A Requirements Monitoring Model for Systems of Systems

Michael Vierhauser Rick Rabiser Paul Grünbacher Benedikt Aumayr
Christian Doppler Laboratory MEVSS
Johannes Kepler University Linz, Austria
michael.vierhauser@jku.at

***Abstract*—Many software systems today can be characterized as systems of systems (SoS) comprising interrelated and heterogeneous systems developed by diverse teams over many years. Due to their scale, complexity, and heterogeneity engineers face significant challenges when determining the compliance of SoS with their requirements. Requirements monitoring approaches are a viable solution for checking system properties at runtime. However, existing approaches do not adequately consider the characteristics of SoS: different types of requirements exist at different levels and across different systems; requirements are maintained by different stakeholders; and systems are implemented using diverse technologies. This paper describes a three-dimensional requirements monitoring model (RMM) for SoS providing the following contributions: (i) our approach allows modeling the monitoring scopes of requirements with respect to the SoS architecture; (ii) it employs event models to abstract from different technologies and systems to be monitored; and (iii) it supports instantiating the RMM at runtime depending on the actual SoS configuration. To evaluate the feasibility of our approach we created a RMM for a real-world SoS from the automation software domain. We evaluated the model by instantiating it using an existing monitoring framework and a simulator running parts of this SoS. The results indicate that the model is sufficiently expressive to support monitoring SoS requirements of a directed SoS. It further facilitates diagnosis by discovering violations of requirements across different levels and systems in realistic monitoring scenarios.**

***Index Terms*—System of systems, requirements monitoring.**

I. Introduction

Complex software-intensive systems are often described as systems of systems comprising heterogeneous 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], [2], [3]. Dahmann and Baldwin [4] distinguish between directed SoS, which are centrally managed; collaborative SoS, in which the systems collaborate voluntarily to fulfill the agreed purposes; virtual SoS with little or no central management authority and highly emergent behavior; as well as acknowledged SoS, which share a common management and resources but remain in independent ownership. As the full behavior of SoS only emerges during operation, system testing is not sufficient to determine compliance with requirements.

Different research communities have been developing approaches for monitoring systems to detect violations of requirements at runtime. Examples include requirements monitoring [5], [6], monitoring of architectural properties [7], complex event processing [8], or runtime verification [9] to name but a few. However, scaling up existing software engineering approaches to large and heterogeneous systems can lead to problems [2], [3]. In particular, SoS imply significant challenges for requirements monitoring: requirements in the SoS are owned by diverse stakeholders and exist at different levels, for different systems, and in different artifacts. They are often overlapping, conflicting, or cross-cutting and it is not always possible to allocate them to specific systems or components. Also, requirements dependencies between systems collaborating in an SoS need to be managed. SoS further operate in a complex environment and interact with third-party and legacy systems. Requirements monitoring for SoS thus needs to support different monitoring scopes regarding the SoS architecture and need to abstract from different system-specific technologies used in an SoS. Also, requirements need to exist not only at design time but as runtime entities [10] to support monitoring different SoS instances.

This paper introduces a three-dimensional requirements monitoring model (RMM) for SoS covering requirements, monitoring scopes, as well as events monitored at runtime (cf. Fig. 1). Each dimension incorporates and extends concepts from existing research: for instance, the SoS monitoring scopes refer to concepts from software architecture modeling [11]. A scope represents one or more SoS systems, components, or connectors that need to be monitored. The requirements dimension covers requirements represented as constraints for checking SoS behavior as in existing requirements monitoring approaches [6]. The events dimension builds on approaches for event-based analysis [12]. Probes are developed to instrument an SoS at different levels and for different systems [13] to provide events and data to an event model. Our approach particularly addresses the dependencies between these three dimensions: each requirement is allocated to a specific scope in the SoS architecture. Requirements are checked at runtime based on the event model. Each probe instrumenting the SoS is assigned to a scope in the architecture to support diagnosis in case of violations.

The paper claims the following contributions: (i) our RMM for SoS considers the scope of requirements to be monitored in SoS with respect to the SoS architecture; (ii) it uses event

978-1-4673-6905-3/15/$31.00 © 2015 IEEE

RE 2015, Ottawa, ON, Canada
Research Paper

models to abstract from heterogeneous technologies in SoS; (iii) the RMM is instantiated at runtime based on the actual SoS configuration. Our model builds on our earlier work: in a research preview paper [14] we briefly summarized the industrial motivation and required infrastructure capabilities for requirements monitoring of SoS. In a follow-up paper, we presented a framework for supporting system instrumentation and collecting events in SoS architectures together with a performance evaluation [15]. We also discussed how SoS variability is managed using our framework [16].

In the remainder of this paper, we first describe the background and challenges of requirements monitoring for SoS and present an industrial case. We then introduce our three-dimensional RMM and describe its definition and runtime instantiation using real-world examples. We further describe the implementation based on an existing monitoring framework and report results of a two-stage evaluation using an industrial SoS. We conclude with a discussion of related work and an outlook on future work.

II. Requirements Monitoring Challenges in SoS

Requirements monitoring approaches allow determining compliance of a system with its requirements during runtime [17], [5], [6]. Monitors are used to detect and analyze violations of requirements. However, the characteristics of SoS [1], [2], [4] impose specific challenges for requirements monitoring not covered by existing work that we aim to address with our approach:

(A) Different scopes of requirements. Existing requirements monitoring approaches (e.g., [18]) assume that a single system is monitored and do not support different monitoring scopes, an important issue in SoS environments. For instance, the constituent systems of an SoS are operationally independent, i.e., they are only weakly integrated and often have not been designed with their interaction in mind. Monitoring requirements of an SoS thus requires instrumenting different systems and their interactions at multiple levels. Depending on the granularity of the requirements to be checked their monitoring scope can cover individual components, specific systems, system interactions, or system-wide properties. The monitoring scope of a requirement is particularly vital for diagnosing and isolating the root cause of violations.

