University of Pennsylvania
ScholarlyCommons

Departmental Papers (CIS) | Department of Computer & Information Science

9-2019

Runtime verification of parametric properties using SMEDL

Teng Zhang
University of Pennsylvania, tengz@cis.upenn.edu

Ramneet Kaur
University of Pennsylvania, ramneetk@seas.upenn.edu

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

Follow this and additional works at: https://repository.upenn.edu/cis_papers

 Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

Teng Zhang, Ramneet Kaur, Insup Lee, and Oleg Sokolsky, "Runtime verification of parametric properties using SMEDL", *From Reactive Systems to Cyber-Physical Systems* Lecture Notes in Computer Science, vol 11500, 276-293. September 2019. http://dx.doi.org/10.1007/978-3-030-31514-6_16

From Reactive Systems to Cyber-Physical Systems. Lecture Notes in Computer Science, vol 11500. Springer, Cham

This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_papers/855
For more information, please contact repository@pobox.upenn.edu.

Runtime verification of parametric properties using SMEDL

Abstract

Parametric properties are typical properties to be checked in runtime verification (RV). As a common technique for parametric monitoring, trace slicing divides an execution trace into a set of sub traces which are checked against non-parametric base properties. An efficient trace slicing algorithm is implemented in *MOP*. Another RV technique, *QEA* further allows for nested use of universal and existential quantification over parameters. In this paper, we present a methodology for parametric monitoring using the RV framework SMEDL. Trace slicing algorithm in MOP can be expressed by execution of a set of SMEDL monitors. Moreover, the semantics of nested quantifiers is encoded by a hierarchy of monitors for aggregating verdicts of sub traces. Through case studies, we demonstrate that SMEDL provides a natural way to monitor parametric properties with more potentials for flexible deployment and optimizations.

Keywords

runtime verification, parametric property, trace slicing, SMEDL

Disciplines

Computer Engineering | Computer Sciences

Comments

From Reactive Systems to Cyber-Physical Systems. Lecture Notes in Computer Science, vol 11500. Springer, Cham

This book chapter is available at ScholarlyCommons: https://repository.upenn.edu/cis_papers/855

Runtime verification of parametric properties using SMEDL*

Teng Zhang, Ramneet Kaur, Insup Lee, and Oleg Sokolsky

University of Pennsylvania, Philadelphia PA 19104, USA,
{tengz,ramneetk,lee,sokolsky}@cis.upenn.edu

Abstract. Parametric properties are typical properties to be checked in runtime verification (RV). As a common technique for parametric monitoring, trace slicing divides an execution trace into a set of sub traces which are checked against non-parametric base properties. An efficient trace slicing algorithm is implemented in *MOP*. Another RV technique, *QEA* further allows for nested use of universal and existential quantification over parameters. In this paper, we present a methodology for parametric monitoring using the RV framework SMEDL. Trace slicing algorithm in MOP can be expressed by execution of a set of SMEDL monitors. Moreover, the semantics of nested quantifiers is encoded by a hierarchy of monitors for aggregating verdicts of sub traces. Through case studies, we demonstrate that SMEDL provides a natural way to monitor parametric properties with more potentials for flexible deployment and optimizations.

Keywords: Runtime verification · Parametric property · Trace slicing · SMEDL.

1 Introduction

Runtime verification (RV) is a technique for monitoring correctness of systems. The objective of RV is to use runtime monitors to check properties against a run of a system (referred as a target system) which can be abstracted as an event trace from the execution or the logging information. Usually, the event stream delivered to a monitor carries data bound to event parameters. The property may depend not only on event order in the trace but also on parameter values of events.

Example 1: unsafeMapIter [26]. An iterator of a collection created from a map is not allowed to be used after the map has been updated. The property that points out the violation of it can be described as a parametric regular expression : $createC(m, c)updateM(m)^* createI(c, i)useI(i)^* updateM(m)^+ useI(i)$ where $createC(m, c)$ denotes creation of a collection c, the key set of a map m;

* This work is supported in part by the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under contract FA8750-16-C-0007 and by ONR SBIR contract N00014-15-C-0126.

createI(c, i) is creation of iterator i from c; *updateM*(m) is update of m; and *useI*(i) is use of i.

To monitor parametric properties, an efficient trace slicing algorithm is implemented in the MOP framework [26]. A parametric event trace is sliced into sub traces according to event parameters. Each sub trace is then checked against a non-parametric property. The property of the whole trace is obtained by aggregation of verdicts from all sub traces. QEA [4] further supports nested use of universal or existential quantifiers over parameters.

In [33], we presented a general RV framework SMEDL. A monitoring system in SMEDL is composed of a set of monitor instances communicating with each other using events, forming a *monitor network*. Instances can be created dynamically by binding monitor parameters with values. A scalable monitor network can not only describe multiple types of properties such temporal properties and numeric properties but also provides a flexible and intuitive way for monitor deployment [32], which is vital for balancing between the overhead of monitoring and timeliness of getting verdicts.

In this paper, we will further use SMEDL to describe and check parametric properties. We will present a transformation from MOP to SMEDL through an example of a MOP specification. The trace slicing algorithm can be represented by execution and evolution of a monitor network. We then will present that the semantics of nested quantifiers in QEA can be described by a hierarchy of SMEDL monitors aggregating verdicts from sub traces. Due to its flexibility in specifying monitors and communications, SMEDL may check parametric properties with more potentials for flexible deployment and optimizations.

The paper is organized as follows. Section 2 gives definitions of SMEDL and introduces MOP and QEA. Section 3 presents a transformation algorithm from MOP to SMEDL and illustrates how to check parametric properties using SMEDL monitors. Section 4 presents how to construct SMEDL monitors to express nested quantifiers in QEA. Section 5 presents the related work and Section 6 concludes the paper and presents the future work.

2 Preliminaries

2.1 Overview of SMEDL

A SMEDL specification contains a set of monitor specifications and an architecture description that captures patterns of communication between them. The relation between a SMEDL specification and a monitor network is illustrated in Fig 1. During execution, each monitor can be instantiated as monitor instances multiple times with different parameters, either statically during startup of the target system or dynamically at runtime, in response to receiving *creation* events. The event communication and creation of instances within the monitor network is controlled by a *global wrapper* according to the architecture description.

