraint languages, however, do not support all three challenges in SoS monitoring, i.e., coping with the diversity of constraints in large-scale systems, supporting the incremental definition and runtime management of constraints, and providing end-user support for constraint definition.

Existing work also often uses *temporal logic* to support monitoring software systems. Several authors have shown the expressiveness and usefulness of such formalisms [10]–[12]. However, these approaches usually operate on event traces and do not support the incremental definition and runtime management of constraints. Also, they are often difficult to use by industrial end users, at least in our experience.

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. [43] in the area of Web services and by Voelter and Visser [44] in the area of product line engineering. Hermans et al. 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. Our evaluation focused on the expressiveness of the constraint DSL and the scalability of the checker. We have also discussed the reusability of our DSL, i.e., that it is flexible and can be adapted an automated manner. Assessing productivity and reliability requires longitudinal studies, which we plan to conduct as part of our future work. Also, we plan to assess usability and learnability in studies with our industry partner. In this regard, we will consider the experimental evaluations performed for PROPEL [45], an approach guiding users through the process of defining formal property specifications. Similar to [45], we also plan to assess the usefulness of our DSL for defining requirements to be used for runtime monitoring.

IX. Conclusions and Future Work

Based on a scenario of monitoring an industrial system of systems we derived challenges regarding the definition and checking of constraints at runtime. Existing languages and checkers cannot easily be applied in an SoS context due to the diversity of constraints required, the need to incrementally define and manage constraints, and the required end-user support. We have described a constraint DSL for industrial end users and an incremental checker we have been developing to address these challenges.

Our approach supports the incremental checking of constraints at runtime and provides live and instant feedback to users. Our checker also supports the incremental definition and dynamic deployment of constraints at runtime, without stopping the checker and the monitored system. We evaluated the expressiveness of our constraint DSL using real constraints from our industrial case and the scalability of our checker in an industrial monitoring scenario. Our experiences suggest designing a constraint DSL in an iterative manner and keeping it as simple as possible. We also learned to keep the mapping of the DSL to the checking engine flexible to allow switching underlying checking technologies.

In our future work we aim to perform a usability assessment of our tools – particularly the constraint DSL editor – involving industrial end users. We further want to apply our approach to systems in other domains and in longitudinal studies. Furthermore, we are currently working on a framework to support the systematic comparison of existing constraint languages and formalisms regarding their usefulness for particular monitoring scenarios.

Acknowledgments

This work has been supported by the Christian Doppler Forschungsgesellschaft, Austria and Primetals Technologies. We particularly want to thank Christian Danner, Klaus Seyerlehner, Stefan Wallner, and Helmut Zeisel for their feedback and support. Special thanks go to Benedikt Aumayr for developing the first version of the constraint DSL and checker.

References

[1] M. W. Maier, "Architecting principles for systems-of-systems," *Systems Engineering*, vol. 1, no. 4, pp. 267–284, 1998.

[2] M. Dwyer, G. Avrunin, and J. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 1999 Int'l Conference on Software Engineering*. IEEE, May 1999, pp. 411–420.

[3] N. Maiden, "Monitoring our requirements," *IEEE Software*, vol. 30, no. 1, pp. 16–17, 2013.

[4] W. N. Robinson, "A requirements monitoring framework for enterprise systems," *Requirements Engineering*, vol. 11, no. 1, pp. 17–41, 2006.

[5] H. Muccini, A. Polini, F. Ricci, and A. Bertolino, "Monitoring architectural properties in dynamic component-based systems," in *Component-Based Software Engineering, LNCS 4608*. Springer, 2007, pp. 124–139.

[6] M. Völz, B. Koldehofe, and K. Rothermel, "Supporting strong reliability for distributed complex event processing systems," in *13th Int'l Conference on High Performance Computing & Communication, Banff, Canada*. IEEE, 2011, pp. 477–486.

[7] R. Calinescu, C. Ghezzi, M. Z. Kwiatkowska, and R. Mirandola, "Self-adaptive software needs quantitative verification at runtime," *Communications of the ACM*, vol. 55, no. 9, pp. 69–77, 2012.