(B) Technological diversity. SoS architectures consist of various systems implemented by different teams or even companies. Due to this diversity, monitoring support needs to be agnostic regarding specific implementation technologies. For instance, while it might be comparably easy to instrument and monitor a Java-based system using aspect-oriented techniques, it is often much harder to instrument legacy systems. Monitoring the interfaces between different systems and different levels of the SoS is particularly challenging due to the diversity of connectors [19]. Instrumenting these different sources of information requires the development of domain-specific probes supporting different technologies and architectural styles. While existing monitoring approaches often focus on particular technologies such as Java [18] or service-oriented architectures [20], a unified representation of runtime information collected from different systems and their interactions is missing. Existing work on complex event processing [8] and event-based analysis [12] provides a good starting point but needs to be extended to deal with the technological heterogeneity in SoS.

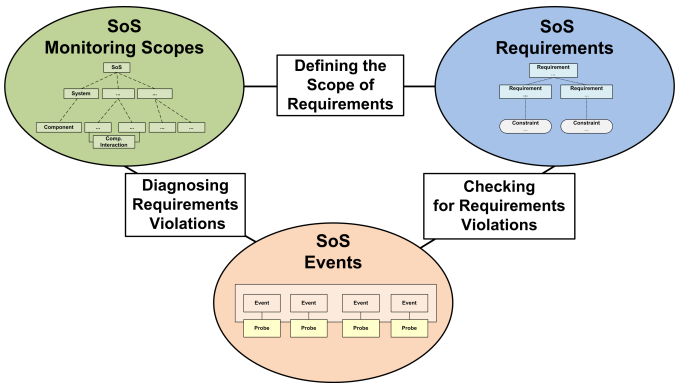


Fig. 1: Requirements Monitoring Model for SoS.

(C) Diagnosis across systems. The managerial independence and stakeholder diversity in SoS means that systems are developed, maintained, and put into operation by independent teams, often even by multiple different companies. SoS thus represent a situation where a key assumption of requirements engineering – the system to be designed is under the control of a single stakeholder who determines a consistent set of requirements – no longer holds [21]. When violations of requirements occur in a running system, engineers and service personnel need support to track the root causes of violations within the SoS architecture.

III. Industrial Case

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.

Primetals Technologies' Plant Automation System (PAS) is a directed SoS comprising centrally managed systems for automating, tracking, and optimizing different stages of iron and steel production. The PAS consists of several independent yet interrelated systems (cf. Fig. 2) sizing up to several million lines of code. These systems have different architectures and communicate via defined interfaces. They have been developed independently by different teams using different technologies as a highly specific technical background and domain knowledge is required for each stage of the metallurgical production process. Specifically, the PAS comprises systems for automating the production of iron (melting ore and raw materials) [22], steel (refining liquid iron and other materials), and steel slabs (casting liquid steel) [23]. The Iron, Steel, and Caster systems share common goals and concepts but are quite diverse with respect to their architectural styles and

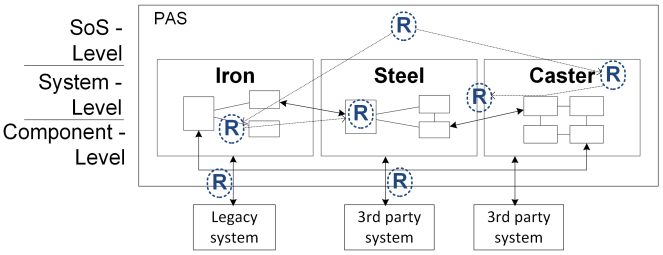


Fig. 2: SoS requirements (R) address different levels, different systems and components, and diverse interactions.

technologies (challenge B). Even though the PAS requirements are carefully managed during development, it is crucial to monitor them after deploying the system to detect inaccurate and erroneous behavior at runtime [24], in particular after upgrading certain components or systems.

Although the software systems are engineered independently, there are manifold dependencies resulting from the metallurgical process that need to be considered when planning their joint operation. Liquid iron is needed for producing liquid steel, which is then used to cast solid steel slabs. As a result the requirements of the automation software systems are often cross-cutting and interrelated. For example, requirements concerning ladle handling in the Caster system have dependencies to requirements on finishing ladles in the Steel system. The PAS is further connected to legacy or third-party systems leading to additional complexity. For instance, the Iron and Steel systems typically rely on input from lab systems providing material analyses. However, there are also dependencies between requirements within one particular system: for instance, a component in the Caster optimizing the arrangement of slabs on a strand to minimize scrap and to ensure steel quality, relies on information from several other components. Requirements monitoring thus has to consider the different scopes of requirements in the SoS (challenge A).

While many tasks in plants are performed automatically, metallurgical production also relies on input from human operators. This can have unforeseen effects on the automation software. For instance, the cut length of steel slabs defined by an operator of the Caster system might conflict with some required plan characteristics from the production planning system. Dependencies between requirements at machine level and automation level can lead to further issues, which need to be detected and diagnosed at runtime (challenge C).

IV. Requirements Monitoring Model for SoS

We propose a RMM addressing the challenges described above to facilitate requirements monitoring in an SoS context. Our proposed approach distinguishes three dimensions (cf. Fig. 3): monitoring scopes for relating requirements with the SoS architecture, requirements formalized as constraints to allow their validation, and events collected in event models to manage runtime information collected from the SoS by diverse probes. We use examples from Primetals

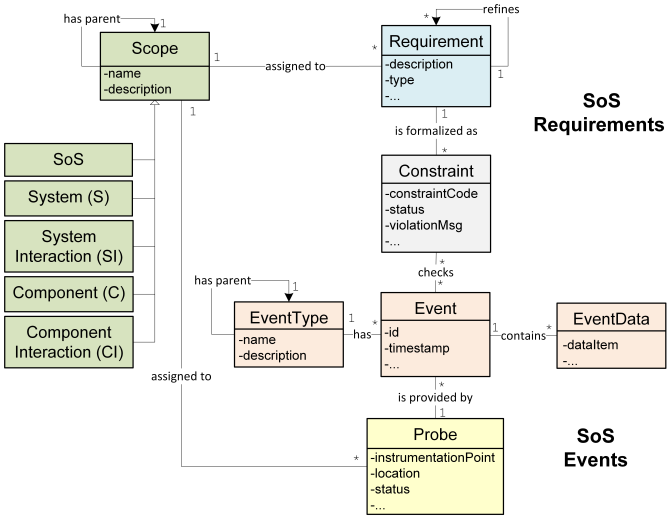


Fig. 3: SoS Requirements Monitoring Model.