Single monitor. A SMEDL monitor is a collection of *scenarios*. Each scenario is an EFSM (Extended Finite State Machine) [33] in which the transitions

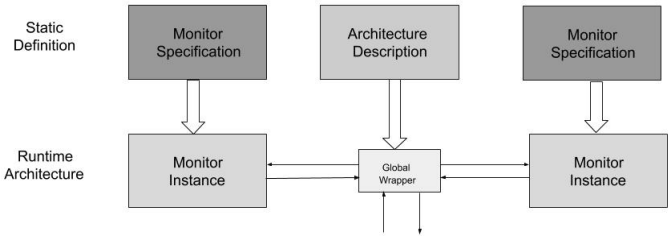

Fig. 1. SMEDL overview

are performed by reacting to events. Scenarios interact with each other using shared state variables or by triggering execution of other scenarios through raised events. There are three types of events: *imported*, *exported* and *internal*. Imported events, which are responsible for triggering the execution of a monitor, are raised from the target system or by other monitors; exported events are raised within the monitor and sent to other monitors; internal events are used to trigger transitions, but are only seen and processed within the monitor. Each transition is labeled with a triggering event and attached to a guard condition and a list of actions to be executed after the transition. Actions on transitions can raise events and update state variables. A monitor may have a set of typed parameters for identification. Multiple instances are created by binding parameters with actual values. The detailed syntax and semantics of a monitor was presented in [34].

Architecture description and monitor network. The architecture description defines the event communication pattern among monitors, which consists of a set of *monitor interfaces* and *event connection specifications*. The interface of an monitor contains the name, parameter list, imported events and exported events of that monitor. If an imported event is labeled as a *creation* event, it can be used to create a instance of that monitor. When multiple instances exist, a finer control on delivery of events is desirable. For instance, we could specify that an event raised by an instance of monitor A is sent to instances of monitor B having the same value on the first parameter. This is achieved by *event connection specifications*.

An event connection specification is a tuple (*SrcMon*, *SrcEv*, *TarMon*, *TarEv*, *PatternExprs*), which specifies how a source event *SrcEv* exported from a source monitor *SrcMon* is delivered to a target monitor *TarMon* as its imported event *TarEv*. Note that *SrcMon* is empty if *SrcEv* is sent from the target system. Each parameter of a monitor or an event corresponds to an index according to its position in the parameter list, starting from 0. Each element of *PatternExpr* is a tuple (*targetIdx*, *source*, *sourceIdx*), meaning that the parameter value of *TarMon* with index *targetIdx* must be matched to the parameter value of *source* with index *sourceIdx*. *source* can be either *SrcMon* or *SrcEv*.

For example, an event connection specification that describes event delivery from $e1(x, y)$ of $mon1\langle a\rangle$ to $e2(x', y')$ of $mon2\langle b, c\rangle$ is defined as ($mon1$, $e1$, $mon2$,

e2, *ps*) where *ps* is $\{(0, mon1, 0), (1, e1, 0)\}$. Note that (*x*,*y*) is the formal parameter list of *e1*; *a* is the formal parameter of *mon1* and so on. When an event instance $e1(x1, y1)$ is sent from an monitor instance $mon1(a1)$, only one monitor instance of *mon2*, $mon2(a1, x1)$ receives it as $e2(x1, y1)$ as *ps* specifies that the first and second parameter of *mon2* are respectively matched to the first parameter of *mon1* and *e1*. *e2* is instantiated by parameters of *e1*.

If there is no instance of *mon2* parameterized with $(a1, x1)$ and *e2* is a creation event for *mon2*, $mon2(a1, x1)$ will be created. Note that if *TarEv* is a creation event, corresponding *PatternExprs* must specify mapping relations to all parameters of *TarMon*. If *PatternExprs* is empty, each raised *SrcEv* is sent to all existing instances of *TarMon*.

The overall flow of event processing conducted by a monitor network is illustrated in Fig. 2. The global wrapper receives/outputs events from/to the environment and controls event dispatch to monitor instances and creation of instances using the architecture description. Two shared data structures, *InnerQueue* and *OutputQueue* are used to store events that are to be consumed within the monitor network and sent to the environment. The execution of the global wrapper begins with an event from the environment put into the *InnerQueue*. The global wrapper waits for the next incoming event from the environment after all events in the *InnerQueue* have been consumed and events in the *OutputQueue* have been sent to the environment.

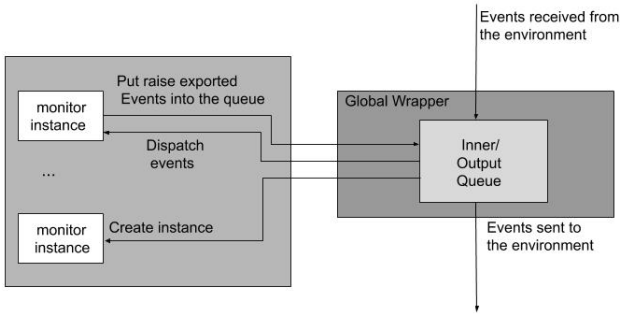

Fig. 2. Architecture of a monitor network

The pseudocode for the global wrapper is shown in Algorithm 1, parameterized by the architecture description. *monTypeList* is the list of all monitors used for checking the property. Initially, an imported event *e* is sent from the environment into *InnerQueue* to trigger the execution of the global wrapper. The global wrapper pulls out the event (denoted as *curE*) at the frontend of *InnerQueue*. *curE* is mapped to the event in *m* (denoted as *ev*) in *matchInComingEvent* by looking up the architecture description. Monitors that cannot handle *curE* are filtered out before traversing *monTypeList*. Process *consume* dispatches *ev* to all compatible instances of *m*. The set of raised events *ies* and

oes are then put into the *InnerQueue* and *OutputQueue* based on whether they are to be consumed within the monitor network. Note that all raised events carry the parameter information of corresponding instances which have raised them. If there is no compatible instance and *ev* is a creation event, an instance of m is created from *ev*. After all events in the *InnerQueue* have been handled, events in the *OutputQueue* will be sent to the environment or raised as alarms. It is worth noting that *consume* is an abstract representation of monitor execution. Moreover, we leave implementation flexibility in the algorithm. For instance, no order is defined in *monTypeList*. In Section 3, we impose a specific order among monitors in *monTypeList* to implement the trace slicing algorithm in MOP.