[8] C. Ghezzi, A. Mocci, and M. Sangiorgio, "Runtime monitoring of component changes with Spy@Runtime," in *34th Int'l Conference on Software Engineering*, Zurich, Switzerland, 2012, pp. 1403–1406.

[9] M. Viswanathan and M. Kim, "Foundations for the run-time monitoring of reactive systems – fundamentals of the MaC language," in *Theoretical Aspects of Computing, 1st Int'l ICTAC Colloquium, (Revised Selected Papers)*, ser. Lecture Notes in Computer Science 3407, Z. Liu and K. Araki, Eds. Springer, 2005, pp. 543–556.

[10] F. Chen, M. d'Amorim, and G. Roşu, "A formal monitoring-based framework for software development and analysis," in *Formal Methods and Software Engineering*. Springer, 2004, pp. 357–372.

[11] H. Gunadi and A. Tiu, "Efficient runtime monitoring with metric temporal logic: A case study in the Android operating system," in *Proceedings Formal Methods 2014 (FM'14)*. Springer, 2014, pp. 296–311.

[12] A. Bauer, M. Leucker, and C. Schallhart, "Monitoring of real-time properties," in *Foundations of Software Technology and Theoretical Computer Science (FSTTCS'06)*. Springer, 2006, pp. 260–272.

[13] A. Paschke, "RBSLA – a declarative rule-based service level agreement language based on RuleML," in *Int'l Conference on Computational Intelligence for Modelling, Control and Automation and Int'l Conference on Intelligent Agents, Web Technologies and Internet Commerce*, vol. 2. IEEE, 2005, pp. 308–314.

[14] W. Robinson, "Extended OCL for goal monitoring," *Electronic Communication of the European Association of Software Science and Technology*, vol. 9, no. 1, pp. 1–12, 2008.

[15] L. Baresi and S. Guinea, "Event-based multi-level service monitoring," in *Proceedings 20th Int'l Conference on Web Services (ICWS)*. IEEE, 2013, pp. 83–90.

[16] H. Phan, G. S. Avrunin, and L. A. Clarke, "Considering the exceptional: Incorporating exceptions into property specifications," *Department of Computer Science, University of Massachusetts, Amherst, MA*, vol. 1003, 2008.

[17] M. Mansouri-Samani and M. Sloman, "Monitoring distributed systems," *IEEE Network*, vol. 7, no. 6, pp. 20–30, 1993.

[18] T. Bures, P. Hnetynka, and F. Plasil, "Strengthening architectures of smart CPS by modeling them as runtime product-lines," in *Proceedings of the 17th Int'l ACM Sigsoft Symposium on Component-based Software Engineering (CBSE'14)*. ACM, 2014, pp. 91–96.

[19] C. Jeffery, M. Auguston, and S. Underwood, "Towards fully automatic execution monitoring," in *Radical Innovations of Software and Systems Engineering in the Future*. Springer, 2004, pp. 204–218.

[20] K. Havelund and G. Roşu, "Efficient monitoring of safety properties," *Int'l Journal on Software Tools for Technology Transfer*, vol. 6, no. 2, pp. 158–173, 2004.

[21] A. van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *Proceedings 3rd Joint Int'l Conference on Performance Engineering*, 2012, pp. 247–248.

[22] A. Egyed, "Instant consistency checking for the UML," in *Proceedings of the 28th Int'l Conference on Software Engineering*. ACM, 2006, pp. 381–390.

[23] M. Vierhauser, P. Grünbacher, A. Egyed, R. Rabiser, and W. Heider, "Flexible and scalable consistency checking on product line variability models," in *Proceedings Int'l Conference on Automated Software Engineering*, 2010, pp. 63–72.

[24] M. Vierhauser, R. Rabiser, P. Grünbacher, K. Seyerlehner, S. Wallner, and H. Zeisel, "ReMinds: A flexible runtime monitoring framework for systems of systems," *Journal of Systems and Software*, 2015.

[25] M. Vierhauser, R. Rabiser, and P. Grünbacher, "A case study on testing, commissioning, and operation of very-large-scale software systems," in *36th Int'l Conference on Software Engineering*. Hyderabad, India: ACM, 2014, pp. 125–134.