Technologies' PAS to illustrate our model (cf. Fig. 4 and Table I).

A. Monitoring Scopes

A *Monitoring Scope* is an area of interest to be monitored with respect to an SoS architecture. For instance, a scope may represent a particular system, one or more components, or a connector (such as interfaces or APIs) between different parts of an SoS.

We distinguish five different types of scopes that are instantiated for a concrete SoS: the type *SoS* represents the root of the hierarchical scope model, i.e., it concerns SoS-wide properties or behavior that cannot be contributed to a single system. This is the case, e.g., for the requirement *PAS-OneTap*, as taps are relevant in several systems of the PAS. The scope *System* is used if requirements regard one particular system in the SoS. The PAS scope model, for instance, defines the System scope types *Iron*, *Steel*, *Caster*, and *Lab System*. The scope type *Component* addresses specific components providing functionality limited to a specific system of the SoS (e.g., scope *Discharge*) or capabilities grouped together to indicate team responsibility (e.g., scope *HMI: human machine interface*). The granularity of a scope type thus depends strongly on the SoS architecture and to a certain extent on the organizational structure of the SoS. Furthermore, the *System Interaction* scope addresses the communication of systems in the SoS (e.g., *L2I* covers the interaction of Lab and Iron) whereas the *Component Interaction* scope addresses the interaction of components. For instance, *H2A* addresses the interaction of the Caster HMI and the Caster Archiving components.

B. Requirements

A *Requirement* describes a functionality, a property, or a behavior of the SoS to be monitored at runtime. We assume that only a subset of the requirements in the SoS are considered as relevant for monitoring and inclusion in the RMM.

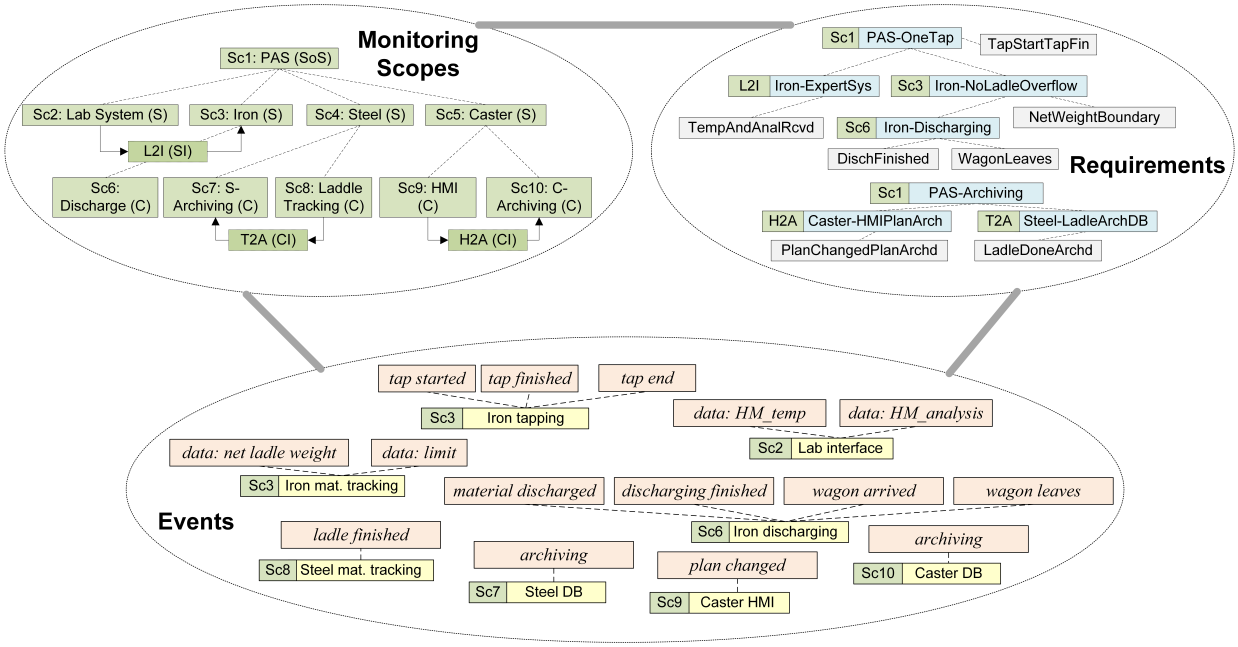


Fig. 4: An example of a Requirements Monitoring Model for Primetals Technologies' SoS. Scope ids are used to indicate references among the dimensions. Table I provides details on requirements, constraints, and their relation to events.

For example, an analyst might decide that the requirement specifying the tap strategy of a plant needs to be monitored.

Requirements may need to be refined before they can be formalized as a constraint or assigned to a particular scope. In our example (cf. Fig. 4) the SoS requirement *PAS-Archiving* is refined into the requirements *SteelLadleArchiveDB* for the Iron system and *CasterHMIPlanArchiving* for the Caster system. Both requirements are related with a requirement stating that changes performed by the operator (e.g., in an HMI) must be captured and persisted in the systems' databases. In case of the Steel system information on the finished ladles needs to be stored, whereas for the Caster system manual changes of the casting plan need to be archived. As both requirements address more than a single scope, they are assigned to a Component Interaction scope covering the communication of the operator components and the archiving components of Steel and Caster, i.e., *T2A* and *H2A*.

Modelers define *Constraints* to formalize the requirements and to allow checking them at runtime based on events and event data. A Constraint comprises a status indicating whether it is active or violated, a message to notify users in case of violations, and the actual source code defining the conditions to be checked. Our model is not limited to specific types of constraints as long as they are related with events and event data provided by probes. Modelers can use a constraint language of their choice depending on the checks that need to be supported. For example, the system requirement *Iron-NoLadleOverflow* requires the definition of an invariant-type constraint *NetWeightBoundary* that checks if *net ladle weight* $<$ *limit*. Other examples (see Table I) require the definition of timing constraints regarding particular event sequences and orders of events. We briefly describe our solution for constraint definition and checking in the implementation section.