Algorithm 1 Global wrapper for parametric monitoring

```
1: InnerQueue ← {e}, OutputQueue ← {}
2: procedure GLOBALSTEP(archDescription)
3:     monTypeList ← monitors declared in archDescription
4:     while InnerQueue ≠ ∅ do
5:         curE ← retrieveFromQueue(InnerQueue)
6:         for m ∈ filter(monTypeList, curE, archDescription) do
7:             ev ← matchInComingEvent(curE, m, archDescription)
8:             (ies, oes) ← consume(m, archDescription, ev)
9:             enQueue(InnerQueue, ies)
10:            enQueue(OutputQueue, oes)
11:    sendEvents(OutputQueue)
```

2.2 Overview of MOP

MOP is a monitoring framework supporting description of properties by multiple logical formalisms. In this paper, we only consider properties that are synthesized into FSMs (Finite State Machines). One can specify different ways of reporting verdicts and handling violations or validations of properties. A MOP monitor $M_{mop}\langle X_{mop}\rangle$ contains two parts. X_{mop} is the parameter set and M_{mop} is an finite state machine (FSM).

MOP implements an efficient trace slicing algorithm [11] for parametric monitoring, which is independent of the base monitor for checking the non-parametric property. The algorithm maintains a mapping Δ from bindings to current states in the base monitor. A *binding* is a partial function $X_{mop} \rightharpoonup Val$ from parameters to values. *Val* represents the set of all possible values for X_{mop}. Parameters of all parametric events are from X_{mop}. We denote $e\langle\theta\rangle$ as an event parameterized by the binding θ.

When an event $e\langle\theta\rangle$ arrives, the algorithm will update states of all existing bindings which has equal or more information than θ using e. If $dom(\theta_1)$ (the domain of θ_1) is the subset of $dom(\theta_2)$ and $\theta_1(x) = \theta_2(x)$ for all x $\in dom(\theta_1)$, we say θ_1 has equal or less information than θ_2, denoted as $\theta_1 \sqsubseteq \theta_2$. If $\Delta(\theta)$

is undefined (and e is defined as a creation event in MOP), the algorithm will define $\Delta(\theta)$ using the state updated from $\Delta(\theta')$ by e where θ' is the largest binding in $dom(\Delta)$ that has less information than $\Delta(\theta)$. New bindings can also be created from extending existing bindings in Δ that are *compatible* with θ. Two bindings θ_1 and θ_2 are *compatible* with each other when $\theta_1(x)$ is equal to $\theta_2(x)$ for all x $\in dom(\theta_1) \cap dom(\theta_2)$. The *combination* between two bindings θ_1 and θ_2 is defined as follows: if θ_1 and θ_2 are compatible, $\theta_1 \sqcup \theta_2(x) = \theta_1(x)$ if $x \in dom(\theta_1)$; $\theta_1 \sqcup \theta_2(x) = \theta_2(x)$ if $x \in dom(\theta_2)$; $\theta_1 \sqcup \theta_2(x)$ is undefined if x is undefined in θ_1 and θ_2. The algorithm will always extend the binding with more information in e to generate the new binding. The detailed description for the slicing algorithm is in [11]. The notations introduced here will be reused in the rest of the paper.

SMEDL vs. MOP. In MOP, the mechanism for creating and updating parameter instances is controlled by the slicing algorithm which is independent of the monitor specification. Partially instantiated monitor instances are maintained in the algorithm. By contrast, SMEDL realizes parametric monitoring at the level of the semantics of monitor network. All monitor instances are created with full instantiation. In section 3, we present a transformation from a MOP specification to a set of SMEDL monitors connecting through events. The idea is to analyze the structure of a parametric FSM in MOP and generate SMEDL monitors that are to be fully instantiated by creation events. The information of how to extend and update bindings is encoded in the single monitor specifications and the architecture description. The architecture description guarantees that events are only sent to compatible monitor instances.

2.3 Overview of QEA

QEA (Quantified Event Automata) is a formalism for parametric monitoring. A QEA is a pair $\langle \Lambda, E \rangle$ where E is an *Event Automaton* and $\Lambda \in (\{\forall, \exists\} \times vars(E) \times Guard)^*$ is a list of quantifiers with guards. An Event Automaton (EA) is an EFSM in which transitions are enriched with guard and assignments to variables; $vars(E)$ is the set of variable names appearing in E. In this paper, we focus on the semantics of nested quantifiers [4, 27]. QEA also uses trace slicing to accomplish parametric monitoring. The acceptance for a parametric property for QEA is defined in [4], as illustrated below. In the terminology of QEA, a *ground* trace contains events of which all parameters are bound to concrete values; $Dom(\tau)(x)$ returns the derived domain for the parameter x in the trace τ; $\theta_1 \dagger \theta_2$ overrides the value in θ_1 by θ_2; $g(\theta)$ is the guard condition over the quantified variable; $E(\theta)$ is an event automaton E with its variables instantiated by θ; $\tau \downarrow_{E(\theta)}$ is the projection of a trace τ over $E(\theta)$; $L(E(\theta))$ is the set of traces accepted by $E(\theta)$.

Definition 1 (Acceptance in QEA). *A QEA accepts a ground trace τ if $\tau \models_{\langle\rangle} \Lambda.E$ where $\models_\theta$ is defined as*

$\tau \models_\theta (\forall x : g)\Lambda'.E$ *iff* $\forall\ d \in Dom(\tau)(x)$, *if* $g(\theta \dagger \langle x \to d \rangle)$ *then* $\tau \models_{\theta \dagger \langle x \to d \rangle} \Lambda'.E$.

$\tau \models_\theta (\exists x : g)\Lambda'.E$ *iff there exists* $d \in Dom(\tau)(x)$, *if* $g(\theta \dagger \langle x \to d \rangle)$ *then* $\tau \models_{\theta \dagger \langle x \to d \rangle} \Lambda'.E$.

$\tau \models_\theta \epsilon.E$ *iff* $\tau \downarrow_{E(\theta)} \in L(E(\theta))$.