[26] R. Rabiser, M. Vierhauser, and P. Grünbacher, "Variability management for a runtime monitoring infrastructure," in *Proceedings of the Ninth Int'l Workshop on Variability Modelling of Software-intensive Systems, VaMoS '15, Hildesheim, Germany, January 21-23, 2015*, 2015, p. 35.

[27] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[28] J. Skene and W. Emmerich, "Engineering runtime requirements-monitoring systems using MDA technologies," in *Trustworthy Global Computing*. Springer, 2005, pp. 319–333.

[29] J. Whittle, P. Sawyer, N. Bencomo, B. Cheng, and J.-M. Bruel, "RELAX: a language to address uncertainty in self-adaptive systems requirement," *Requirements Engineering*, vol. 15, no. 2, pp. 177–196, 2010.

[30] P. Zhang, B. Li, H. Muccini, and M. Sun, "An approach to monitor scenario-based temporal properties in web service compositions," in *Advanced Web and Network Technologies, and Applications*. Springer, 2008, pp. 144–154.

[31] K. Kiviluoma, J. Koskinen, and T. Mikkonen, "Run-time monitoring of architecturally significant behaviors using behavioral profiles and aspects," in *Proceedings of the 2006 Int'l Symposium on Software Testing and Analysis*. ACM, 2006, pp. 181–190.

[32] L. Baresi, S. Guinea, O. Nano, and G. Spanoudakis, "Comprehensive monitoring of BPEL processes," *IEEE Internet Computing*, vol. 14, no. 3, pp. 50–57, 2010.

[33] D. C. Luckham, *Event processing for business: Organizing the real-time enterprise*. John Wiley & Sons, 2011.

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

[35] S. Fickas and M. S. Feather, "Requirements monitoring in dynamic environments," in *2nd IEEE Int'l Symposium on Requirements Engineering, York, England*, 1995, pp. 140–147.

[36] M. Bubak, W. Funika, M. Smetek, Z. Kiliański, and R. Wismüller, "Event handling in the J-OCM monitoring system," in *Parallel Processing and Applied Mathematics*. Springer, 2004, pp. 344–351.

[37] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode, "OMIS: Online Monitoring Interface Specification (v. 2.0)," Technische Universität München, Tech. Rep. TUM-I9733, 1997.

[38] G. Spanoudakis, C. Kloukinas, and K. Mahbub, "The SERENITY runtime monitoring framework," in *Security and Dependability for Ambient Intelligence*. Springer, 2009, pp. 213–237.

[39] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky, "JavaMaC: A run-time assurance approach for Java programs," *Formal Methods in System Design*, vol. 24, no. 2, pp. 129–155, 2004.

[40] M. Montali, F. M. Maggi, F. Chesani, P. Mello, and W. M. van der Aalst, "Monitoring business constraints with the event calculus," *ACM Trans. Intell. Syst. Technol.*, vol. 5, no. 1, pp. 17:1–17:30, Jan. 2014.

[41] A. Bertolino, A. Calabrò, F. Lonetti, A. Di Marco, and A. Sabetta, "Towards a model-driven infrastructure for runtime monitoring," in *Software Engineering for Resilient Systems*. Springer, 2011, pp. 130–144.

[42] I. Aktug, M. Dam, and D. Gurov, "Provably correct runtime monitoring," in *Formal Methods (FM'08)*. Springer, 2008, pp. 262–277.

[43] F. Hermans, M. Pinzger, and A. van Deursen, "Domain-specific languages in practice: A user study on the success factors," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science 5795, A. Schürr and B. Selic, Eds. Springer Berlin Heidelberg, 2009, pp. 423–437.

[44] M. Voelter and E. Visser, "Product line engineering using domain-specific languages," in *15th Int'l Software Product Line Conference (SPLC 2011)*. Munich, Germany: IEEE CS, 2011, pp. 70–79.

[45] R. L. Cobleigh, G. S. Avrunin, and L. A. Clarke, "User guidance for creating precise and accessible property specifications," in *Proceedings of the 14th ACM SIGSOFT Int'l Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. ACM, 2006, pp. 208–218.