C. Events

A uniform data representation is essential to allow reasoning across different systems in an SoS. We thus use event models defining the types of events collected by probes from different locations within the SoS as part of the RMM.

An *Event* is a relevant system operation or interaction with the system that happens at a specific scope at a specific point in time. *Event Types* allow modelers to define a taxonomy for structuring and filtering events. Events often contain *Event Data* to capture information collected from different systems. Examples of data items range from complex data structures containing material analysis data or information on the current casting progress to PAS machine data or sensor values captured by low-level machine interface probes.

Events are provided by probes and used to check constraints. A *Probe* is a single and encapsulated unit that provides arbitrary information extracted or intercepted from a system under observation [13]. Probes depend on the system to be monitored and are mapped to a scope. The status indicates if a Probe is active and the location provides runtime information about the instrumented process. For example, the Iron system needs to be instrumented in several places to monitor the requirement *Iron-NoLadleOverflow* and to provide the necessary events and data for checking the constraint *NetWeightBoundary*. For instance, a probe for the material tracking of the Iron system is required that provides *ladle weighted* events together with the actual weight data. Also, several probes are required that provide *weight limit defined* events together with the actual limit values. Multiple probes are needed as the user of the

TABLE I: Selected PAS requirements and constraints.

Requirements	
PAS-OneTap	Only one tap is allowed at any given time
Iron-ExpertSys	Within x min after tap end, hot metal temperature and analysis must be available from the lab system
Iron-NoLadleOverflow	When a ladle is full and a tap is running, a new ladle must be started
Iron-Discharging	Wagons at tipping stations should only discharge if containing required material; discharging must finish within x min
PAS-Archiving	HMI changes must be archived for later analyses
Caster-HMIPlanArch	When the caster operator changes a plan, the plan change must be archived within x seconds
Steel-LadleArchDB	When the steel operator declares a ladle finished, this event must be archived within x minutes
Constraints	
TapStartTapFin	event *tap started* $\rightarrow$ event *tap finished*
TempAndAnalRcvd	event *tap end* $\rightarrow$ data HM_temp received AND data HM_analysis received within x minutes
NetWeightBoundary	net ladle weight $<$ limit
DischFinished	event *material discharged* $\rightarrow$ event d*ischarging finished* within x minutes
WagonLeaves	event *wagon arrived* $\rightarrow$ event *material discharged* OR event *wagon leaves*
PlanChangedPlanArchd	event p*lan changed* $\rightarrow$ event *archiving* within x seconds
LadleDoneArchd	event *ladle finished* $\rightarrow$ event *archiving* within x minutes

Iron system can either define the limit via the HMI or by changing a configuration file.

D. Dependencies

Links are established between requirements and scopes to ease the diagnosis of violated constraints and to enable tracing violations to their origin, i.e., the affected elements of the SoS architecture. Each constraint in the RMM is related with one or more events, which are needed to perform the constraint check. This dependency allows *checking for requirements violations*. More specifically, when a constraint representing a requirement is violated, the events causing the violation can be determined. Probes are also related to monitoring scopes, which supports *diagnosing violations* as for each event that led to a constraint violation the elements of the SoS architecture can be determined based on the probe. Please note that the scope related with the violated requirement is not necessarily the same as the scope related with the probe producing the events. For example, while the requirement *IronExpertSys* is related with the scope *L2I* to indicate it regards the interaction of the Lab System and the Iron System, the probe *Lab interface* instruments the Lab System, which contains the interface for communication with the Iron System (cf. Fig. 4).

V. Defining and Instantiating the RMM

Fig. 5 shows the steps of creating and evolving the RMM as well as populating the model at runtime in a monitoring infrastructure [15] based on an actual SoS configuration.

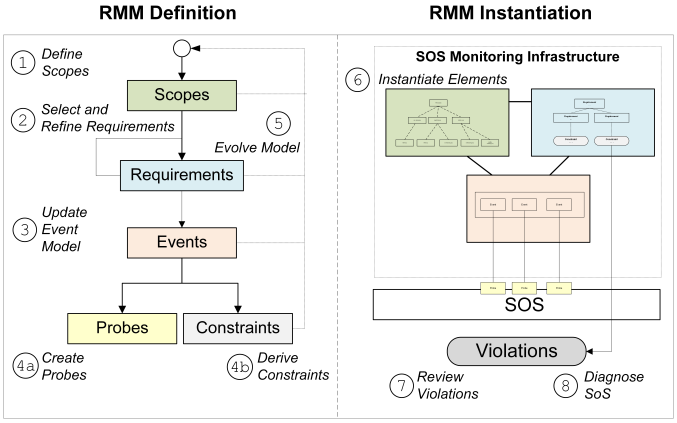


Fig. 5: Defining and instantiating the RMM.

A. Model Definition

The definition of the RMM comprises six key activities: (1) SoS architects *define the areas of interest* to be monitored, i.e., scopes and their interactions. A simple hierarchy of SoS, system, and component scopes allows to model the architectural view on the SoS from a monitoring perspective and to assign responsibilities. Existing architecture documentation of the SoS and its constituent systems can be reused. (2) System architects and analysts then *select requirements* to be monitored based on existing system specifications as well as domain experience, e.g., they may select requirements which frequently led to problem reports, system maintenance, or user support requests in the past. Coarse-grained requirements that cannot be monitored directly (cf. *PAS-archiving*) are refined into several requirements that can be assigned to particular monitoring scopes.

Based on the selected requirements (3) engineers *define the event types* to be monitored in an event model by specifying the structure and hierarchy of events, their possible attributes, and related data types. Depending on the size of the SoS, and the number and diversity of event types in particular, it can make sense to define several event models, e.g., one for each system in the SoS. Engineers can rely on pre-defined default event types such as system or user interface event or define arbitrary new event types. For example, when defining the *CasterHMIPlanArchiving* requirement, the engineer needs to define a new type *plan changed* representing the action of an operator changing a casting plan. Depending on the technology of the underlying system, (4a) engineers *implement probes* to provide the desired types of events specified in the event model. They also relate each probe with a particular monitoring scope depending on which part of the SoS they are instrumenting. (4b) Analysts then *derive constraints* to formalize the requirements. This task can be performed independently from the actual instrumentation of the systems, e.g., by project managers or domain experts familiar with the specific scope and characteristics to be checked. They further relate constraints with the events and data to be checked.