Bindings are generated by inductively traversing the derived domain of each variable in the nested quantifiers. When a full binding is created, the verdict is retrieved from the corresponding event automaton. The aggregation of the result is decided by which quantifier is used for a parameter variable. The interpretation of nested quantifiers in QEA leads to one significance difference between QEA and MOP in generating bindings: QEA records any binding that can be built from the derived domain that has a non-empty projection. For example, if $e(\theta)$ arrives where $\theta = \langle x \to x1, y \to y1 \rangle$ and there is a binding $\theta_1 = \langle y \to y2, z \to z1 \rangle$ ($y1 \neq y2$), a new binding $\theta' = \langle x \to x1, y \to y2, z \to z1 \rangle$ would be created with e adding to its trace projection.

SMEDL vs. QEA. QEA has a uniform algorithm to handle the semantics of nested quantifiers. However, if the property indicates relations between quantified variables by events which cannot be described by the guard condition, bindings that do not comply with the relations may be generated. In Section 4, we will show that the semantics of nested quantifiers can be encoded through hierarchical *aggregation* monitors in SMEDL. Moreover, we will demonstrate that SMEDL can check the property involving the relation between quantified variables by properly generating monitor instances.

3 Implementation of trace slicing in SMEDL

This section presents how to use SMEDL to implement the trace slicing algorithm in MOP. Through an example, we first present a transformation from an FSM-based MOP monitor into a set of SMEDL monitors. Then, we propose the detailed design of the global wrapper mentioned in Section 2.1 and demonstrate that a monitor network in SMEDL controlled by the global wrapper can correctly monitor parametric properties.

We present a transformation from an FSM-based MOP monitor to a SMEDL specification based on the *Example 1* in Section 1. Recall that *Example 1* states a property *UnsafeMapIter* that an iterator of a collection must not be used after the corresponding map of that collection is updated. *UnsafeMapIter* has a parameter set with three variables: map(m), collection(c) and iterator(i). The FSM definition is illustrated in Fig 3. The shaded states are accepting states, meaning there is no violation of the property. Note that the original FSM is complete (which means for each event in the alphabet of the FSM, there exists at least one transition triggered by the event from all states of that FSM) while self-looping transitions are omitted for clearer illustration. The process of constructing a set of SMEDL monitors corresponding to *UnsafeMapIter* are presented below.

Recall that when a SMEDL monitor instance is created, all its parameters should be bound to a value. As a result, multiple monitors with different parameters are necessary. In *UnsafeMapIter*, there are four events, *createC*(m, c),

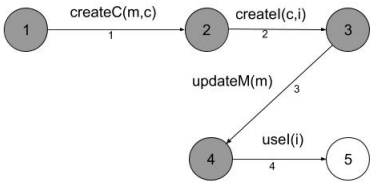

Fig. 3. FSM definition of *UnsafeMapIter*

updateM(m), *createI*(c, i) and *useI*(i). All possible combinations of parameter variables include $\langle m \rangle$, $\langle i \rangle$, $\langle m, c \rangle$, $\langle c, i \rangle$, $\langle m, i \rangle$ and $\langle m, c, i \rangle$. Generally, a SMEDL monitor should be created for each combination. However, since *createC*(m, c) is the only event that can start a trace [26], only two bindings $\langle m, c \rangle$ and $\langle m, c, i \rangle$ will be generated and maintained in MOP. Two SMEDL monitors, $mc\langle m, c \rangle$ and $mci\langle m, c, i \rangle$ are to be constructed.

The specifications of *mc* and *mci* are illustrated in Fig 4. *mc* is responsible for storing all seen value pairs of (m,c) carried by *createC*. When *mc* receives *createI*(c, i), *createM2*(i) is raised to trigger creation of a new instance of *mci* carrying the value of (m,c,i). Note that *createM2*(i) only carries i because *mci* knows which instance of *mc* has raised it. *mci* then checks whether *useI*(i) happens after *updateM*(m).

To construct SMEDL monitors from an FSM specification, the first step is to map states in the FSM into states in the SMEDL specifications. In *UnsafeMapIter*, m and c are bound in state 2 while i is further bound in state 3, 4 and 5. As a result, we map state 2 into *mc* and state 3, 4 and 5 into *mci*. In the rest of the paper, we assume that corresponding states between the FSM and the SMEDL specifications have the same name.

Then, the transitions in the FSM are mapped into SMEDL monitors. If the source and target state of a transition in the FSM carry the same parameter information, then it can be directly mapped to the corresponding SMEDL specification. For instance, transition 8 and 9 in *mci* are mapped from transition 3 and 4 in the FSM definition. If a transition $tr : s1 \rightarrow s2$ by an event e has the source and target state with different parameter information θ_1 and θ_2, there are two cases. If θ_1 is empty, a transition from the initial state s to $s2$ is generated in the SMEDL monitor $m\langle\theta_2\rangle$. For instance, transition 5 in *mc* is mapped from transition 1 in the FSM. If θ_1 is not empty, two transitions are generated. One is in $m\langle\theta_1\rangle$ from $s1$ to $s1$ triggered by e with raising an event re. Another one is in $m'\langle\theta_2\rangle$ from the initial state s to $s2$, triggered by re. For instance, transition 6 in *mc* and transition 7 *mci* are generated from transition 2 in the FSM. Omitted transitions in the FSM are also mapped to *mc* and *mci* in the same way. We could also further optimize *mc* and *mci* by removing unnecessary transitions. For instance, *mc* does not need to receive *useI* or *updateM* while *mci* does not need to receive *createI* and *createC*.

Finally, the communication is specified in the architecture description. The communication between *mc* and *mci* is specified as: $\langle$*mc*, *createM2*, *mci*, *createM2*

$, ps\rangle$ where ps is $\{\langle 0, mc, 0\rangle, \langle 1, mc, 1\rangle, \langle 2, createM2, 0\rangle\}$. Note that the two *createM2* in the architecture description represent the exported event of mc and the imported event of mci and ps specifies that m and c of mci are from the first and second parameter of mc while i is from the first parameter of *createM2*. The communication between mc and the environment is defined as: 1) $\langle null, createC, mc, createC, ps1\rangle$ where $ps1$ is $\{\langle 0, createC, 0\rangle, \langle 1, createC, 1\rangle\}$; 2) $\langle null, createI, mc, createI, \{\langle 1, createI, 0\rangle\}\rangle$, respectively specifying how $createC$ and $createI$ are sent to mc.

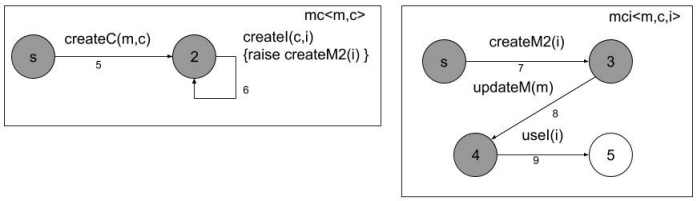

Fig. 4. SMEDL definition of *UnsafeMapIter*

The monitor design and connection specified in the architecture description statically describe how bindings are created or extended by other bindings. To fully implement the trace slicing algorithm, we need to impose an order to elements in *monTypeList* in Algorithm 1 according to the relation $\sqsubseteq$ over monitor parameters: if $\theta_2 \sqsubseteq \theta_1$, $m\langle\theta_1\rangle$ is placed before $m'\langle\theta_2\rangle$ in *monTypeList*. Note that no two monitors in *monTypeList* will have identical parameter list. A monitor with more parameter information (which means in the front of *monTypeList*) will be executed before the one with less parameter information. Corresponding raised events will also be placed in the *InnerQueue* following this order. This ensures that an instance will be created by the creation event carrying the most parameter information, complying with the slicing algorithm that always creates a new binding by extending the most informative binding if possible.

We use an event trace $\tau : updateM\langle m_1\rangle, createC\langle m_1, c_1\rangle, createC\langle m_2, c_2\rangle, createI\langle c_1, i_1\rangle, useI\langle i_1\rangle$ [26] to illustrate the execution of the global wrapper. The state evolution of mc and mci is given in Table 1. Since $updateM$ is not the creation event of mc or mci, no instance is created. When $createC\langle m_1, c_1\rangle$ and $createC\langle m_2, c_2\rangle$ arrive, two instances of mc are created and transitioned to state 2. $createI\langle c_1, i_1\rangle$ triggers the creation of $mci\langle m_1, c_1, i_1\rangle$ by sending *createM2*(i_1) to mci. $mci\langle m_1, c_1, i_1\rangle$ is in state 3 after creation. $useI\langle i_1\rangle$ is sent to $mci\langle m_1, c_1, i_1\rangle$ and a self-looping transition is executed. It is worth noting that no instance of $mci\langle m_2, c_2, i_1\rangle$ is created. This indicates that SMEDL can not only implement the trace slicing but provide a flexible way for optimization.

In the more general case, one monitor may have more than one creation event and a subset of them may be raised reacting to an incoming event. We modify the property *UnsafeMapIter*, changing the parameters of *createI* to $\langle m, c, i\rangle$ and trying to catching the illegal behavior that *createI* arrives before

Table 1. State update of SMEDL monitors given τ

updateM(m_1)	createC(m_1,c_1)	createC(m_2,c_2)	createI(c_1,i_1)	useI(i_1)
$\emptyset$	$mc\langle m_1, c_1\rangle$:2	$mc\langle m_1, c_1\rangle$:2 $mc\langle m_2, c_2\rangle$:2	$mc\langle m_1, c_1\rangle$:2 $mc\langle m_2, c_2\rangle$:2 $mci\langle m_1, c_1, i_1\rangle$:3	$mc\langle m_1, c_1\rangle$:2 $mc\langle m_2, c_2\rangle$:2 $mci\langle m_1, c_1, i_1\rangle$:3

createC. The SMEDL specification is illustrated in Fig. 5(a). Suppose a setting in which there is an instance $mc\langle m_1, c_1\rangle$ and no instance $mci\langle m_1, c_1, i_1\rangle$. When $createI\langle m_1, c_1, i_1\rangle$ is sent to *mc* and *mci*, it will first trigger the execution of *mci* before *mc* because it has more parameter information than *mc*, as presented below. As a result, a new instance $mci\langle m_1, c_1, i_1\rangle$ is created and transitioned to state 5. However, it is not consistent with the semantics of the slicing algorithm, which would create $mci\langle m_1, c_1, i_1\rangle$ by *createM2* raised from $mc\langle m_1, c_1\rangle$ by $createI\langle m_1, c_1, i_1\rangle$. To achieve the desired result, the SMEDL specification is modified as shown in Fig. 5(b), which removes *createI* as a creation event of *mci*. Instead, *createI* is a creation event of *mc*, corresponding to transition 11. This modification guarantees that an instance of *mci* can always be created by the correct event.

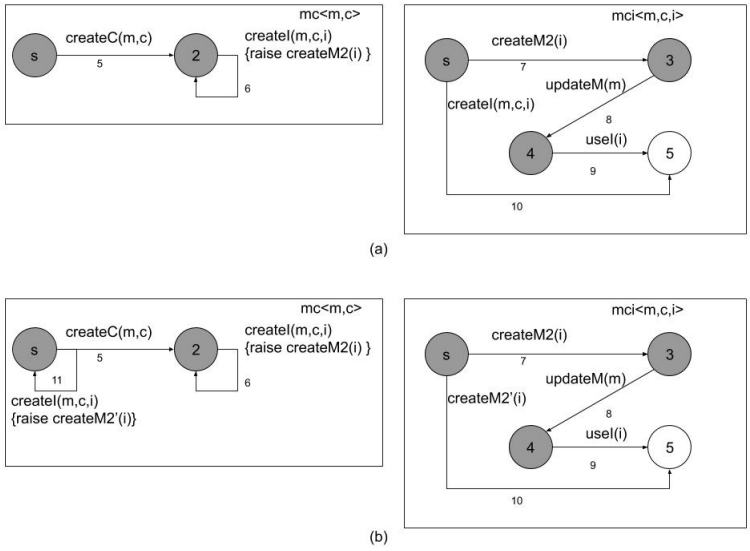

Fig. 5. Modification of SMEDL definition of *UnsafeMapIter*

We test the SMEDL specification in Fig. 5(b) using two traces $\tau_1 : createI\langle m_1, c_1, i_1\rangle$ and $\tau_2 : createC\langle m_1, c_1\rangle, createI\langle m_1, c_1, i_1\rangle$. For τ_1, $mc\langle m_1, c_1\rangle$ and $mci\langle m_1, c_1, i_1\rangle$ are created in state s and state 5. For τ_2, $mc\langle m_1, c_1\rangle$ and $mci\langle m_1, c_1, i_1\rangle$ are in state 2 and state 3.