The RMM will hardly be created in a single step but (5) analysts will incrementally *evolve the model*. Monitoring thus can already start with just a few scopes, requirements, constraints, events, and probes. Over time, more elements can be added to check additional behavior at runtime or to address particular issues that occurred during system operation.

B. Model Instantiation

The RMM is automatically (6) *instantiated* by a monitoring infrastructure at runtime for a particular SoS. Instantiation starts with probes registering at the infrastructure and providing events to the runtime instance of the event model. A scope is instantiated if a probe assigned to it starts providing events and event data. The events generated by the probe are sent to the monitoring infrastructure which checks the constraints to determine violations of the requirements they represent. (7) The engineer can *review each violation*, i.e., data on the requirement/constraint that has been violated, its origin, as well as the involved events. (8) The analyst performs an initial *diagnosis of the system* by exploiting the trace links in the RMM, between requirements and monitoring scopes as well as between probes and monitoring scopes.

VI. Implementation

We implemented the RMM on top of an existing monitoring framework described in [15]. The different dimensions of the model are implemented in Java as separate, independent parts each providing an API for communication and establishing trace links. Our approach allows continuous changes to the model. New event types can be added and new probes can register to the monitoring infrastructure at any time. New requirements and additional or modified constraints can also be added without restarting the constraint engine.

Our mechanism for checking constraints is based on an incremental consistency checker [25], [26] that supports arbitrary DSLs for defining constraints. We extended the original checker to support runtime monitoring and developed a DSL for defining constraints on runtime events. However, the DSL itself is out of scope of this paper.

The tools for managing the framework infrastructure and the constraints, event models, and requirements are based on Eclipse RCP (http://www.eclipse.org/). The client-server monitoring infrastructure is implemented using Java, Java RMI, and JSON. Probe implementations vary in terms of the underlying system to be instrumented. Java systems are instrumented using AspectJ (http://eclipse.org/aspectj). Further probes are implemented in C++ and plain Java. We developed probe templates to ease instantiating frequently required types of probes. We implemented several prototype tools for our industry partner which are also used in the evaluation. For instance, the user is supported by various editors for defining requirements and constraints using our DSL as well as for managing scopes and event models. At runtime a graphical overview is provided of all scopes and their states (cf. Fig 6). While the RMM has to be created manually – based on existing system specifications and architecture models – its instantiation is automated by the monitoring infrastructure.

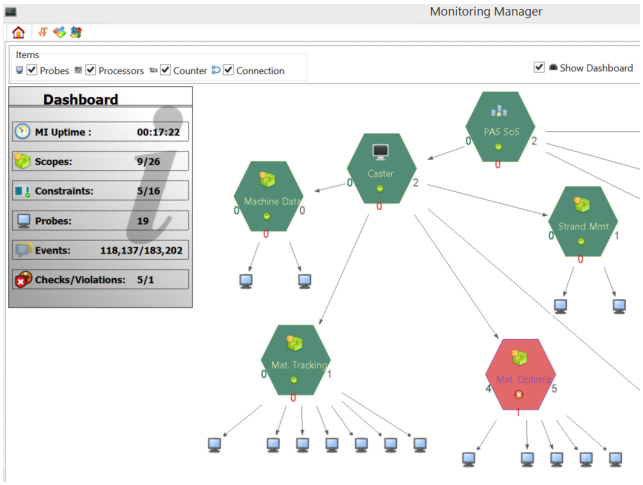


Fig. 6: The RMM tool manages scopes, requirements, and constraints to support runtime diagnosis.

VII. Evaluation

Our long-term cooperation with Primetals Technologies allowed us to evaluate our approach by applying the RMM to several systems in a realistic setting. We explore two research questions regarding the expressiveness and usefulness of our RMM in a typical requirements monitoring scenario based on Primetals Technologies' directed SoS.

A. Research Questions and Method

RQ1 – *is the approach sufficiently expressive to allow its use in a real-world industrial SoS?* – covers the definition of the RMM. We discussed monitoring needs together with Primetals Technologies 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 [24]. Furthermore, we studied specification documents and analyzed different systems part of the PAS. We then selected and defined requirements covering the different systems of the PAS, defined constraints, customized the event model, and created probes to experiment with monitoring real-world requirements.

RQ2 – *can the approach handle realistic SoS monitoring scenarios?* – addresses the instantiation of the RMM. The company gave us access to their PAS simulation environment, which is used to test the automation software before and during commissioning. It allows experimenting with parts of the PAS in a virtual environment, i.e., machinery and production planning components are simulated. In this way we could execute our model in an SoS environment without interfering with real production systems in a plant. We describe the results of simulations to address RQ2. More specifically, we present data on the numbers and types of probes, events, constraints, and requirements managed in two different scenarios: one shorter simulation run of six hours and a long-time simulation

run of 168 hours (i.e., one week of continuous monitoring). For the six-hour run we measure the monitoring load in terms of the number of events, constraint checks performed, and violations that occurred. For the one-week run we assess the applicability of our approach for long-term continuous and uninterrupted operation.

B. RQ1 – Defining the Requirements Monitoring Model

We describe how we defined a RMM together with Primetals Technologies engineers and architects (cf. Fig. 5).

(1) Monitoring Scopes. Together with lead engineers of the different systems we defined the scope hierarchy necessary to monitor PAS requirements. We derived 24 different scopes on three levels – one SoS scope, four system scopes, and 19 component scopes.

(2) Requirements. We performed five two-hour workshops and several follow-up meetings with seven system experts to capture requirements and constraints from different parts of the PAS. Our goal for this evaluation was to cover all different types of scopes. 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. Other requirements cover the interaction of different parts of the system (e.g., user interface and database) or the checking of pre-conditions and post-conditions for certain events. As part of the evaluation, for example, we chose a requirement on the maximum duration of a metallurgical calculation. We also monitored 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 selected and modeled 40 requirements from several different systems of the PAS and assigned them to appropriate monitoring scopes.