4 Expressing quantifiers in SMEDL

This section further explores expressing parametric properties with nested quantifiers introduced in QEA. We first propose a methodology to implement aggregation using a SMEDL monitor network through *Example 2* below. Then we use a modified version of *Example 2* to illustrate the flexibility of SMEDL to implement aggregation when the relation between parameters needs to be considered. The SMEDL specifications for *Example 2* (also *Example 4* below) are available online[1].

Example 2: candidateSelection [4]. For every voter there must exist a party that the voter is a member of, and the voter must rank all candidates for that party. The QEA specification is shown in Fig 6, which contains two parts, the declaration of nested quantifiers and an event automaton (EA). There are three quantified variables, v(voter), c(candidate) and p(party) and three parametric events *member*, *candidate* and *rank*. The third parameter r of *rank* is an unquantified variable. The shaded circles in the EA represent accepting states. Self-looping transitions are omitted. To simplify the presentation, we impose a restriction on event order of traces: all *candidate* events always happen after all *member* events and all *rank* events happen after all *candidate* events.

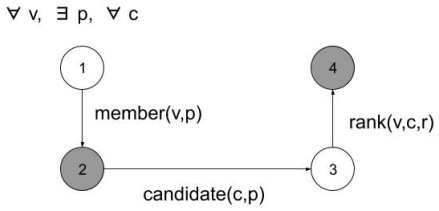

Fig. 6. QEA specification for candidate selection

The EA is transformed into a set of SMEDL monitors using the same process proposed in Section 3, as illustrated in Fig 7. When fed with event trace τ_3 : $member(tom, red)$,$member(ali, blue)$, $candidate(jim, red)$, $candidate(flo,red)$, $candidate(don, blue)$, $rank(tom, jim, 1)$, $rank(ali, don, 1)$, corresponding state evolution for the monitor network is shown in Table 2. Compared with bindings generated by execution of QEA in [4], fewer instances are generated. For example, there is a binding $\langle v : a, p : r, c : j\rangle : candidate(j, r)$ in QEA (all values are abbreviated to the initial alphabet) but not in SMEDL. This binding does not influence the verdict of the property for τ_3 because *ali* is not a member of *red* and the property only requires the existence of a party.

The architecture is illustrated in Fig 8. The high level idea is to use a hierarchy of *aggregation monitors* to implement the semantics of nested quantifiers.

[1] https://github.com/tengz2019/parametricSMEDL

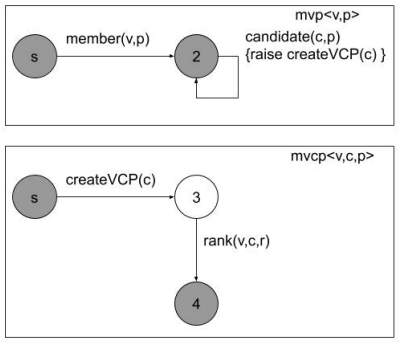

Fig. 7. SMEDL monitors for candidate selection

Table 2. State update of SMEDL monitors given τ_3

member(t,r)	member(a,b)	candidate(j,r)	candidate(f,r)	candidate(d,b)	rank(t,j,1)	rank(a,d,1)
$mvp\langle t,r\rangle$:2	$mvp\langle t,r\rangle$:2 $mvp\langle a,b\rangle$:2	$mvp\langle t,r\rangle$:2 $mvp\langle a,b\rangle$:2 $mvcp\langle t,j,r\rangle$:3	$mvp\langle t,r\rangle$:2 $mvp\langle a,b\rangle$:2 $mvcp\langle t,j,r\rangle$:3 $mvcp\langle t,f,r\rangle$:3	$mvp\langle t,r\rangle$:2 $mvp\langle a,b\rangle$:2 $mvcp\langle t,j,r\rangle$:3 $mvcp\langle t,f,r\rangle$:3 $mvcp\langle a,d,b\rangle$:3	$mvp\langle t,r\rangle$:2 $mvp\langle a,b\rangle$:2 $mvcp\langle t,j,r\rangle$:4 $mvcp\langle t,f,r\rangle$:3 $mvcp\langle a,d,b\rangle$:3	$mvp\langle t,r\rangle$:2 $mvp\langle a,b\rangle$:2 $mvcp\langle t,j,r\rangle$:4 $mvcp\langle t,f,r\rangle$:3 $mvcp\langle a,d,b\rangle$:4

For each quantified variable, an aggregation monitor is constructed which receives checking results from other monitor instances and aggregates them using logical operations such as conjunction or disjunction. Each $mvcp\langle v,c,p\rangle$ instance checks whether the voter v belonging to the party p has ranked the candidate c in the trace and sends the result in *resultVCP* to *collectC*. Moreover, when a new instance of *mvcp* is created, a *countC* event is raised and sent to *collectC*. $collectC\langle v,p\rangle$ is the conjunction of all verdicts from instances of *mvcp* matching v and p to check whether all candidates of p have been ranked by v. By calculating disjunction on all verdicts from *collectC* matching v, $collectP\langle v\rangle$ further checks whether there exists a party to which v belongs that all candidates of p have been ranked by v. Finally, $collectV\langle\rangle$ is the conjunction of verdicts collected from all instances of *collectP* to compute the verdict for the property. Event *end* is used to trigger outputting verdicts from *mvcp*. It is also sent to *collectV* in case the trace is empty. From *mvcp* to *collectP*, each monitor sends two types of events to its downstream neighbor. One type is to count number of instances of the upstream monitor while another type carries the verdict for each instance. In this way, the downstream monitor knows whether it has already received all verdicts from its upstream monitor.