(3) Event model. Checking requirements at runtime requires defining event types in the event model. For example, the non-functional requirement addressing hard-disk capacity for the log files of the systems requires a *Disk-Event* as a sub-element of a default *System-Event*. Similarly, the requirements covering the optimization component led to seven new event types in the Caster event model. In total we defined 109 different event types for the various systems.

(4a) Probes. Each event type has to be provided by a specific probe instrumenting the system. We instrumented Java classes using AspectJ leading to several method pointcuts (i.e., one for each method that needs to be instrumented). Each Aspect is represented by one probe. We directly instrumented C++ systems to forward the required events and data to the monitoring infrastructure. In this evaluation, we implemented 22 different probes for collecting information from C++ and Java systems as well as from the operating system.

(4b) Constraints. Based on the requirements we defined constraints supported by Primetals Technologies engineers who are familiar with the actual system and its expected behavior. For example, a requirement led to a constraint checking the start event of the optimization run and the event produced when returning the actual calculated optimization result. As soon as the the necessary data is provided (indicated by the tracked event *feedCyclicData_START*), the optimization result has to be calculated within five seconds and fed to the subsequent component (captured by the event *resultReceived*). We defined one constraint for each requirement during the evaluation.

TABLE II: The RMM for Primetals' PAS and its instantiation in two evaluation runs.

PAS RMM		Evaluation Runs		
Element	#	Element	6h Run	168h Run
Scopes:	24	Active Scopes:	5	5
Req./Constraints:	40	Checks Performed:	10,572	262,979
Event-Types:	109	Events Captured:	12,484	363,491
Probes:	22	Active Probes:	14	14

C. RQ2 – Instantiating the Requirements Monitoring Model

We used our monitoring infrastructure [15], the PAS simulation environment, and the RMM described above to monitor requirements from Primetals Technologies' PAS. Specifically, we selected eleven requirements which could be monitored in the simulation environment.

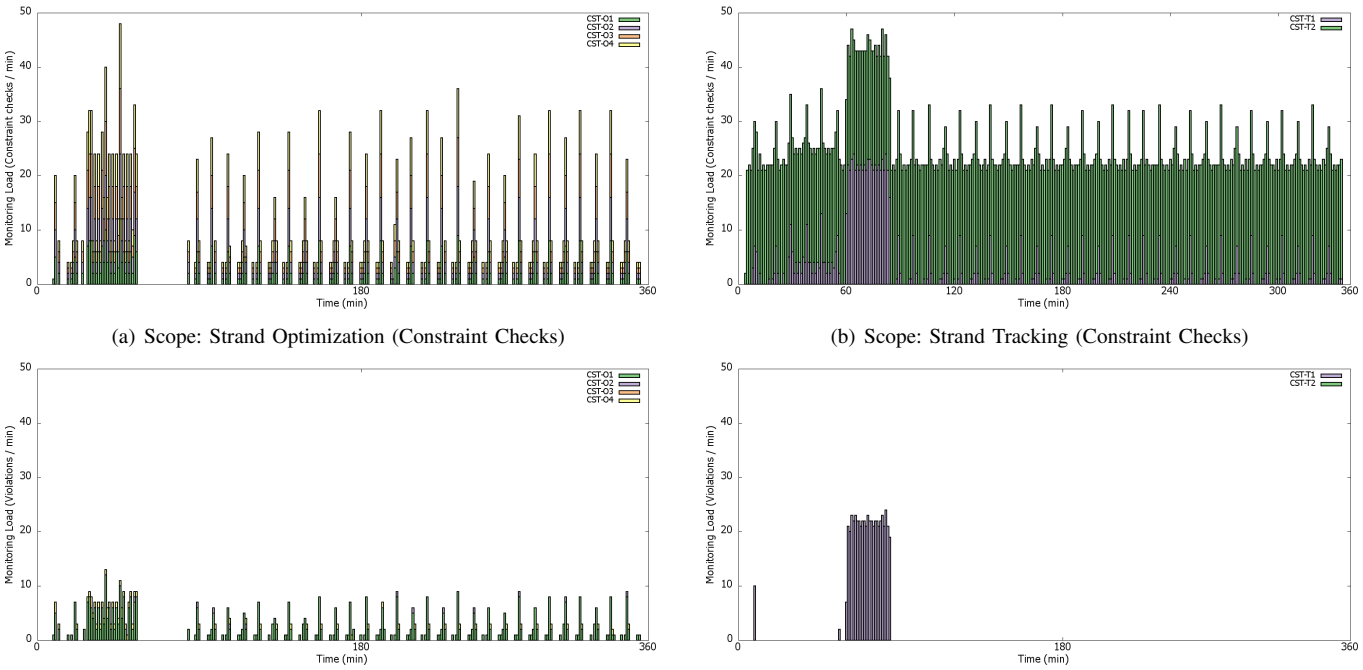
Evaluation setup.

After setting up the infrastructure and the simulation environment, we performed two separate evaluation runs: one shorter run of six hours and a long run of 168 hours. The aim of the six-hour run was to investigate the monitoring load in terms of the average number of constraints checked and violations detected. We deactivated one of the PAS systems after one hour to simulate a system failure and deviation from the normal course of operation. We then assumed that the system has been restarted by an operator, and re-activated the system after 30 minutes. The second evaluation run of 168 hours was used to assess whether the instantiated model together with the infrastructure is capable of long-term and uninterrupted operation, i.e., capable of processing the number of events, the amount of data, and the necessary constraint checks.

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.

Evaluation results. The results in Table II show that the monitored requirements led to a high number of events, constraints, and checks. For all monitored scopes, within the six hours of the first simulation run, 12,484 events and their related data were captured, resulting in a monitoring load of 34.6 events per minute. 10,572 constraints were checked (29 per minute) and 1,298 violations were detected (3.6 per minute). The high number of violations can be explained with the deactivation of the system and the use of the simulation environment.

For a closer look into the SoS we investigated the behavior of two monitoring scopes. The first scope represents a component providing calculations for arranging slabs on the strand



(a) Scope: Strand Optimization (Constraint Checks)

(b) Scope: Strand Tracking (Constraint Checks)

(c) Scope: Strand Optimization (Violations)

(d) Scope: Strand Tracking (Violations)

Fig. 7: Monitoring load of the two scopes *Strand Optimization* and *Strand Tracking* in the 6h evaluation run.

such that scrap is minimized (*Strand Optimization*). The second scope represents a component of the PAS responsible for tracking strand data in the casting system (*Strand Tracking*). Fig. 7 provides an overview of the Monitoring Load of the two different scopes regarding constraint checks performed per minute (upper half) and violations per minute resulting from those constraint checks (lower half). As casting is a cyclic process, we observed a recurring pattern of constraint checks and violations, besides minor deviations during the initialization phase of the simulation. We observed between 20 to 40 constraint checks per minute for both scopes, less than ten violations per minute for the Strand Optimization scope, and no violations for the Strand Tracking scope during normal operation.

After one hour we shut down the system responsible for material optimization, thereby simulating a failure of the system and its communication link to other systems. As can be seen in Fig. 7(a) the constraint checks dropped to zero as no more events were received. On the other hand, the load of the constraint checking the communication between the two scopes in Fig. 7(b) rose significantly, as requests to optimize were issued but not responded to and multiple retries were performed. Also, the number of violations in Fig. 7(d) increased accordingly. After reactivating the optimization system the situation returned to the normal recurring pattern.

For the one-week evaluation run 363,491 events were captured in total. The eleven requirements led to 262,979 constraint checks (roughly 26 checks per minute) varying from a rather low number of 32 checks performed for a requirement monitoring the cast sequences occurring up to 177,000 checks performed for a requirement monitoring watch-dog states within various parts of the continuous casting system of the PAS.

D. Discussion and Threats to Validity

Our evaluation confirms that diverse systems and components need to be considered when monitoring an SoS. Requirements exist for different systems, are maintained by different teams, and result in many different types of properties to be monitored (e.g., timing constraints, data checks, or invariants). We were able to represent the selected SoS requirements and their relations to the SoS architecture and events intercepted from the system in the RMM. Also, the underlying framework was able to handle the high number of incoming events and constraint checks. This confirmed our earlier test of the low-level infrastructure performance [15].

The results and findings are based on a single directed SoS in the domain of industrial automation. We are confident that our approach would also be helpful for other types of SoS [4]. Specifically, monitoring virtual SoS with little or no central management authority and highly emergent behavior would likely require other types of constraints than for centrally managed directed SoS due to additional coordination issues. Still, we think our approach could be applicable, however, an additional evaluation is needed. In collaborative SoS the systems voluntarily collaborate to fulfill the agreed purposes. As an agreement exists about the requirements defining a RMM will be similar to a directed SoS. Acknowledged SoS, which share a common management and resources but remain in independent ownership, do not differ much from directed

SoS regarding the definition of an RMM. The common management and resources of this type of SoS will help to define and manage the models.

Together with engineers of Primetals Technologies we selected requirements with different scopes, which are representative for the PAS. The requirements selected for the evaluation might not cover the full range of requirements existing in the SoS. However, given the scale and complexity of the PAS we consider our evaluation a good starting point representing a realistic case. In future work we will augment our model with more requirements to cover multiple levels of granularity.

Concerning the evaluation runs, a period of one week may not seem sufficient for a continuously operating system. However, this period covers multiple runs of common scenarios in the metallurgical process and no additional information could be gained from longer runs for the purpose of this study.

Our evaluation did not measure the overhead caused by the probes instrumenting the system. However, the probes we used are small, atomic code fragments only collecting data in the SoS. All processing and analyses tasks are performed independently on a separate machine without immediate impact on the monitored systems.

We also did not measure the overhead for defining the RMM. However, as discussed above the RMM will hardly be created in a single step but analysts will incrementally evolve the model. Our tools allow users to add new elements to the RMM at any time.

VIII. Related Work

We discuss requirements monitoring in general, techniques for processing and verifying events and data at runtime, as well as approaches for relating requirements and architecture.

Requirements monitoring approaches aim at determining the compliance of a system with its requirements during runtime [17], [5]. Monitors are used to detect requirements violations and serve as a starting point for revealing the root cause of problems. Robinson [6] 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. While this approach provides a solid basis for monitoring specific aspects of single systems – e.g., performance, data-flow, or system communication – it is limited regarding the diversity of monitors and support for monitoring SoS architectures, i.e., it does not consider monitoring scopes. Furthermore, relating monitoring scopes and requirements and/or events is out of scope of this and other existing requirements monitoring approaches.

Event processing and verification is the purpose of many existing approaches from diverse research areas. For instance, complex event processing (CEP) [27] 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. Also, CEP aims to combine multiple event streams gathered from multiple sources. Managing multiple probes instrumenting different systems in an SoS is also the aim of our work.

In the domain of service-oriented systems, Baresi et al. [20] propose the Service-Centric Monitoring Language (SECMOL), which allows to specify monitors for service-oriented systems including monitoring rules and operation protocols. They propose a high-level specification language for monitors to distinguish between application and monitoring, thus facilitating the evolution of the monitors in accordance with changing system requirements. This aim for flexibility is also our main goal, however, we do not focus on monitoring service-oriented systems but heterogeneous systems in SoS.

In the software architecture domain, Muccini et al. [7] present a framework for runtime monitoring of architectural properties, which uses aspect-oriented technologies for instrumenting the system according to selected architectural properties. The framework can monitor events occurring at runtime at defined extension points of an architecture to support evolution in component-based architectures. While we also use aspect-oriented techniques, we are not limited to this particular instrumentation approach and do not focus only on architectural properties when monitoring systems.

Furthermore, in the domain of runtime verification various approaches have been proposed to support monitoring and verifying system properties. Ghezzi et al. [9] 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. Runtime verification approaches can be used also in our context, i.e., to verify constraints based on event models at runtime.