To justify the correctness of the structure above, we need to prove that 1) the generated *mvcp* instances are sufficient to check the property and 2) the structure of aggregation monitors correctly implement the semantics of nested quantifiers. For 1), instances of *mvcp* only contains all tuples of $\langle v,c,p\rangle$ satisfying the relation that v is a member of p and c is a candidate of p, which is sufficient for checking

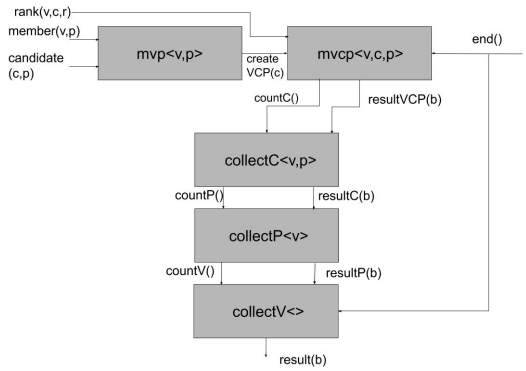

Fig. 8. Architecture for candidate selection

the property. For 2), *collectC* guarantees that given a voter and a party, the verdicts for all candidates belonging to the party are aggregated by conjunction, complying with the semantics of $\forall c$. Similarly, we could justify that *collectP* implements $\exists c$. For each voter, if there exist candidates for a party to which the voter belongs, *collectV* collects the verdict from the corresponding *collectP*. If all parties to which the voter belongs do not have candidate, *collectV* does not need to check that voter because no instance of *mvcp* is instantiated with the voter and *mvp* only contains accepting states. As a result, *collectV* implements $\forall v$ by conjunction over verdicts from all *collectP*. As mentioned above, fewer bindings are generated by SMEDL monitors than QEA, which illustrates that SMEDL has good efficiency in memory use.

Furthermore, by using the hierarchy structure, SMEDL can implement the semantics of nested quantifiers where quantified variables are related to each other using events. Two properties modified from *candidateSelection* are given below.

Example 3. For each voter and for each party that the voter is a member of, the voter must rank all candidates for that party.

Example 4. Each voter must belong to each party and he/she must rank all candidates for that party.

The same architecture illustrated in Fig. 8 can be used to monitor *Example 3*, except that *collectP* is a conjunction over verdicts from *collectC* instead of disjunction. *Example 4* is different from *Example 3* in the sense that the monitor needs to check whether each voter is bound with all parties appearing in the trace. The architecture for monitors checking *Example 4* is shown in Fig. 9. $countPFront\langle p\rangle$ and $countP\langle\rangle$ work together to count the domain of the party in the trace and send it to $collectPUniv\langle v\rangle$ (conjunction version of *collectP*) to check whether v is the member of all parties. *collectC* is created using *createVP* because the monitor needs to check whether *member* is received for all parties given each voter. Moreover, *end* triggers the output of *countP*, which triggers *collectC* to

send the verdict to *collectPUniv*. This order ensures that *collectPUniv* can get the number of parties before receiving verdicts from *collectC*.

Recall that the binding $\langle v : a, p : r, c : j \rangle : candidate(j, r)$ is generated in τ_3 for the original QEA specification. This would lead to violation of *Example 3* even if the voter *ali* does not belong to *red*. To monitor it using QEA, apart from changing existential quantifier to universal one for p, we also need to add restriction to p in the guard condition or modify EA by setting state 1 as an accepting state.

Through examples presented above, we demonstrate that SMEDL is capable of describing and checking parametric properties with nested quantifiers while generating fewer bindings. Moreover, the hierarchy of aggregation monitors is flexible to describe relation between quantified variables. As a future work, we will propose a general process to generate aggregation monitors for parametric properties.

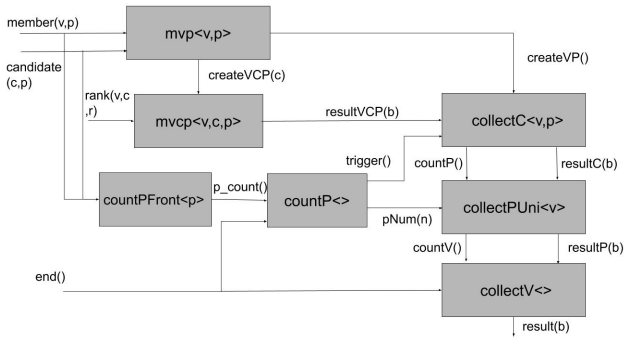

Fig. 9. Modification of architecture for candidate selection

5 Related work

Apart from MOP and QEA, there have been a considerable number of studies about handling data in RV. In [1], Allan et al. present *Tracematches* in *AspectJ* [24] to support event matching with values of parameters. *RV-monitor* [25] and *Movec* [12] use the trace slicing algorithm proposed in [11] to support parametric monitoring. In [3], Ballarin presents a generalization of the slicing algorithm in [11] to support slicing with patterns and constraints. *Larva* [14] and its derived tool *polyLarva* [13] and *Valour* [2] support parametric monitoring by dynamic creation of monitor instances. But all parameters are quantified by universal quantifier.

Several formalisms of temporal logic have been proposed for parametric monitoring such as *JLO* [30], *LTL-FO*$^+$ [20], *LTL*^{FO} [10], *MFOTL* [9], *Monitor Mod-*

ulo Theories [16]. Rule-based RV technique is expressive to support data parameterization [5, 8], from which a lot of tools and techniques have been derived such as *LogScope* [6], *TraceContract* [7], *LogFire* [22] and *data automata* [21]. There are also more research on exploring the relation between specification techniques for parametric monitoring. In [28] Reger et al. present a subset of syntactic fragments in first-order temporal logic that are sliceable and transform them into automata for slicing. In [29], a transformation from QEA to rule-based system is presented and differences between these two techniques with respect to parametric monitoring are highlighted.

In [18], Goubault-Larrecq and Olivain present *Orchids*, an intrusion detection tool. Monitors can by dynamically spawned reacting to possible beginnings of attacks. In [19], *TOPL automata* is presented based on register automata [23] for runtime verification of systems with unbounded resource generation. The key features of TOPL automata are use of registers and non-determinism. In [31], Yamagata et al. present a formalism CSP_E for monitoring concurrent systems. Parametric properties are expressed by recursive parametric processes. *Lola* [15] is a stream-based language for monitoring of synchronous systems. In [17], Lola 2.0 is presented for complex security properties. Parameterized stream templates and dynamic stream generation are added to the language to better support parametric monitoring.