Our SoS requirements monitoring model aims at integrating different techniques and technologies at the level of events without being restricted to a particular domain or technology of the system that needs to be monitored. Furthermore, we aim to model the dependencies among the different dimensions relevant for runtime monitoring, particularly among requirements and architecture. Relating *Requirements and Architecture* is challenging as requirements and architectures use different terms and concepts to capture elements relevant to each [28]. The CBSP taxonomy [29] covers a set of general architectural concerns derived from existing software architecture research. These dimensions are applied to systematically classify and refine requirements and to capture architectural trade-off issues and options. Each requirement is assessed for its relevance to the system architecture's components, connectors (buses), topology of the system or a particular system, and their properties. This idea is also applied in our RMM when defining the scope of a requirement, which also refers to

an architectural element in the SoS. Interaction scopes, for instance, are similar to the concept of connectors in various ADLs [11]. Our monitoring scopes differ from ADL-based approaches as we also consider the runtime perspective, i.e., we also consider the relation of the architecture to events occurring at runtime.

IX. Conclusions and Future Work

Organizations are not willing to use software for business-critical operations if the relationship to its requirements is vague [6]. In this paper we described the challenges of requirements monitoring in SoS motivated by an industrial system. We presented a requirements monitoring model for SoS combining three different dimensions: (1) monitoring scopes representing the SoS architecture from a monitoring point of view; (2) requirements describing the desired behavior of the SoS; as well as (3) event models providing an abstraction layer from different heterogeneous systems and technologies. We demonstrated that our model is sufficiently expressive to be tailored to a real-world SoS and that our approach can be used in a realistic requirements monitoring scenario. Our model allows to capture and manage the relations of requirements and their scopes, constraints, events and data, as well as probes.

We will be working on facilitating the traceability information the model provides to support error visualization and diagnosis of SoS. We also plan to extract information from system specifications, e.g., using NLP techniques, to ease populating our model. As in an SoS multiple requirements monitoring models may need to be defined for different systems maintained by different teams we are also investigating how to support model composition and decomposition and how to define the interfaces and relations of such models.

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.

References

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

[2] C. B. Keating, J. J. Padilla, and K. Adams, "System of systems engineering requirements: challenges and guidelines," *Engineering Management Journal*, vol. 20, no. 4, pp. 24–31, 2008.

[3] B. Boehm and J. A. Lane, "21st century processes for acquiring 21st century software-intensive systems of systems," *Crosstalk: Journal of Defence Software Engineering*, vol. 19, no. 5, pp. 4–9, 2006.

[4] J. S. Dahmann and K. J. Baldwin, "Understanding the current state of US defense systems of systems and the implications for systems engineering," in *2nd Annual IEEE Systems Conf.* IEEE, 2008, pp. 1–7.

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

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

[7] H. Muccini, A. Polini, F. Ricci, and A. Bertolino, "Monitoring architectural properties in dynamic component-based systems," in *10th Int'l Symposium on Component-Based Software Engineering*. Springer, 2007, pp. 124–139.

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

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

[10] N. Bencomo, J. Whittle, P. Sawyer, A. Finkelstein, and E. Letier, "Requirements reflection: requirements as runtime entities," in *Proc. 32nd Int'l Conf. on Software Engineering (vol. 2)*. ACM, 2010, pp. 199–202.

[11] N. Medvidovic and R. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, 2000.

[12] J. Cleland-Huang, C. K. Chang, and M. Christensen, "Event-based traceability for managing evolutionary change," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 796–810, 2003.

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

[14] M. Vierhauser, R. Rabiser, and P. Grünbacher, "A requirements monitoring infrastructure for very-large-scale software systems," in *Proc. 20th Int'l Working Conf. on Requirements Engineering: Foundation for Software Quality*. Springer, 2014, pp. 88–94.

[15] M. Vierhauser, R. Rabiser, P. Grünbacher, C. Danner, S. Wallner, and H. Zeisel, "A flexible framework for runtime monitoring of system-of-systems architectures," in *11th Working IEEE/IFIP Conference on Software Architecture*. IEEE, 2014, pp. 57–66.

[16] R. Rabiser, M. Vierhauser, and P. Grünbacher, "Variability management for a runtime monitoring infrastructure," in *9th Int'l WS on Variability Modelling of Software-intensive Systems*. ACM, 2015, pp. 35–42.

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

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

[19] N. R. Mehta, N. Medvidovic, and S. Phadke, "Towards a taxonomy of software connectors," in *Proc. 22nd Int'l Conf. on on Software Engineering*. ACM, 2000, pp. 178–187.

[20] 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.

[21] R. J. Hall, "Open modeling in multi-stakeholder distributed systems: Research and tool challenges," in *Proc. of the 9th Int'l Symposium on Static Analysis*. Springer, 2002.

[22] A. Klinger, T. Kronberger, M. Schaler, B. Schürz, and K. Stohl, *Expert Systems Controlling the Iron Making Process in Closed Loop Operation*. InTech, 2010.

[23] C. Federspiel, J. Bogner, N. Hübner, R. Leitner, W. Oberaigner, K. König, and L. Lindenberger, "Next generation level2 systems for continuous casting," in *5th European Continuous Casting Conf.* IOM Communications Ltd, 2005.

[24] 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 Conf. on Software Engineering (vol. 2)*. Hyderabad, India: ACM, 2014, pp. 125–134.

[25] 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 *15th Int'l Conf. on Model Driven Engineering Languages and Systems*. Springer, 2012, pp. 531–545.

[26] A. Egyed, "Instant consistency checking for the UML," in *Proc. 28th Int'l Conf. on Software engineering*. ACM, 2006, pp. 381–390.

[27] D. C. Luckham, *Event processing for business: organizing the real-time enterprise*. Wiley, 2011.

[28] B. Nuseibeh, "Weaving together requirements and architectures," *IEEE Computer*, vol. 34, no. 3, pp. 115–117, 2001.

[29] P. Grünbacher, A. Egyed, and N. Medvidovic, "Reconciling software requirements and architectures with intermediate models," *Software and System Modeling*, vol. 3, no. 3, pp. 235–253, 2004.