6 Discussion and conclusion

In this paper, we compared the approach to parametric monitoring adopted in SMEDL with well established frameworks of MOP and QEA. Through a transformation from MOP and QEA-inspired specification to SMEDL, we showed how SMEDL can reproduce monitoring behavior of these frameworks. In addition, SMEDL does not encode quantifiers in its semantics but rather implements them as additional aggregator monitors. We note that the size of monitoring specifications in SMEDL can grow as we avoid partial instantiations with multiple monitors. We believe that we can resort to monitor templates and automatic transformation to compensate for the increased specification size. In our future work we will study whether this affects the usability of our approach. Also note that communication between monitors is necessary in our approach, which may affect the efficiency of monitoring. At the same time, communicating monitors allow us to exploit the structure of the problem through distributed deployment of monitors, improving efficiency when monitoring large-scale systems. Carefully exploring this balance is also the subject of future work.

We are formalizing the transformation algorithm from MOP and QEA to SMEDL with correctness proof. We will also formally compare the expressiveness and parametric monitoring algorithm between SMEDL and MOP, QEA and other techniques. A preliminary prototype of the method presented in this paper has been completed. However, the work to implement the tools necessary to automatically generate and deploy the monitors is still in progress.

References

1. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., De Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: ACM SIGPLAN Notices. vol. 40, pp. 345–364. ACM (2005)
2. Azzopardi, S., Colombo, C., Ebejer, J.P., Mallia, E., Pace, G.J.: Runtime verification using valour (2017)
3. Ballarin, C.: Two generalisations of roşu and chens trace slicing algorithm a. In: International Conference on Runtime Verification. pp. 15–30. Springer (2014)
4. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified event automata: Towards expressive and efficient runtime monitors. In: International Symposium on Formal Methods. pp. 68–84. Springer (2012)
5. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 44–57. Springer (2004)
6. Barringer, H., Groce, A., Havelund, K., Smith, M.: Formal analysis of log files. Journal of aerospace computing, information, and communication **7**(11), 365–390 (2010)
7. Barringer, H., Havelund, K.: Tracecontract: A scala dsl for trace analysis. In: International Symposium on Formal Methods. pp. 57–72. Springer (2011)
8. Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for run-time monitoring: from Eagle to RuleR. Journal of Logic and Computation **20**(3), 675–706 (2010)
9. Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. Journal of the ACM (JACM) **62**(2), 15 (2015)
10. Bauer, A., Küster, J.C., Vegliach, G.: From propositional to first-order monitoring. In: International Conference on Runtime Verification. pp. 59–75. Springer (2013)
11. Chen, F., Roşu, G.: Parametric trace slicing and monitoring. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 246–261. Springer (2009)
12. Chen, Z., Wang, Z., Zhu, Y., Xi, H., Yang, Z.: Parametric runtime verification of c programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 299–315. Springer (2016)
13. Colombo, C., Francalanza, A., Mizzi, R., Pace, G.J.: polylarva: runtime verification with configurable resource-aware monitoring boundaries. In: International Conference on Software Engineering and Formal Methods. pp. 218–232. Springer (2012)
14. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: International Workshop on Formal Methods for Industrial Critical Systems. pp. 135–149. Springer (2008)
15. d'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: Lola: Runtime monitoring of synchronous systems. In: Temporal Representation and Reasoning, 2005. TIME 2005. 12th International Symposium on. pp. 166–174. IEEE (2005)
16. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. International Journal on Software Tools for Technology Transfer **18**(2), 205–225 (2016)
17. Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A stream-based specification language for network monitoring. In: International Conference on Runtime Verification. pp. 152–168. Springer (2016)

18. Goubault-Larrecq, J., Olivain, J.: A smell of orchids. In: International Workshop on Runtime Verification. pp. 1–20. Springer (2008)
19. Grigore, R., Distefano, D., Petersen, R.L., Tzevelekos, N.: Runtime verification based on register automata. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 260–276. Springer (2013)
20. Hallé, S., Villemaire, R.: Runtime enforcement of web service message contracts with data. IEEE Transactions on Services Computing **5**(2), 192–206 (2012)
21. Havelund, K.: Monitoring with data automata. In: International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. pp. 254–273. Springer (2014)
22. Havelund, K.: Rule-based runtime verification revisited. International Journal on Software Tools for Technology Transfer **17**(2), 143–170 (2015)
23. Kaminski, M., Francez, N.: Finite-memory automata. Theoretical Computer Science **134**(2), 329–363 (1994)
24. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: European Conference on Object-Oriented Programming. pp. 327–354. Springer (2001)
25. Luo, Q., Zhang, Y., Lee, C., Jin, D., Meredith, P.O., Şerbănuţă, T.F., Roşu, G.: Rv-monitor: Efficient parametric runtime verification with simultaneous properties. In: International Conference on Runtime Verification. pp. 285–300. Springer (2014)
26. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. International Journal on Software Tools for Technology Transfer **14**(3), 249–289 (2012)
27. Reger, G.: Automata based monitoring and mining of execution traces. Ph.D. thesis, University of Manchester (2014)
28. Reger, G., Rydeheard, D.: From first-order temporal logic to parametric trace slicing. In: Runtime Verification. pp. 216–232. Springer (2015)
29. Reger, G., Rydeheard, D.: From parametric trace slicing to rule systems. In: International Conference on Runtime Verification. pp. 334–352. Springer (2018)
30. Stolz, V., Bodden, E.: Temporal assertions using AspectJ. Electronic Notes in Theoretical Computer Science **144**(4), 109–124 (2006)
31. Yamagata, Y., Artho, C., Hagiya, M., Inoue, J., Ma, L., Tanabe, Y., Yamamoto, M.: Runtime monitoring for concurrent systems. In: International Conference on Runtime Verification. pp. 386–403. Springer (2016)
32. Zhang, T., Eakman, G., Lee, I., Sokolsky, O.: Flexible monitor deployment for runtime verification of large scale software. In: International Symposium on Leveraging Applications of Formal Methods. pp. 42–50. Springer (2018)
33. Zhang, T., Gebhard, P., Sokolsky, O.: SMEDL: Combining synchronous and asynchronous monitoring. In: International Conference on Runtime Verification. pp. 482–490. Springer (2016)
34. Zhang, T., Wiegley, J., Giannakopoulos, T., Eakman, G., Pit-Claudel, C., Lee, I., Sokolsky, O.: Correct-by-construction implementation of runtime monitors using stepwise refinement. In: International Symposium on Dependable Software Engineering: Theories, Tools, and Applications. pp. 31–49. Springer (2018)